Computer Science and Information Systems

Computer Science and Information Systems

Published by ComSIS Consortium

Special Issue on Advances in Computer Languages, Modeling and Agents

Volume 9, Number 3
September 2012

Vol 9, No 3, September 2012

ComSIS is an international journal published by the ComSIS Consortium

ComSIS Consortium:

University of Belgrade:
Faculty of Organizational Science, Belgrade, Serbia
Faculty of Mathematics, Belgrade, Serbia
School of Electrical Engineering, Belgrade, Serbia
Serbian Academy of Science and Art:
Mathematical Institute, Belgrade, Serbia
Union University:
School of Computing, Belgrade, Serbia

University of Novi Sad:
Faculty of Sciences, Novi Sad, Serbia
Faculty of Technical Sciences, Novi Sad, Serbia
Faculty of Economics, Subotica, Serbia
Technical Faculty "M. Pupin", Zrenjanin, Serbia
University of Montenegro:
Faculty of Economics, Podgorica, Montenegro

EDITORIAL BOARD:

Editor-in-Chief: Mirjana Ivanović, University of Novi Sad
Vice Editor-in-Chief: Ivan Luković, University of Novi Sad
Managing editor: Zoran Putnik, University of Novi Sad
Managing editor: Miloš Radovanović, University of Novi Sad
Managing editor: Gordana Rakić, University of Novi Sad
Editorial Assistant: Vladimir Kurbalija, University of Novi Sad
Editorial Assistant: Jovana Vidaković, University of Novi Sad
Editorial Assistant: Ivan Pribela, University of Novi Sad
Editorial Assistant: Slavica Aleksić, University of Novi Sad
Editorial Assistant: Srđan Škrbić, University of Novi Sad

Associate editors:

P. Andreae, Victoria University, New Zealand
D. Bojić, University of Belgrade, Serbia
D. Bečejski-Vujaklija, University of Belgrade, Serbia
I. Berković, University of Novi Sad, Serbia
G. Bhutkar, Vishwakarma Institute of Technology, India
L. Böszörmenyi, University of Clagenfurt, Austria
K. Bothe, Humboldt University of Berlin, Germany
Z. Budimac, University of Novi Sad, Serbia
G. Devedžić, University of Kragujevac, Serbia
V. Devedžić, University of Belgrade, Serbia
J. Đurkovic, University of Novi Sad, Serbia
S. Guttormsen Schar, ETH Zentrum, Switzerland
P. Hansen, University of Montreal, Canada
L. C. Jain, University of South Australia, Australia
V. Jovanović, Georgia Southern University, USA
Lj. Kašćelan, University of Montenegro, Montenegro
P. Mahanti, University of New Brunswick, Canada
M. Maleković, University of Zagreb, Croatia
N. Mitić, University of Belgrade, Serbia
A. Mitrović, University of Canterbury, New Zealand
N. Mladenović, Serbian Academy of Science, Serbia
P. Mogin, Victoria University, New Zeland
S. Mrdalj, Eastern Michigan University, USA
G. Nenadić, University of Manchester, UK
Z. Ognjanović, Serbian Academy of Science, Serbia
A. Pakstas, London Metropolitan University, England
P. Pardalos, University of Florida, USA
M. Racković, University of Novi Sad, Serbia
B. Radulović, University of Novi Sad, Serbia
M. Stanković, University of Niš, Serbia
D. Tošić, University of Belgrade, Serbia
J. Trninić, University of Novi Sad, Serbia
J. Woodcock, University of Kent, England

EDITORIAL COUNCIL:

S. Ambroszkiewicz, Polish Academy of Science, Poland
Z. Arsovski, University of Kragujevac, Serbia
D. Banković, University of Kragujevac, Serbia
T. Bell, University of Cantrebury, New Zealand
S. Bošnjak, University of Novi Sad, Serbia
Ž. Branović, University of Novi Sad, Serbia
H. D. Burkhard, Humboldt University of Berlin, Germany
B. Chandrasekaran, Ohio State University, USA
V. Ćirić, University of Belgrade, Serbia
D. Domazet, Faculty of Information Technology, Belgrade, Serbia
D. Janković, University of Niš, Serbia
P. Hotomski, University of Novi Sad, Serbia
Z. Jovanović, University of Belgrade, Serbia
L. Kalinichenko, Russian Academy of Scence, Russia
Z. Konjović, University of Novi Sad, Serbia
I. Koskosas, University of Western Macedonia, Greece
W. Lamersdorf, University of Hamburg, Germany
T. C. Lethbridge, University of Ottawa, Canada
A. Lojpur, University of Montenegro, Montenegro
Y. Manolopoulos, Aristotle University, Greece
A. Mishra, Atilim University, Turkey
S. Mishra, Atilim University, Turkey
J. Protić, University of Belgrade, Serbia
D. Simpson, University of Brighton, England
D. Starčević, University of Belgrade, Serbia
D. Surla, University of Novi Sad, Serbia
M. Tuba, University of Belgrade, Serbia
P. Tumbas, University of Novi Sad, Serbia
P. Zarate, IRIT-INPT, Toulouse, France
K. Zdravkova, University of Ss. Cyril and Methodius, Skopje, FYR of Macedonia

ComSIS Editorial office:
University of Novi Sad, Faculty of Sciences,
Department of Mathematics and Informatics
Trg Dositeja Obradovica 4, 21000 Novi Sad, Serbia
Phone: +381 21 458 888; **Fax:** +381 21 6350 458
www.comsis.org; Email: comsis@uns.ac.rs

Volume 9, Number 3, 2012
Novi Sad

Computer Science and Information Systems

Special Issue on Advances in Computer Languages,
Modeling and Agents

ISSN: 1820-0214

ComSIS Journal is sponsored by:

Ministry of Education, Science and Technological Development of Republic of Serbia -
http://www.mpn.gov.rs/

Polskie Towarzystwo Informatyczne - PTI (Polish Information Processing Society) -
http://www.pti.gda.pl/

Computer Science and Information Systems

AIMS AND SCOPE

Computer Science and Information Systems (ComSIS) is an international refereed journal, published in Serbia. The objective of ComSIS is to communicate important research and development results in the areas of computer science, software engineering, and information systems.

We publish original papers of lasting value covering both theoretical foundations of computer science and commercial, industrial, or educational aspects that provide new insights into design and implementation of software and information systems. ComSIS also welcomes survey papers that contribute to the understanding of emerging and important fields of computer science. Regular columns of the journal cover reviews of newly published books, presentations of selected PhD and master theses, as well as information on forthcoming professional meetings. In addition to wide-scope regular issues, ComSIS also includes special issues covering specific topics in all areas of computer science and information systems.

ComSIS publishes invited and regular papers in English. Papers that pass a strict reviewing procedure are accepted for publishing. ComSIS is published semiannually.

Indexing Information

ComSIS is covered or selected for coverage in the following:

- Science Citation Index (also known as SciSearch) and Journal Citation Reports / Science Edition by Thomson Reuters, with 2011 two-year impact factor 0.625,
- Computer Science Bibliography, University of Trier (DBLP),
- EMBASE (Elsevier),
- Scopus (Elsevier),
- Summon (Serials Solutions),
- EBSCO bibliographic databases,
- IET bibliographic database Inspec,
- FIZ Karlsruhe bibliographic database io-port,
- Index of Information Systems Journals (Deakin University, Australia),
- Directory of Open Access Journals (DOAJ),
- Google Scholar,
- Journal Bibliometric Report of the Center for Evaluation in Education and Science (CEON/CEES) in cooperation with the National Library of Serbia, for the Serbian Ministry of Education and Science,
- Serbian Citation Index (SCIndeks),
- doiSerbia.

Information for Contributors

The Editors will be pleased to receive contributions from all parts of the world. An electronic version (MS Word or LaTeX), or three hard-copies of the manuscript written in English, intended for publication and prepared as described in "Manuscript Requirements" (which may be downloaded from http://www.comsis.org), along with a cover letter containing the corresponding author's details should be sent to official Journal e-mail.

Criteria for Acceptance

Criteria for acceptance will be appropriateness to the field of Journal, as described in the Aims and Scope, taking into account the merit of the content and presentation. The number of pages of submitted articles is limited to 25 (using the appropriate Word or LaTeX template).

Manuscripts will be refereed in the manner customary with scientific journals before being accepted for publication.

Copyright and Use Agreement

All authors are requested to sign the "Transfer of Copyright" agreement before the paper may be published. The copyright transfer covers the exclusive rights to reproduce and distribute the paper, including reprints, photographic reproductions, microform, electronic form, or any other reproductions of similar nature and translations. Authors are responsible for obtaining from the copyright holder permission to reproduce the paper or any part of it, for which copyright exists.

Computer Science and Information Systems

Volume 9, Number 3, Special Issue, September 2012

CONTENTS

Editorial

Regular Papers

983 A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools
Daniel Rodríguez-Cerezo, Antonio Sarasa-Cabezuelo, José-Luis Sierra

1019 Implementation of EasyTime Formal Semantics using a LISA Compiler Generator
Iztok Fister Jr., Marjan Mernik, Iztok Fister, Dejan Hrnčič

1045 Using Aspect-Oriented State Machines for Detecting and Resolving Feature Interactions
Tom Dinkelaker, Mohammed Erradi, Meryeme Ayache

1075 A MOF based Meta-Model and a Concrete DSL Syntax of IIS*Case PIM Concepts
Milan Čeliković, Ivan Luković, Slavica Aleksić, Vladimir Ivančević

1105 LL Conflict Resolution using the Embedded Left LR Parser
Boštjan Slivnik

1125 Indexing Ordered Trees for (Nonlinear) Tree Pattern Matching by Pushdown Automata
J. Trávníček, J. Janoušek, B. Melichar

1155 A Programming Language Independent Framework for Metrics-based Software Evolution and Analysis
Črt Gerlec, Gordana Rakić, Zoran Budimac, Marjan Heričko

1187 High-level Multicore Programming with C++11
Zalán Szűgyi, Márk Török, Norbert Pataki, Tamás Kozsik

1203 Supporting Heterogeneous Agent Mobility with ALAS
Dejan Mitrović, Mirjana Ivanović, Zoran Budimac, Milan Vidaković

1231 Language Engineering for Syntactic Knowledge Transfer
Mihaela Colhon

1249 Implementing an eXAT-based Distributed Monitoring System Prototype
Gleb Peregud, Julian Zubek, Maria Ganzha, Marcin Paprzycki

1287 Modeling a Holonic Agent based Solution by Petri Nets
Carlos Pascal, Doru Panescu

1307 Information Resource Management in an Agent-based Virtual Organization — Initial Implementation
Maria Ganzha, Adam Omelczuk, Marcin Paprzycki, Mateusz Wypysiak
1331 Decentralized Management of Building Indoors through Embedded Software Agents
Giancarlo Fortino, Antonio Guerrieri

EDITORIAL

The interest of authors to publish their contributions in ComSIS Journal is constantly increasing in recent years. At the same time, there is an evident interest of conference and workshop organizers to relate their events with recognized international journals. On the one hand side, we believe that it is a common trend in a global research community. On the other side, it was an inspiration for ComSIS editors how to attract contributions of a recognized quality and also create proper publishing capacities in ComSIS. Facing to the issue, this year we decided to keep our policy of publishing additional special issue, apart from two regular ones. The first of them, titled Advances in Computer Languages, Modeling and Agents, is in front of you.

Editors of this special issue were inspired by several events they organized during 2011 year in the following, somehow closely related domains: Advances in Programming Languages; Computer Languages, Implementations and Tools; Software Technologies for Intelligent Collaborative Systems; and Applications of Software Agents. These events included: (i) Workshop on Advances in Programming Languages (WAPL) organized within the scope of the Federated Conference on Computer Science and Information Systems (FedCSIS) in Szczecin, Poland; (ii) Symposium on Computer Languages, Implementations and Tools (SCLIT) organized within the scope of the International Conference of Numerical Analysis and Applied Mathematics (ICNAAM) in Kassandra – Halkidiki, Greece; (iii) Special Session on Software Technologies for Intelligent Collaborative Systems (STIX) organized within the scope of the International Conference on System Theory, Control and Computing (ICSTCC) in Sinaia, Romania; and (iv) Workshop on Applications of Software Agents (WASA) organized at the University of Novi Sad in Serbia. After an open call to the prospective authors to submit their papers, we received 54 submissions. After rigorous reviewing procedure, the same as for regularly submitted papers, we finally accepted 14 papers presenting both theoretical and practical contributions in field of Advances in Computer Languages, Modeling and Agents.

Daniel Rodríguez-Cerezo, Antonio Sarasa-Cabezuelo, and José-Luis Sierra in the paper "A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools" present structure-preserving coding patterns to code arbitrary non-circular attribute grammars as syntax-directed translation schemes for bottom-up and top-down parser generation tools. In order to bridge the gap between attribute grammar-based specifications and final implementations, they propose articulating the

language processor development process as the explicit transformation of the initial attribute grammar-based specification to the final implementation.

One of often emergent issues in the area of domain specific languages (DSL) is how to come to a successful implementation of a DSL in some problem domain. In the paper "Implementation of EasyTime Formal Semantics using a LISA Compiler Generator" by Iztok Fister Jr., Marjan Mernik, Iztok Fister, and Dejan Hrnčič, a DSL EasyTime is presented. It enables the controlling of an agent by writing the events within a database. The EasyTime language is implemented by a LISA tool that enables the automatic construction of compilers from language specifications, using Attribute Grammars. Let us notice that EasyTime was successfully applied for measuring time at the World Championship for the double ultra triathlon in 2009.

The next paper "Using Aspect-Oriented State Machines for Detecting and Resolving Feature Interactions", by Tom Dinkelaker, Mohammed Erradi, and Meryeme Ayache also targets the area of DSL development and implementation. The authors present an approach to manage feature interactions. They used a formal model based on the finite state machines and Aspect-Oriented technology to specify, detect and resolve features interactions. They also developed a DSL to handle finite state machines using a pattern matching technique.

Milan Čeliković, Ivan Luković, Slavica Aleksić, and Vladimir Ivančević in the paper "A MOF based Meta-Model and a Concrete DSL Syntax of IIS*Case PIM Concepts" use a meta modeling approach based on MOF to specify a meta model and a DSL for creating platform independent models of information systems. They implemented their meta model in the IIS*Case – a software tool that provides a model driven approach to information system design and generating application prototypes.

Boštjan Slivnik in "LL Conflict Resolution using the Embedded Left LR Parser" starts from the fact that choosing the right parsing method is an important issue in a design of a modern compiler. He presented a method for resolving LL(k) conflicts using small LR(k) parsers and also proposed the use of an embedded left LR(k) parser within an LL(k) parser, instead of a deterministic finite automation. As it produces the left parse it does not require rescanning of tokens already scanned or backtracking, and thus guarantees the linear parsing time for all LR(k) grammars.

Jan Trávníček, Jan Janoušek, and Borivoj Melichar in the paper "Indexing Ordered Trees for (Nonlinear) Tree Pattern Matching by Pushdown Automata" address finding occurrences of tree patterns in trees as one of the important problems with many applications. The authors presented a new kind of acyclic pushdown automata, the tree pattern pushdown automaton and the nonlinear tree pattern pushdown automaton, constructed for an ordered tree. These automata accept all tree patterns and nonlinear tree patterns, respectively,

which match the tree and represent a full index of the tree for such patterns. They analyzed timings of their implementation and showed that for a given tree the running time is linear to the size of the input pattern.

Črt Gerlec, Gordana Rakić, Zoran Budimac and Marjan Heričko in the paper "A Programming Language Independent Framework For Metrics-Based Software Evolution And Analysis" provide a framework for evaluating software metrics and analyzing software structure during software development. The main contribution of the framework is a programming language independency based on universal representation of the source code by the enriched Concrete Syntax Tree (eCST). This universality leads to consistency in change analysis and quality control during the software evolution. This characteristic gets additional value today when software projects are complex and heterogonous consisting of numerous components developed in broad spectra of languages.

Paper High-level Multi-core Programming with C++11 by Zalan Szugyi, Mark Torok, Norbert Pataki and Tamas Kozsik proposes extensions of the C++ Standard Template Library based on the features of C++11. These extensions provide enhancement of the standard library to be more powerful in the field of the multi-core programming. Approach of the authors is based on functors and lambda expressions. The contribution of the paper is illustrated by three case studies.

Developing an agent that can operate in heterogeneous network of multi-agent systems requires regeneration of the agent's executable code, as well as modifications in the way it communicates with the environment. The main goal of the paper "Supporting Heterogeneous Agent Mobility with ALAS" is providing an effective solution to the heterogeneous agent mobility problem. Following this goal a novel agent-oriented programming language, named ALAS, is proposed. Authors (Dejan Mitrović, Mirjana Ivanović, Zoran Budimac and Milan Vidaković) provide the design of the ALAS platform and an experiment to illustrate that an agent written in ALAS is able to work in truly heterogeneous networks of multi-agent systems.

In the paper "Language Engineering for Syntactic Knowledge Transfer," Mihaela Colhon presents an English-Romanian parallel treebank construction method. The method relies on a bilingual, word-aligned corpus with morphosyntactic annotations, in order to construct a syntactic annotated parallel corpus. An important fact is that the presented algorithm is defined at the abstract level of syntactic components, and is therefore language independent.

Gleb Peregud, Julian Zubek, Maria Ganzha and Marcin Paprzycki, in "Implementing an eXAT-based distributed monitoring system prototype," introduce a prototype system for monitoring resource consumption in distributed networks, such as LAN, Grid, and Cloud. A special care is

dedicated to maintaining fault-tolerance of dynamic networks, in which nodes may “disappear” at any given moment. The prototype is developed in Erlang, which proved to be a good implementation platform.

In the article “Modeling a Holonic Agent based Solution by Petri Nets,” Carlos Pascal and Doru Panescu use Petri nets to highlight certain problems in holonic manufacturing execution systems. Namely, the inadequate resource allocation and its negative consequence on agent planning are first shown by Petri net models, and then experimentally proven. The authors consider a solution based on a staff holon, in charge of conflict detection and resolution.

Next, a prototype system for resource management is proposed by Maria Ganzha, Adam Omelczuk, Marcin Paprzycki and Mateusz Wypysiak, in the paper “Information resource management in an agent-based virtual organization – initial implementation.” Their system operates in virtual organizations, matching entities in need of information resources with entities that can provide those resources in an adequate manner. The virtual organizations themselves are modeled in terms of different roles played by intelligent agents, with the roles being hierarchical ordered.

Finally, in “Decentralized Management of Building Indoors through Embedded Software Agents,” Giancarlo Fortino and Antonio Guerrieri deal with decentralized, agent-based management of smart buildings. The proposed architecture relies on wireless sensors and actuator networks for monitoring and control, and can be easily configured with any existing building. Their system has been tested in computer laboratories used by students, and was able to detect significant energy waste of idle workstations.

On behalf of the ComSIS Consortium and Editorial Board, let us express our great thanks to the reviewers and all the authors for their high-quality work and extraordinary enthusiasm. We are convinced that all of their work significantly contributes in meeting the high quality standards aiming ComSIS to a recognized international journal.

Mirjana Ivanović,
Ivan Luković,
Zoran Budimac,
Editors of the special issue

DOI:10.2298/CSIS111223022R

A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools

Daniel Rodríguez-Cerezo[1], Antonio Sarasa-Cabezuelo[1], and José-Luis Sierra[1]

[1] Computer Science School,
Complutense University of Madrid
Calle Profesor José García Santesmases, s/n
28040 Madrid, Spain
{drcerezo, asarasa, jlsierra}@fdi.ucm.es

Abstract. This article describes structure-preserving coding patterns to code arbitrary non-circular attribute grammars as syntax-directed translation schemes for bottom-up and top-down parser generation tools. In these translation schemes, semantic actions are written in terms of a small repertory of primitive attribution operations. By providing alternative implementations for these attribution operations, it is possible to plug in different semantic evaluation strategies in a seamlessly way (e.g., a demand-driven strategy, or a data-driven one). The pattern makes possible the direct implementation of attribute grammar-based specifications with widely-used translation scheme-driven tools for the development of both bottom-up (e.g. YACC, BISON, CUP) and top-down (e.g., JavaCC, ANTLR) language translators. As a consequence, initial translation schemes can be successively refined to yield final efficient implementations. Since these implementations still preserve the ability to be extended with new features described at the attribute grammar level, the advantages from the point of view of development and maintenance become apparent.

Keywords: Attribute Grammars, Parser Generators, Language Processor Development Method, Grammarware

1. Introduction

Attribute grammars were introduced by Donald E. Knuth [25] as an extension of context-free grammars for describing the syntax and semantics of context-free languages, and are widely used as a high-level specification method for the first stages of the design and implementation of a computer language [2][35].

In order to make an attribute grammar-based specification executable, it is possible to use one of the many specialized tools that support the formalism

(see, for instance,[12][17][31][33][35]). However, regardless the recognized advantages of these tools, in practice, traditional implementations of language processors are rarely based on artifacts directly generated from attribute grammars. On the contrary, attribute grammars are taken as initial specifications of the tasks to carry out, while final implementations are usually achieved by using scanner and parser generators (e.g., ANTLR, CUP, Flex, Bison…), general-purpose programming languages, or a suitable combination of the two techniques [2]. The process of transforming the initial specification into a final implementation is usually ill-defined, and typically depends solely on the programmer's art –a programmer who many times discards formal specifications while he or she directly hacks the final implementation. It seriously hinders the systematic development and maintenance of language processors.

In order to bridge the gap between attribute grammar-based specifications and final implementations, we propose articulating the language processor development process as the explicit transformation of the initial attribute grammar-based specification to the final implementation. According to our proposal, the first step to convey during the implementation stage is to explicitly code the attribute grammar in the input language of the development tool (usually, a parser generator like Bison, CUP, JavaCC or ANTLR). This will make it possible to yield an initial running implementation, which subsequently could be refined to achieve greater efficiency. In addition, since the refined implementation still supports the explicit incorporation and subsequent refinement of attribute grammar-based features, the incremental development and subsequent maintenance of the language processor can be greatly facilitated. Therefore, it is important to notice that the rationale of the present work is not to provide new methods to automatically generate language processors from attribute grammars (in this case, undoubtedly the best choice would be one of the pre-existing tools based on attribute grammars). Instead, the rationale is to start from an attribute grammar specification and then to systematically refine it across several stages, finishing with a final, highly efficient implementation in a conventional compiler construction tool -a process which is not the aim of any typical attribute grammar tool.

This paper is mainly focused on the first step of our proposal, i.e. how to code an attribute grammar in terms of the input language supported by a conventional parser generation tool, although we also illustrate some aspects of the latter refinement. In order to cover the most widely used parser generation tools, we address both bottom-up parser generators of the YACC and CUP type and top-down parser generators of the JavaCC or ANTLR style. Unlike works in L-attributed [28] or LR-attributed grammars [4] and similar approaches (e.g., [23]), our approach will support the implementation of arbitrary non-circular attribute grammars. In addition, the coding pattern will be independent of the final evaluation style chosen. Indeed, attribute grammars will be coded by using a small repertory of *attribution* operations. Finally, by providing alternative implementations for these operations, it will be possible to set up the semantic evaluation style that will finally be used.

The structure of the rest of the paper is as follows: section 2 introduces some preliminaries. Section 3 details the dependency description operations and outlines two alternative implementations, which makes apparent how to plug in different evaluation styles. Section 4 describes the coding pattern for bottom-up parser generation tools. Section 5 describes the pattern for top-down ones. Section 6 presents some work related to ours. Finally, section 7 concludes the paper and outlines some lines of future work. A preliminary version of this work, which only deals with a former pattern for bottom-up translation schemes, can be found in [41].

2. Preliminaries

In this section we introduce some basic concepts concerning the two main language-processing specification tools addressed in this paper: attribute grammars (subsection 2.1) and translation schemes (subsection 2.2).

2.1. Attribute grammars

The formalism of attribute grammars was initially proposed by Donald E. Knuth at the end of the 1960s to characterize the semantics of context-free languages [25]. Attribute grammars introduce a syntax-directed, dependency-driven language processing style. This processing style is syntax-directed because the processing of each sentence is driven by its syntactic structure, and it is dependency-driven because it is directed by the dependencies among the computations involved. Figure 1 shows an example of an attribute grammar that models the evaluation of simple arithmetic expressions, followed by declarations of constants. In the formalized process, declarations are used to build an *environment* (a set of variable-value pairs), which is subsequently used to determine the value of variables. For the sake of conciseness, only the addition operator is considered.

Attribute grammars extend *context-free grammars* with *semantic attributes* and *semantic equations*. Indeed, *context-free* grammars are standard mechanisms to define the syntax of computer languages. In a context-free grammar:

- Syntax is defined by means of *syntax rules* (or *productions*), which determine the structure of syntactic constructions in terms of sequences of simpler constructions. For instance, in Figure 1 `Sent::= Exp` **`where`** `Decs` is a syntax rule that describes the top-level structure of the kind of sentences considered in this example.
- Syntactic constructions are represented by means of *syntax symbols*: composite structures by *non-terminal* symbols and simple structures by *terminal* symbols. For instance, in Figure 1 `Sent`, `Exp` and `Decs` are non-terminal symbols that represent, respectively, sentences,

expressions and declarations. In turn, **where**, **var** or **num** are terminal symbols (these symbols represent, respectively, the *where* reserved word, variables and numbers in the language considered).

- For each non-terminal there are one or several rules defining its structure. Each rule is made up of a *left-hand side rule* (*LHS*; the non-terminal whose structure is defined) and of a *right-hand side rule* (*RHS*; the sequence of symbols which define such a structure). For instance, the previously referred to rule established that a sentence (`Sent`, the rule's LHS) maybe (the rule's RHS): an expression (`Exp`), followed by the **where** reserved word, and followed by a block of declarations (`Dec`).
- There is also a distinct non-terminal (the grammar's initial symbol or the *grammar*'s *axiom*), which represents the language's highest level structure. In Figure 1, the grammar's initial symbol is `Sent`.

```
Sent ::= Exp where Decs
    Exp.env↓ = Decs.env↑
    Sent.val↑ = Exp.val↑
Exp ::= Exp + Opnd
    Exp1.env↓ = Exp0.env↓
    Opnd.env↓ = Exp0.env↓
    Exp0.val↑ = Exp1.val↑ + Opnd.val↑
Exp ::= Opnd
    Opnd.env↓ = Exp.env↓
    Exp.val↑ = Opnd.val↑
Opnd ::= num
    Opnd.val↑ = toNum(num.lex↑)
```

```
Opnd ::= var
    Opnd.val↑ = valOf(var.lex↑,Opnd.env↓)
Opnd ::= (Exp)
    Exp.env↓ = Opnd.env↓
    Opnd.val↑ = Exp.val↑
Decs ::= Decs , Dec
  Decs0.env↑ = extendWith(Dec.env↑,Decs1.env↑)
Decs ::= Dec
  Decs.env↑ = Dec.env↑
Dec ::= var = num
  Dec.env↑ = {(var.lex↑,toNum(num.lex↑))}
```

Figure 1. An example of attribute grammar

In a context-free grammar, syntax rules enable the description of the structure of each language's sentence in terms of a tree, which is called the *parse tree* of the sentence. Inner nodes are non-terminals, while leaves are terminals. Each parent node, together with its ordered sequence of child nodes, corresponds to the application of a syntax rule. Finally, the root node corresponds to the grammar's axiom. Figure 2a shows an example of sentence in the language considered in Figure 1, and Figure 2b shows the parse tree for this sentence. Notice how this tree makes the structure of the sentence explicit. Thus, subsequent processes can be driven by this structure.

As indicated before, an attribute grammar adds a set of *semantic attributes* to the symbols of an underlying context-free grammar. These attributes will take values in the corresponding nodes of the parse trees. Attributes can be of two types:

- *Synthesized attributes*: their values are computed from synthesized attributes in the owner node's child nodes and from the inherited attributes of this owner node. Thus, the value of a synthesized attribute represents (part of) the meaning of the symbol(s) to which this attribute is

associated. In the grammar of Figure 1, synthetized attributes are terminated with ↑. Thus, `val↑` is an example of synthetized attribute in this grammar, which is used to contain the values of operands (`Opnd` non-terminal), expressions (`Exp` non terminal) and sentences (`Sent` non-terminal). In turn, the synthesized attribute `env↑` is used to build the aforementioned environment from declarations. Finally, notice that terminal symbols can also have synthesized attributes; these synthesized attributes are called *lexical attributes*, and they should be set during lexical analysis. For instance, in the grammar of Figure 1 we use a lexical attribute, `lex↑`, which contains the actual string (the *lexeme*) of each token (e.g., for **`num`** it will contain the actual number, for **`var`** the actual variable, …).

(a) `x+y+5` **`where`** `x=5,y=6`

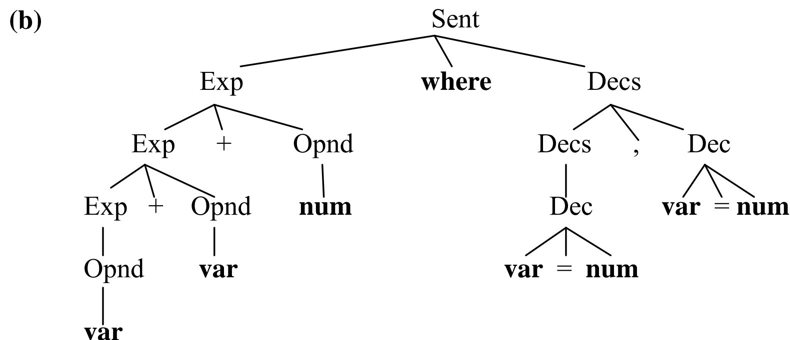


Figure 2. (a) A sentence of the language defined by the context-free grammar behind Figure 1, (b) Parse tree for the sentence in (a)

- *Inherited attributes*: their values are computed from inherited attributes in the parent and/or from synthesized attributes in the siblings. Thus, inherited attributes provide additional contextual information needed to determine the meanings of the symbols to which they are associated. In the grammar of Figure 1, we use an `env↓` inherited attribute to propagate the environment to the *expression* part of the input sentence, since this information is necessary to correctly determine the value of the constant appearing in such an expression part.

The attribute grammar will also add a set of *semantic equations* to each syntax rule. These equations will indicate how to compute the values of synthesized attributes in the rule's LHS, as well as the inherited attributes in the RHS symbols. More precisely:

- There will be exactly one semantic equation for each synthetized attribute on the LHS, and another one for each inherited attribute on the RHS.
- Each equation will apply *semantic functions* to other attributes in the rule. We will assume that, in the computation expressed by each equation, it

will only be possible to use inherited attributes from the LHS and synthesized attributes from the RHS (i.e., we will consider attribute grammars in Bochmann's normal *form* [9]).

For instance, the semantic equation `Exp`$_0$`.val`↑ = `Exp`$_1$`.val`↑ + `Opnd.val`↑ for the syntax rule `Exp ::= Exp + Opnd` in the grammar of Figure 1 establishes that, in order to compute the value of a sum (`Exp`$_0$`.val`↑)[1], it is necessary to add the value of the first operand (`Exp`$_1$`.val`↑) to the value of the second operand (`Opnd.val`↑).

Attribute grammars enable *semantic evaluation* on *attributed parse trees* (i.e., parse trees along with the semantic attributes for each node). Semantic evaluation is *dependency-driven*, since it is solely constrained by the dependencies that exist among these semantic attributes (i.e., to compute the value of an attribute, the only rule that must be obeyed is to have the values available of all the other attributes required by this computation according to a suitable semantic equation). Aside from this basic constraint, evaluation order does not matter. In consequence, attribute grammars result in a high-level specification formalism, since it is possible to specify language-processing tasks by focusing on the meaning of the syntax structures, without being distracted by lower-level implementation details, like the exact order in which attribute instances must finally be evaluated. In addition, the formalism is highly modular: it facilitates the addition of new attributes and semantic equations without affecting the existing ones, since the dependencies among attribute instances will be responsible for automatically rearranging the order in which to carry out the evaluation.

A convenient way of describing dependencies between attributes in an attributed parse tree is by means of a *dependency graph.* Nodes in this graph are the attributes in the symbols on the tree. Each arc denotes that the source attribute must be used to compute the value of the target one. Figure 3 shows the attributed parse tree and the dependency graph for the sentence in Figure 2a.

An attribute grammar is *non-circular* when it is not possible to find an attribute instance in a parse tree depending (directly or indirectly) on itself. For the contrary, the grammar is called a *circular* attribute grammar. Although semantic evaluation can be extended to manage circular attribute grammars (see, for instance [19]), for translation purposes non-circular attribute grammars usually suffice. Therefore, in this paper we will deal with non-circular attribute grammars. Semantic evaluation in these grammars can be meaningfully explained as follows [2]:

- First, find a topological order of the nodes in the dependency graph for the sentence being processed (since the grammar is non-circular, the

[1] Notice that, in order to refer to particular occurrences of a non-terminal symbol in a rule, it is possible to use subscripts: thus, Exp_0 refers to the first occurrence of Exp, Exp_1 to the second occurrence, etc.

dependency graph will be acyclical). In this order, each attribute instance will precede all the attribute instances depending upon it.

- Then, evaluate the attribute instances according to this order:

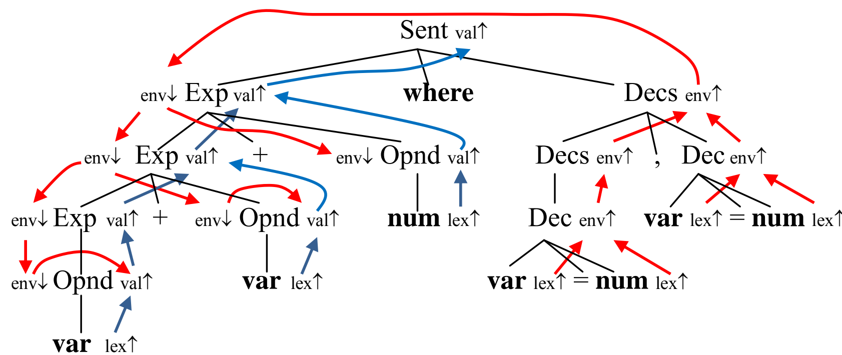


Figure 3. Attributed parse tree and dependency graph for the sentence in Figure 2a

However, it is only a conceptual execution model. In practice, semantic evaluation can be carried out by following different strategies which are only constrained by dependencies among attributes. Also, a particular evaluation strategy may not require the explicit construction of a parse tree. In fact, for remarkable subclasses of attribute grammars (many *s-attributed grammars*, which only involve synthesized attributes, and some classes of *l-attributed grammars*, in which inherited attributes of symbols only depend on the inherited attributes of their parents and synthesized attributes of their preceding siblings), it is possible to yield implementations that evaluate the attributes *on-the-fly* during parsing of the input sentence, without requiring the explicit construction of the syntax tree. Notice the grammar in Figure 1 is not s-attributed (it is needed to propagate the environment to the expression in order to evaluate it), nor l-attributed (because declarations are placed after the expression, and constant values are required to compute the value of such an expression).

2.2. Translation schemes

Translation schemes constitute another formalism that extends context-free grammar to allow the specification of syntax-directed processing [2]. For this purpose:

- Translation schemes adopt explicit visit orders for the nodes of the parse trees. Although many others are possible, two well-known visit orders are *left-to-right bottom-up* and *top-down* ones. In both of them child nodes are visited from left-to-right. However, in a bottom-up visit, nodes are visited in post-order, while in a top-down visit are visited in pre-order. In

addition, in a bottom-up visit order the visit to each node has only one *significant point*, once all its children have been visited. On the other hand, in a top-down one there are many significant points: (i) when the node is entered the first time, (ii) after a child has been exited and before the next one is entered, and (iii) when the node itself is exited.

- Translation schemes also adopt explicit ways of storing computed semantic information. For this purpose, it can be stored in semantic attributes, as in the case of attribute grammars, but also by using other means. For instance, typical execution models for bottom-up translation schemes use stacks for storing semantic information, while typical execution models for top-down ones assume implementations based on mutually recursive subprograms and use subprogram parameters and the runtime stack as a semantic storage mechanism. In addition, both bottom-up and top-down translation schemes can use global variables to facilitate some translation tasks.
- These artifacts conceive of the syntax rules as *visit plans*. For this purpose, they introduce a *semantic reference* mechanism to consult and update semantic information, as well as interleave chunks of code (*semantic actions*) at those points of the rule's RHS corresponding to significant visit points. Semantic actions will be executed each time the corresponding significant visit point is reached during the translation process. In particular, in bottom-up translators it will be possible to place a semantic action at the end of each syntax rule, while in top-down ones it will be possible to place semantic actions in any point of the rules' RHSs. In consequence, the latter will allow more natural translation patterns than the former. This is particularly true for the managing of inherited semantic information.

Although, in principle, translation schemes are independent of parser generation tools, as they can be conceived of as artifacts for processing parse trees, they are usually used as input specification formalisms for these tools. The resulting tree processors are then coupled with the parsing algorithms, and the explicit construction of the parse trees is definitively avoided. In particular:

- Bottom-up translation schemes are used as input to shift-reduce, LR parser generation tools of the YACC type (e.g., YACC, Bison, CUP, …). The resulting parsers use a stack to attach a semantic value to each syntax symbol, and they can also use global variables to manage additional semantic information. These tools constrain underlying context-free grammars to the LR type (usually, LALR(1) grammars) [2], although there are tools accepting more general grammars (e.g.,30).
- Top-down translation schemes are used as input to predictive descent parser generation tools of the JavaCC or ANTLR type. Since these tools

usually generate recursive descent parsers[2], semantic information is managed as parameters and return values of the subprograms generated, as well as in global variables, and the explicit construction of the parse tree is also avoided. These tools usually impose stronger constraints on the underlying context-free grammars: LL grammars. Although modern generation tools like ANTLR provide many useful extensions to basic LL(k) grammars (in particular, it supports the so-called LL(*) parsing method, which provides unbounded look-ahead enabled by finite-state predictors [37][38]), they are unable to manage features like left-recursion. However, as indicated before, they enable more natural mechanisms for dealing with inherited information.

Figure 4a shows an example of a bottom-up translation scheme. The language processed is the classical language of binary numbers proposed by Knuth in [25] to illustrate basic concepts in attribute grammars, and the processing task is to compute the values of the numbers. As in the other bottom-up translation schemes in this paper, we do not commit to any particular generation tool, and we do use a YACC-like notation [2] to refer to semantic values of symbols in the parse stack. Figure 4b shows a top-down, predictive-recursive translation scheme for this task. The underlying grammar is changed to LL(1), and the semantic actions are changed in consequence. Therefore, it will allow its implementation by using any of the mentioned top-down parser generation tools. As in the case of bottom-up translation schemes, we will not commit to particular generators. In addition, we will use ↓ to annotate input parameters and ↑ to annotate output ones.

(a)

```
Num ::= Num Bit {$$ := $1*2+$2}
Num ::= Bit {$$ := $1}
Bit ::= 0 {$$:=0}
Bit ::= 1 {$$:=1}
```

(b)

```
N(↑v) ::= Num(0,v)
Num(↓cv,↑v) ::= Bit(vb) RNum(vb,v)
RNum(↓cv,↑v) ::= Bit(vb) RNum(cv*2+vb,v)
RNum(↓cv,↑v) ::= {v := cv}
Bit(↑v) ::= 0 {v := 0}
Bit(↑v) ::= 1 {v := 1}
```

Figure 4. (a) An example of bottom-up translation scheme

3. The Attribute Evaluation Framework

Our coding pattern is largely based on the explicit description of the attribution structure of each grammar rule. For this purpose, we needed to develop an attribute evaluation framework, to be used in the semantic actions of the translation schemes. In this section we describe such a framework. For this purpose:

[2] It is also possible to generate non-recursive, table-driven descent parsers [2], but the mainstream in top-down parser generators is geared to the *recursive* model.

- Subsection 3.1 describes the set of basic *attribution operations* used in the translation schemes. These attribution operations make it possible to describe, for each syntax rule: (i) the dependencies between attribute occurrences in the symbols of this rule, and (ii) the functions to be used in order to compute the value of the attributes. They also make it possible to build *semantic contexts* for syntax rules (i.e., tables of references to attributes), to consult and set the value of individual attributes, and to control garbage collection.
- Subsection 3.2 introduces *semantic function managers* as the main extension points of the framework. Semantic function managers are the components used to execute semantic functions.
- Finally, subsections 3.3 and 3.4 describe two alternative implementations of the attribution operations, each based on a different *evaluation style* (a *demand-driven* style in subsection 3.3, and a *data-driven* one in subsection 3.4). In the demand-driven evaluation style, the values of attributes are computed in a lazy way, as they are required. On the other hand, in the data-driven style, values of attributes are computed in an eager way, as soon as the values of the attributes on which they depend become available. These implementations can be interchanged in a transparent way, without further changes in the translation schemes.

3.1. Attribution Operations

Table 1 outlines the repertory of basic attribution operations along with their intended meanings. As such a description makes apparent, the purpose of these operations is to provide the developer with the tools necessary to describe how the *attribute dependency graph* associated with a sentence can be built as this sentence is analyzed by the parser. In addition, it also lets the developer indicate the semantic functions for computing each attribute instance. It does not necessarily mean the graph must be fully stored in memory: depending on the actual implementation of the attribution operations, it will be possible to optimize, to a greater or lesser extent, the heap footprint, as the following subsections make apparent.

Table 1. Attribution operations

Operation	Intended Meaning
mkCtx(n)	It creates and initializes a *semantic context*: the list of attribute instances for a syntax symbol.
mkDep (a_0, a_1)	It sets a dependency between two attribute instances. Indeed, it declares that the attribute instance a_0 depends on the attribute instance a_1.
inst(a,f)	It *instruments* the attribute instance a by establishing f as the semantic function to be applied during evaluation (f is actually an integer identifier of such a semantic function)
release(as)	It invokes garbage collection on the attribute instance list as.
release(a)	It invokes garbage collection on the attribute instance a
set(a,val)	It fixes the value of the attribute instance a to val.
val(a)	It retrieves the value of the attribute instance a.

3.2. Semantic Function Managers

Before proceeding with the implementation of the attribution operations, it is convenient to introduce the concept of *semantic function manager*. In our approach, given a particular attribute grammar, the *semantic function manager* is an auxiliary component that supports the execution of semantic functions. Therefore, it is the main extension point of the evaluation framework, since it makes it possible to tailor it to each particular attribute grammar.

A semantic function manager can be conceived as a procedure that, taking the semantic function's identifier and the sequence of attribute instances as input, returns the result of applying the function to the attribute instances. It is important to remark that this component must be provided for each particular attribute grammar. Nevertheless, in our minimalistic conceptualization, we will assume this manager has the pre-established name `exec`. The implementation of this `exec` procedure will be changed from coding to coding[3].

As an example, Figure 5 depicts the pseudo-code for a semantic function manager for the grammar in Figure 1. Notice that, for each equation it is necessary to: (i) substitute attribute references in the equation's RHS for values of the semantic function manager's attribute arguments (e.g., $\texttt{Exp}_1\texttt{.val}\uparrow$ `+ Opnd.val`$\uparrow$ becomes `val(ARGS[0])+ val(ARGS[1])`, and (ii) associate a suitable integer number to the underlying semantic function (e,g., the `ADD` constant in Figure 4).

```
def IDEN=0; def ADD=1; def TONUM=2; def VALOF=3;
def EXTEND=4; def SINGLEENV=5;

procedure exec(FUN,ARGS) {
case FUN of
  IDEN →
   return val(ARGS[0]);
  ADD →
   return val(ARGS[0])+ val(ARGS[1]);
  TONUM →
   return toNum(val(ARGS[0]));
  VALOF →
   return valOf(val(ARGS[0]),val(ARGS[1]));
  SINGLEENV →
   return {( val(ARGS[0]), toNum(val(ARGS[1])) ) }
  EXTEND →
   return extendsWith(val(ARGS[0],val(ARGS[1]))
end case
}
```

Figure 5. Semantic function manager for the attribute grammar in Figure 1

[3] Although it is possible to achieve more elegant solutions by using a programming language with minimal higher-order support (e.g., a conventional object-oriented language), our conceptualization is deliberately maintained as simple as possible to preserve the essence of the evaluation approaches.

3.3. Demand-Driven Evaluation

According to the *demand-driven* evaluation style, semantic evaluation starts once the sentence has been completely parsed (see, for instance [18][29]). At this point, there is an in-memory representation of the part of the dependency graph required for performing semantic evaluation. During evaluation, the values of the attribute instances will be calculated only when they are required. For the sake of simplicity, we will ignore the detection of potential circularities in the underlying dependency graphs, although it would not be difficult to extend the framework to support it.

The first step in setting this implementation is to decide how to represent semantic attributes. For this purpose, the instances of the semantic attributes can be conceived as records. Table 2 outlines the fields required together with their intended purposes. Thus, this representation makes it possible to build a dependency structure in which:

Table 2. Structure of attribute instances in the demand-driven evaluation framework.

Field	Purpose	Initial value
value	It keeps the value of the instance of the semantic attribute.	$\perp$
available	A boolean flag that indicates whether the value is available.	`false`
deps	It keeps the links to those attribute instances required to compute the value.	*The empty list*
semFun	It stores the integer code of the semantic function required to compute the value.	$\perp$
refcount	A counter of references to this attribute instance (used to enable garbage collection).	1

- Each attribute instance points to those attribute instances required to compute it (in a similar way to the *reversed* dependency graph used in [18]).
- In addition, it explicitly stores the identifier of the semantic function to be used in this computation.

Once this representation is decided, it is possible to proceed with the coding of the operations themselves. Table 3 outlines it using pseudo-code. In this pseudo-code, references are intended to work as in Java, although we do not assume automatic garbage collection (instead, a `delete` primitive is explicitly invoked). Indeed, this is why we explicitly include `release` attribution operations.

The different operations behave as follows:

- `mkCtx` collects, in a list, as many fresh attribute instances as needed. This list actually represents a *semantic context* for a syntax symbol, since it gives access to all its semantic attributes.
- `mkDep` adds the second attribute instance in the `deps` list of the first one.
- `inst` stores the semantic function code in the `semFun` field.
- `release`, when applied to a list of semantic attribute instances, releases each instance and de-allocates the list itself.

- On the other hand, when `release` is applied to an attribute instance, it decreases its reference count by 1. If this count reaches 0, the instances on which it depends are released; finally, the original instance itself is de-allocated.
- `set` sets the `value` field and records its availability.
- `val` recovers the value of an attribute instance as follows: (i) if the value is available, it returns such a value, (ii) otherwise, it calls the semantic function manager to compute such a value and sets and returns it.

Table 3. Implementation of the attribution operations to allow a demand-driven evaluation style

Operation	Implementation	Operation	Implementation

mkCtx(*n*)	*as* := **new list**
	for i := 1 **to** *n* **do**
	  add(*as,* **new attribute**)
	end for
	return *as*
		release(*a*)	*a*.refcount:= *a*.refcount-1
			if *a*.refcount = 0 **then**
			  foreach *a'* **in** *a*.deps **do**
			    release(*a'*)
			  end foreach
			  delete *a*.deps
			  delete *a*
			end if

mkDep (a_0, a_1)	**add** (a_0.deps, a_1) a_1.refcount := a_1.refcount + 1	set(*a*,*val*)	*a*.value := *val* *a*.available := **true**

inst(*a*,*f*)	*a*.semFun := *f*	val(*a*)	**if** ¬ *a*.available **then**
			  set(*a*,
			    exec(*a*.semFun,*a*.deps))
			  release(*a*.deps)
			end if
			return *a*.value

release(*as*)	**foreach** *a* **in** *as* **do** release(*a*) **end foreach** **delete** *as*		

Thus, the demand-driven evaluation process arises from the interplay of the `val` attribution operation and the semantic function manager. Also notice how explicit garbage collection can be readily interleaved in the implementation of the attribution operation by appropriately managing the reference counters and by de-allocating lists and records as soon as they become unreachable. Although in this evaluation style, most of the dependency graph remains in memory until parsing is finished, automatic garbage collection makes it possible to de-allocate useless parts of the graph when they become unreachable. This can be due to attribute instances that are not ultimately required in any computation, or to successive evolutions of the implementation, combining pure attribute grammar features with implementation-oriented optimizations (e.g., global variables, on-the-fly evaluation of semantic attributes, …).

3.4. Data-Driven Evaluation

In the *data-driven* evaluation style, attribute instances are scheduled to be evaluated as soon as the values for all the instances on which it depends are available (see, for instance, [24]). Thus, this method can shorten the duration of attribute instances. Additionally, it can interleave evaluation with parsing. These features can be of interest while processing very long sentences, or sentences made available asynchronously (e.g., on a network communication channel). However, this method can do useless evaluations on attribute instances not required to yield the final results.

Table 4 outlines the representation of attribute instances in this case. Notice that, in addition to the list of instances on which an instance depends, the reverse relationship needs to be maintained (i.e., each attribute instance must refer to those instances which depend on it). Indeed, this representation is similar to that used by networks of *observables-observers* in the *observer* object-oriented pattern [14][4].

Table 4. Structure of attribute instances in the data-driven evaluation framework

Field	Purpose	Initial value
value	It keeps the value of the instance of the semantic attribute.	$\perp$
available	A boolean flag that indicates whether the value is available.	`false`
deps	It keeps the links to those attribute instances required to compute the value.	*The empty list*
obs	It keeps the links to those attribute instances *observing* it (i.e., which depend on it to compute their values).	*The empty list*
required	Counter which records the number of attribute instances in *deps* whose values have not yet been determined.	0
semFun	It stores the integer code of the semantic function required to compute the value.	$\perp$
instrumented	*True* if *semFun* was set, *false* otherwise.	`false`
refcount	A counter of references to this attribute instance (used to enable garbage collection).	1

Table 5 outlines the pseudo-code of the attribution operations whose implementation differs from those in the demand-driven style. This way, we only need to redefine `mkDep`, `inst`, `set` and `val`:

- In addition to updating `deps` in the first instance, `mkDep` must test whether the second instance has already been computed. If it is not available, the first instance must be added to its `obs` list, since such an instance depends on its value, which is not yet available.
- Note `inst` must take care of whether the value can be computed. Indeed, if the corresponding attribute instance has all the instances on which it depends computed, it can thereby be computed. It assumes the

[4] As with the demand-driven style, this representation could be simplified by inferring the values of flags (in this case, *available* and *instrumented*) from the other fields. However, we prefer to explicitly preserve these flags to increase the readability of pseudo-code.

establishment of all the required dependencies before instrumentation, which is ensured by our coding pattern.
- `Set` must take care to decrement the `required` counters in all the instances depending on the current one. In addition, if a counter reaches 0, it must force the evaluation of the corresponding instance.
- Finally, `val` immediately computes the value, unless the instance has not yet been instrumented.

Notice how, in this case, evaluation can be interleaved with parsing. Indeed, evaluation is fired when the values of attribute instances are explicitly set, and also when attributes are instrumented. In consequence, garbage collection also interplays with parsing, and, therefore, this method can mean less heap usage. However, this method assumes all the semantic functions used are *strict,* in the sense that all their arguments must be evaluated before they are applied. On the contrary, the demand-driven method described in the previous subsection also supports *non-strict* functions, in which the way of evaluating the arguments can differ from function to function.

Table 5. Implementation of the attribution operations to allow a data-driven evaluation style (only those implementations differing from Table 3 are presented)

Operation	**Implementation**	**Operation**	

mkDep (a_0, a_1)	**add** (a_0.deps, a_1)
	a_1.refcount := a_1.refcount + 1
	if ¬ a_1.available **then**
	  add (a_1.obs, a_0)
	  a_0.required := a_0.required + 1
	  a_0.refcount := a_0.refcount + 1
	end if
		set(a,val)	a.value := val
			a.available := **true**
			foreach a’ **in** a.obs **do**
			  a’.required := a’.required – 1
			  if a’.required = 0 **then**
			    val(a’)
			  end if
			end foreach
			release(a.obs)

inst(a,f)	a.semFun := f
	a.instrumented := **true**
	if a.required = 0 **then**
	  val(a)
	end if
		val(a)	**if** ¬ a.available ∧
			    a.instrumented **then**
			  set(a, exec(a.semFun,a.deps))
			  a.available := **true**
			  release(a.deps)
			end if
			return a.value

4. A Coding Pattern for Bottom-up Parser Generation Tools

In this section we introduce a coding pattern for bottom-up parser generation tools. In this way:
- In order to keep the translation scheme as independent as possible of changes in the attribute grammar’s semantic part, we will promote an intermediary representation of the attribute grammar based on *attribution functions* (subsection 4.1). For this purpose, with each rule will be assigned a function that takes the semantic contexts of the rule’s RHS as arguments and builds and returns the semantic context for the rule’s LHS. In addition,

using the basic attribution operations introduced in the previous section, attribution functions establish dependencies among attributes, associate semantic functions with attributes as necessary, and control garbage collection.
- Then, these functions will be used in the actions of the resulting bottom-up translation scheme (subsection 4.2). More precisely, the semantic action associated with each rule will invoke the attribution function for this rule with the suitable set of arguments.
- The analysis of the memory footprint required by the overall method will be depicted in subsection 4.3 by considering both the demand-driven and the data-driven evaluation styles.
- Finally, subsection 4.4 briefly illustrates some potential refinements of the initial implementation. These refinements will be oriented to anticipate the computation of inherited attributes by using *marker non-terminals* (i.e., new non-terminals defined by rules with empty RHS), and to simplify implementation by means of global variables.

4.1. The attribution functions

The implementation of the attribute grammar using a bottom-up parser generation tool can be naturally thought of as the *bottom-up* construction of the attribute dependency graph for each processed sentence using basic attribution operations. In this construction, the dependency graph for a syntactic structure is built by taking the dependency graphs of the substructures as building components. Thus, the process can be facilitated by introducing a set of *attribution functions*, which, for each rule in the grammar, take cares of this construction. These attribution functions will be used to set up the semantic actions of the bottom-up translation scheme that feeds the parser generation tool. Therefore, the set of attribution functions can be conceived of as the implementation of a sort of *abstract* version of the attribute grammar, which subsequently can be attached to a concrete syntax by using a suitable translation scheme.

Each attribution function takes the semantic contexts of the symbols in the rule's RHS as input, and it outputs the semantic context for the LHS non-terminal using basic attribution operations. In order to do so, it is possible to apply the following guidelines:
- First at all, we need to create the semantic context for the LHS. This is done by using an `mkCtx` operation. We only need to indicate the number of semantic attributes for the LHS non-terminal.
- Next, we need to describe the dependencies among the attribute instances. Such dependencies are directly determined by examining the semantic equations, and they must be stated by using the `mkDep` operation.
- Once this has been done, it is necessary to *instrument* the synthesized attribute instances in the rule's LHS, as well as the inherited attribute

instances of the RHS symbols. Once more, the code is straightforward: an `inst` operation for each equation. Notice we need to code the semantic functions with integer identifiers, which can be interpreted by the semantic function manager.

- Finally, we need to release the attribute instance lists for the symbols in the rule's RHS.

This process can be further facilitated by using a procedure establishing the corresponding dependencies for each attribute as well as the instrumentation. This procedure, which will be called `eq` (since it actually serves to represent semantic equations), is sketched in Figure 6. Finally, notice that, although we need to provide an attribution function for each rule in the grammar, the same function can be shared by several rules. Therefore, in addition to contributing to more readable translation schemes, attribution functions also make it possible to reuse common attribution patterns. Indeed, it is possible to provide attribution functions with additional parameters in order to increase the reuse degree.

```
procedure eq(lhsAtr,rhsAtrs,semFun) {
   foreach rhsAtr in rhsAtrs
     mkDep(lhsAtr,rhsAtr)
   end foreach
   inst(lhsAtr,semFun)
}
```

Figure 6. The `eq` procedure

As an example, Figure 7 depicts the attribution functions for the attribute grammar in Figure 1. For instance, the `addition` function codes the attribution for the rule `Exp ::= Exp + Opnd` in the grammar of Figure 1 as follows:

- Since `Exp`, the rule's LHS, has two semantic attributes (`env` and `val`), we need to invoke `mkCtx` with 2 as the number of attributes to be allocated.
- From the first equation, we get $Exp_1.env$ depends on $Exp_0.env$. In addition, the semantic function to be applied is the identity. Therefore, the equation is coded by `eq(Exp1[env], (Exp0[env]),IDEN)`.
- The other equations are coded in a similar manner. For instance, the equation $Exp_0.val = Exp_1.val + Opnd.val$ is coded by `eq(Exp0[val], (Exp1[val],Opnd[val]),ADD)`. Notice that, for each equation, it is important to establish the dependencies in the order in which the attribute references appear in its RHS, and therefore it must be taken into account in the coding of each equation.
- Finally, we include a `release` action for each symbol in the rule's RHS having semantic attributes.

Concerning the allocation of lexical attribute instances, it must be performed by the scanner, which will return the corresponding attribute instance list using a suitable field in the token.

```
def env=0; def val=1; def vs=0; def lex=0;
function init(Exp,Decs) {
   Sent := mkCtx(1)
   eq(Sent[vs], (Exp[val]),IDEN)
   eq(Exp[env],(Decs[env]),IDEN)
   release(Exp)
   release(Decs)
   return Sent
 }
function addition(Exp1,Opnd){
   Exp0 := mkCtx(2)
   eq(Exp1[env], (Exp0[env]),IDEN)
   eq(Opnd[env], (Exp0[env]),IDEN)
   eq(Exp0[val],
        (Exp1[val],Opnd[val]),ADD)
   release(Exp1)
   release(Opnd)
   return Exp0
 }
 function chain(Child) {
  Parent := mkCtx(2)
  eq(Child[env],(Parent[env]),IDEN)
  eq(Parent[val],(Child[val]),IDEN)
  release(Child)
  return Parent
}
function num(num) {
   Opnd := mkCtx(2)
   eq(Opnd[val], (num[lex]),TONUM)
   release(num)
   return Opnd
 }
function var(var) {
   Opnd := mkCtx(2)
   eq(Opnd[val],
       (var[lex],Opnd[env]),VALOF)
   release(var)
   return Opnd
 }
function mutiEnv(Dec,Decs1) {
  Decs0 = mkCtx(1)
  eq(Decs0[env],
     (Dec[env],Decs1[env]),EXTEND)
  release(Dec)
  release(Decs1)
  return Decs0
}
function singleEnv(Dec) {
  Decs = mkCtx(1)
  eq(Decs[env],(Dec[env]),IDEN)
  release(Dec)
  return Decs
}
function entry(var,num) {
  Dec = mkCtx(1)
  eq(Dec[env],
      (var[lex],num[lex]),SINGLEENV)
  release(var)
  release(num)
  return Dec
}
```

Figure 7. Attribution functions for the attribute grammar in Figure 1

4.2. The bottom-up translation scheme

In order to finish the coding, it is necessary to provide a suitable translation scheme. It can be done in a straightforward way, by using the attribution function that corresponds to each rule. Indeed, for each syntax rule A ::= α in the grammar, we only need to add a rule A::= α {$$:= $\phi(\$_\alpha)$} to the translation scheme. Here, ϕ is the attribution function for A::= α, and $\$_\alpha$ denotes the list of RHS semantic contexts. This pattern makes further advantages to using attribution functions apparent, instead of directly coding the semantic equations in the rule's actions (like we did in our previous work [41]): the concrete syntax can be readily changed without changing the attribution functions (which, as indicated before, are actually the implementation of an abstract version of the original attribute grammar).

Figure 8 exemplifies the coding pattern by showing the bottom-up translation scheme that implements the attribute grammar of Figure 1. Coded in the input language of a tool like YACC, Bison or CUP, and with a suitable implementation of the attribution functions and the basic attribution operations, it can be automatically turned onto a running implementation.

```
Sent ::= Exp where Decs      {$$ := init($1,$3)}
Exp ::= Exp + Opnd           {$$ := addition($1,$3)}
Exp ::= Opnd                 {$$ := chain($1)}
Opnd ::= num                 {$$ := num($1) }
Opnd ::= var                 {$$ := var($1) }
Opnd ::= (Exp)               {$$ := chain($2) }
Decs ::= Dec, Decs           {$$ := multiEnv($1,$3)}
Decs ::= Dec                 {$$ := singleEnv($1)}
Dec  ::= var = num           {$$ := entry($1,$3)}
```

Figure 8. Bottom-up translation scheme for the attribute grammar in Figure 1

4.3. Analysis of the method

The efficiency of the language processor generated will be manifested in the memory footprint of the recognition and evaluation process, which will in turn depend on the evaluation strategy used and on the kind of the initial attribute grammar:

- If the implementation uses the demand-driven evaluation style, it will incur in the highest amount of auxiliary memory required by the method. Indeed, the memory usage will be rather independent of the kind of the grammar, and proportional to the length of the input sentences. Indeed, the dependency graph will be almost entirely built before evaluation is initiated, and the process will be divided into two well differentiated phases: (i) a first one in which the input sentence is recognized and the dependency graph is built, and (ii) a second one in which the attribute values are computed.
- If the implementation uses the data-driven evaluation style, the performance will be optimal for s-attributed grammars. Indeed, the values of the attributes will be computed as soon as they are instrumented, and the amount of additional memory required for semantic evaluation will remain constant. However, in the presence of inherited information, the evaluation will be delayed until this information is *injected* into the process. The worst case happens when the overall evaluation process depends on inherited information to be set up in the grammar's initial symbol. In this case, most of the dependency graph must be built before initiating evaluation, and thus the method becomes equivalent to using a demand-driven strategy.

This analysis does not mean the method does not provide good (even nearly optimal) solutions for non s-attributed attribute grammars, since inheritance is not required to be global. For instance, for grammars like that of the example, the method, in combination with a data-driven evaluation style, yields not only nearly optimal, but also elegant implementations.

4.4. Refinements

Once the initial coding is available, the initial implementation can be systematically refined in an efficient implementation by using well-known techniques for dealing with inherited information during bottom-up parsing. In particular:

(a)
```
Sent ::= Mo Exp where Decs   {$$ := init($1,$2,$4) }
Mo ::=                       {$$ := mkEnv() }
Exp ::= Exp + Opnd           {$$ := addition($1,$3)}
Exp ::= Opnd                 {$$ := chain($1)}
Opnd ::= num                 {$$ := num($1) }
Opnd ::= var                 {$$ := var($1,$0) }
Opnd ::= (M1 Exp)            {$$ := chain($3) }
M1 ::=                       {$$ = $-1}
Decs ::= Dec, Decs           {$$ := multiEnv($1,$3)}
Decs ::= Dec                 {$$ := singleEnv($1)}
Dec ::= var = num            {$$ := entry($1,$3)}
```

(b)
```
...
function mkEnv() {
   return mkCtx(1)
}
...
function init(ExpEnv,Exp,Decs) {
   Sent := mkCtx(1)
   eq(Sent[vs], (Exp[val]),IDEN)
   eq(Exp ExpEnv[env],(Decs[env]),IDEN)
   release(ExpEnv)
   release(Exp)
   release(Decs)
   return Sent
}
...
```

```
function addition(Exp1,Opnd){
   Exp0 := mkCtx(2 1)
   eq(Exp1[env], (Exp0[env]),IDEN)
   eq(Opnd[env], (Exp1[env]),IDEN)
   eq(Exp0[val],
       (Exp1[val],Opnd[val]),ADD)
   release(Exp1)
   release(Opnd)
   return Exp0
}
...
function var(var, Env) {
   Opnd := mkCtx(2 1)
   eq(Opnd[val],
       (var[lex],Opnd Env[env]),VALOF)
   release(var)
   return Opnd
}
```

Figure 9. (a) Refinement of the translation scheme in Figure 8 by means of marker non-terminals; (b) modification of some attribution functions and the addition of a new one (erased code appears in strikethrough light-gray text, and new added coded appears shaded)

- Use of *marker non-terminals* (i.e., new non-terminal symbols defined by empty rules [2]) to mark the beginning of *left spines* (i.e., chains of elements generated by left-recursion). These non-terminals can store inherited attributes to which can be accessed from any point of the left spines without requiring explicit propagation. Using this technique, it is possible to deal with many l-attributed grammars with bounded memory footprint. The technique can be applied to the implementation exemplified before, yielding the translation scheme of Figure 9a. In this refinement it is possible to eliminate the inherited environment, since it can be remotely stored in the marker symbol **`Mo`** and referred from the marker symbol **`M1`**. In addition, the marker contexts can be passed on as an additional argument to the `var` attribution function. In Figure 9b we show the new attribution function `mkEnv` and how the old attribution functions `init`,

`addition` and `var` must be modified to fit in the new refinement. The other attribution functions can be modified in an analogous way, and therefore they will be omitted here.

- Use of global state. In order to integrate this global state in the evaluation machinery, it is possible to create *views* of this state as semantic attributes. The technique can be illustrated with the example discussed above, since the environment can be completely managed as a global variable. Thus, all the machinery concerning propagation of environments can be completely eliminated. Figure 10a shows the resulting translation scheme. Notice how the environment is managed as a global variable, and is also exposed as a globally accessible semantic attribute. With the exception of `init` (see Figure 10b), the attribution functions coincide with those used in the refinement sketched in Figure 9

(a)

```
global env = ∅
global aenv = mkCtx(1)
procedure addEntry(env,Var,Num) {
  env := extendWith({(val(var[lex]),
                      toNum(val(Num[lex])))},env)
}

Sent ::= Exp where Decs  {set(aenv[env],env); $$ := init($1); release(aenv); }
Exp ::= Exp + Opnd       {$$ := addition($1,$3)}
Exp ::= Opnd             {$$ := chain($1)}
Opnd ::= num             {$$ := num($1) }
Opnd ::= var             {$$ := var($1,aenv) }
Opnd ::= (Exp)           {$$ := chain($2) }
Decs ::= Dec, Decs       {}
Decs ::= Dec             {}
Dec  ::= var = num       {addEntry(env,$1,$3)}
```

(b)

```
function init(Exp) {
   Sent := mkCtx(1)
   eq(Sent[vs], (Exp[val]),IDEN)
   release(Exp)
   return Sent
}
```

Figure 10. (a) Use of a global environment to simplify the translation scheme in Figure 8; (b) the `init` attribution function in this refinement.

5. A Coding Pattern for Top-Down Parser Generation Tools

This section describes the coding pattern for top-down parser generation tools. For this purpose, it follows a similar structure to that of the previous one:

- Subsection 5.1 describes the structure of attribution functions in this pattern. In one sense, these attribution functions arose by *reversing* the bottom-up ones. Now, each attribution function takes the semantic context

of the LHS as argument, and it builds and returns the semantic contexts for each symbol in the RHS. As in the bottom-up cases, they also use the basic attribution operations to set up all the attribute evaluation machinery.

- Subsection 5.2 describes the general guidelines to code the translation scheme. As in the bottom-up case, it is carried out by placing attribution functions at strategic points in the syntax rules.
- Subsection 5.3 describes how to deal with underlying non-LL grammars. Indeed, bottom-up parser generation tools usually deal with predictive grammars of the LL-type, in which it is possible to determine which rule to expand by using a finite amount of input look-ahead. However, some grammatical features (e.g., left-recursion, common left-factors) destroy this capability to predict the rule to be applied. Fortunately, many of these grammars can be systematically transformed to forms suitable for top-down parsing. These transformations must be accompanied by the transformation of the semantic part, however. Thus, we researched how to perform these transformations for the case of our encoding scheme.
- As in the bottom-up case, subsection 5.4 briefly analyzes the method, and subsection 5.5 describes some subsequent refinements (the most prominent one deals with the systematic replacement of recursion by iteration in the resulting translation schemes).

5.1. The attribution functions

Although it is possible to undertake implementation by thinking of the bottom-up construction of the attribute dependency graph, as in the bottom-up case, it is possible to obtain more advantages if we think of the *top-down* construction of this graph. In particular, it will facilitate the propagation of inherited information during parsing.

```
function addition(Exp0){
   Exp1 := mkCtx(2)
   Opnd := mkCtx(2)
   eq(Exp1[env], (Exp0[env]),IDEN)
   eq(Opnd[env], (Exp0[env]),IDEN)
   eq(Exp0[val],
        (Exp1[val],Opnd[val]),ADD)
   release(Exp0)
   return (Exp1,Opnd)
 }
```

Figure 11. Top-down geared version of the attribution function `addition`

To enable the top-down construction of the dependency graph, we need to reverse the flow of semantic contexts in the attribution functions. Now, these functions will take the LHS context as input and it will return the RHS contexts as output. Thus, a typical attribution function begins by creating the RHSs contexts. Then it establishes the dependencies between attributes and instruments the attributes as in the bottom-up case. Finally, it releases the LHS context. Figure 11 exemplifies it by showing the top-down geared

version of the `addition` attribution function. The other attribution functions can be adapted in a similar way, and therefore they will be not detailed here.

5.2. The top-down translation scheme

As in the bottom-up case, the coding of the translation scheme is carried out in terms of the attribution functions. In addition, due to the inversion of the flow of semantic contexts in the attribution functions, it is necessary to connect the terminal contexts created in these functions to the contexts created by the scanner. This can be done by using the `conn` procedure sketched in Figure 12 (the name is an abbreviation for *connect*).

```
procedure conn(termCtx,lexCtx) {
   eq(termCtx[lex],(lexCtx[lex]),IDEN)
   release(termCtx); release(lexCtx)
}
```

Figure 12. Procedure for connecting terminal contexts.

Thus, for each syntax rule A::=X_0 .. X_n in the grammar, we need to add a rule A(↓ctxA)::={(ctx_0,…,ctx_n):=ϕ(ctxA) } I_0 … I_n where: (i) ϕ is the rule's attribution function, (ii) (ctx_0,…,ctx_n) collects the RHS contexts (this assignment is optional; it can be omitted if the attribution function does not return any context), and (iii) each I_i is X_i(ctx_i) if X_i is a non-terminal, X_i($lexctx_i$) {`conn(`ctx_i`,`$lexctx_i$`)`} if it is a terminal with semantic charge, or X_i if it is a terminal without semantic charge (a keyword, a punctuation symbol, etc.). These guidelines are illustrated in Figure 13, which shows the top-down translation scheme for the grammar in Figure 1.

```
Sent(↓co) ::= {(c1,c2) := init(co)} Exp(c1) where Decs(c2)
Exp(↓co)  ::= {(c1,c2) := addition(co)} Exp(c1) + Opnd(c2)
Exp(↓co)  ::= {c1 := chain(co)} Opnd(c1)
Opnd(↓co) ::= {c1 := num(co) } num(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := var(co) } var(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := chain(co) } (Exp(c1))
Decs(↓co)  ::= {(c1,c2) := multiEnv(co)} Dec(c1) , Decs(c2)
Decs(↓co) ::= {c1 := singleEnv(co)} Dec(c1)
Dec(↓co) ::= {(c1,c2) := entry(co)} var(lc1) {conn(c1,lc1)} = num(lc2) {conn(c2,lc2)}
```

Figure 13. Top-down translation scheme for the attribute grammar in Figure 1 (warning: this translation scheme is not yet implementable with a top-down parser generator!)

Unfortunately, since top-down translators usually require LL underlying context-free grammars, translation schemes obtained according to the stated guidelines can require further transformation before allowing their implementation during parsing. In particular, the context-free grammar of the translation scheme in Figure 1 exhibits left-recursion, which make this coding

unsuitable for top-down parser generation. Next subsection deals with this problem.

5.3. Factoring and immediate left-recursion elimination

In many cases the problematic top-down translation schemes and the associated attribution functions can be systematically tuned by applying similar patterns to the well-known factoring and left-recursion elimination transformations presented in any compiler construction textbook [2]. In particular:

- Figure 14a sketches a transformation pattern for removing common factors in a rule-set. Notice this transformation supposes the explicit construction of the common factor's semantic context. It will be carried out by a *context-construction* function (denoted by $\texttt{mkCtx}_\alpha$ in Figure 14a). In addition, it is necessary to keep this context alive, regardless whether it will be released in the common factor. For this purpose, we need to create another twin context ($c_{\alpha p}$ in Figure 14a), and to connect it to the actual common factor's semantic context. This connection is achieved with a *context connection procedure*, denoted by $\texttt{conn}_\alpha$ in Figure 14a. Finally, it will require explicitly modifying the attribution functions for each rule affected. The modified attribution functions (denoted by ϕ'_i in Figure 14a) do not need to create the semantic context for the common factor; instead, they will take it as a parameter.
- Figure 14b shows a transformation pattern for removing immediate left-recursion. The pattern requires the explicit construction of the context for the recursive non-terminal, which is achieved by using a context-construction function ($\texttt{mkCtx}_\texttt{A}$ in Figure 14b). As usual, the chain generated by left-recursion in the original grammar is generated by using right-recursion in the transformed one. Each stage of this right-recursive process can be associated with a stage in the bottom-up construction of the parse tree in the original grammar. Therefore, it is possible to take the context associated to the root of the already constructed sub-tree as input, and then to modify the corresponding attribution function to take this as an additional argument instead of creating it (the modified functions are noted ϕ'_i in Figure 14b, and they must take care of releasing the semantic context once they are not necessary). In addition, it is necessary to provide a context connection procedure for performing the connection between the input and the last context created once the right-recursion is finished (it is denoted by $\texttt{conn}_\texttt{A}$ in Figure 14b).

Figure 15 illustrates the application of these patterns to the translation scheme of Figure 13. The grammar of the transformed scheme is LL(1) and, therefore, suitable for its implementation in any of the top-down parser generation tools mentioned.

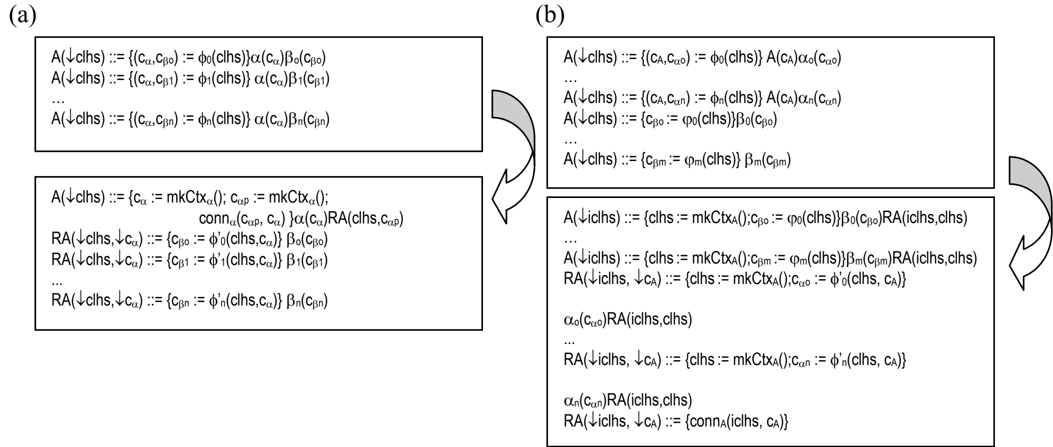


Figure 14. (a) Factoring pattern; (b) Immediate left-recursion elimination pattern

```
function mkCtxExp() {return mkCtx(2)}
function mkCtxDec() {return mkCtx(1)}
procedure connExp(ic,c) {eq(c[env],(ic[env),IDEN); eq(ic[val],(c[val]),IDEN) }
procedure connDecs(cp,c) {eq(cp[env],(c[env),IDEN);}

Sent(↓co) ::= {(c1,c2) := init(co)} Exp(c1) where Decs(c2)
Exp(↓ic)  ::= {co := mkCtxExp(); c1 := opnd(co)} Opnd(c1) RExp(ic,co)
RExp(↓ic, ↓c1) ::= {co := mkCtxExp(); c2 := addition(co,c1)} + Opnd(c2) RExp(ic,co)
RExp(↓ic, ↓co) ::= {connExp(ic,co)}
Opnd(↓co) ::= {c1 := num(co) } num(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := var(co) } var(lc1) {conn(c1,lc1)}
Opnd(↓co) ::= {c1 := chain(co) } (Exp(c1))
Decs(↓co)  ::= {c1 := mkCtxDec(); c1p := mkCtxDec(); connDecs(c1p,c1) }
                                                          Dec(c1) RDecs(co,c1p)
RDecs(↓co, ↓c1) ::= {c2 := multiEnv(co,c1)} , Decs(c2)
RDecs(↓co, ↓c1) ::= {singleEnv(co,c1)}
Dec(↑co) ::= {(c1,c2) := entry(co)} var(lc1) {conn(c1,lc1)} = num(lc2) {conn(c2,lc2)}
```

Figure 15. Result of eliminating common factors and immediate left-recursion in the top-down translation scheme of Figure 13 (the transformed parts are shadowed) in order to obtain an artifact implementable with a top-down parser generator.

5.4. Analysis of the method

As in the bottom-up case, the use of a demand-driven evaluation style will imply explicitly constructing dependency graphs, and therefore the highest memory overhead. As in bottom-up implementations, it can be alleviated by using data-driven evaluation. In this case, the method will incur in the lowest auxiliary evaluation memory overhead for l-attributed grammars. Indeed, for these grammars, data-driven evaluation will yield a behavior equivalent to a one-pass, on-the-fly translation process.

Finally, since the initial coding encourages the explicit coding of the plain, BNF grammar, the resulting translators will be highly recursive, which should be taken into account if the final implementation language does not support tail recursion optimization. Fortunately, as will be indicated in the next section, by using EBNF notation in the underlying context-free grammars, it will be possible to easily turn many right-recursions into iteration.

5.5. Refinements

As in the bottom-up case, it is possible to use global state to simplify the propagation of context. Nevertheless, due to the nature of top-down translators, this refinement is less critical from a performance perspective. Concerning the use of marker non-terminals, it is nonsense in this scenario.

However, as indicated in the previous subsection, an interesting refinement would be to exploit the support of EBNF notation provided by typical predictive recursive parser generation tools in order to overcome the potential stack overflow problem associated with the recursive implementation of genuinely iterative processes[5]. Indeed, it is equivalent to performing a tail-recursion optimization process by hand[6].

In addition, it is possible to carry out several simplifications oriented to minimizing the use of temporary variables (e.g., by passing complex expressions as parameters to non-terminal symbols).

```
Sent(↓co) ::= {(c1,c2) := init(co)} Exp(c1) where Decs(c2)
Exp(↓ico)  ::= {co := mkCtxExp()} Opnd(chain(co)) RExp(ico,co)
RExp(↓ic, ↓c1) ::= ({co := mkCtxExp()} + Opnd(addition(co,c1)) {c1:=co})*
                                                          {connExp(ic,c1)}
Opnd(↓co) ::= num(lc1) {conn(num(co),lc1)}
Opnd(↓co) ::= var(lc1) {conn(var(co),lc1)}
Opnd(↓co) ::= ( Exp(chain(co)) )
Decs(↓co)  ::= {c1 := mkCtxDec(); c1p := mkCtxDec(); connDecs(c1p,c1) }
                                                      Dec(c1) RDecs(co,c1p)
RDecs(↓co, ↓c1) ::= ({co := multiEnv(co,c1)} ,
                    {c2 := mkCtxDec(); c1 := mkCtxDec(); connDecs(c1,c2) }
                     Dec(c2))* {singleEnv(co,c1)}
Dec(↑co) ::= {(c1,c2) := entry(co)} var(lc1) {conn(c1,lc1)} = num(lc2) {conn(c2,lc2)}
```

Figure 16. Refinement of the translation scheme in Figure 15

Figure 16 exemplifies the result of applying these refinements on the translation scheme of Figure 15. The resulting scheme can be readily implemented on any typical recursive predictive parser generation tool (e.g., JavaCC or ANTLR), or directly by hand in a general-purpose programming language. As this example makes apparent, after applying this refinement,

[5] Notice this problem does not affect bottom-up parsers, provided sequences are represented by means of left-recursion.

[6] Indeed, it could be possible to directly formulate the immediate left-recursion elimination pattern in iterative terms.

recursion will only be used to express nesting (in the example, it is due to the use of parenthesis in expressions), which constitutes the most natural use of this grammar feature.

6. Related Work

As indicated in the introduction, the standard way of implementing an attribute grammar is to use one of the tools that directly supports the formalism. Indeed, as [35] makes apparent, since its invention by Knuth at the end of the sixties of the past century, the computer language community has proposed many of these tools, starting with classical systems like GAG [22], FNC-2 [20], ELI [15] or Elegant [7], and ending with recent proposals like LISA [17][31][33], Silver [51] or JastAdd [29]. These tools take attribute grammars as input, and generate operative language processors as output. In addition, they support metalanguages by adding many extensions to the basic formalism (e.g., modules [21], generics [42], higher-order [48], object [16] and aspect orientation [39][40], etc.), which facilitate the production and maintenance of complex specifications.

Attribute grammar-based systems as the abovementioned promote orchestrating the development entirely in terms of attribute grammars, and, in particular, in terms of the metalanguages supported. On the contrary, the goal of our approach is not to provide yet another attribute grammar system, but to propose systematic ways of integrating attribute grammars in conventional language implementation processes, by using conventional parser generation tools. In this way, in our approach attribute grammars are used at the initial stages of the development process, as a formal specification tool. In addition, our work promotes an initial design-preserving coding in a conventional parser generation tool, in the form of a suitable translation scheme. Beyond this point, the development process proceeds through several refinements, making use of the parser generation tool facilities and the tool's target implementation language.

In consequence, our approach promotes straightforward coding patterns, which can be applied by hand to get initial codings, and which make it possible to identify the different pieces of the original attribute grammar in these codings. On the other hand, the code generated by an attribute grammar-based tool is usually a highly optimized artifact, usually generated following a *static* approach in which evaluation and storage strategies are determined as the result of a static analysis of the input grammar [1], and which is not intended to be inspected and modified by humans.

In addition, our approach is oriented to converge with conventional development processes. Because of it, on one hand we encourage the use of semantic evaluation methods that can be easily coupled with parsing. This is not necessarily true for attribute grammar-based tools, many of which promote final implementations that operate on (concrete or abstract) syntax trees. Of course the patterns described in this paper could be automated in

the form of attribute-grammar based tools. Indeed, tools for the processing of XML based on attribute grammars like those described in [43] are inspired by these patterns (in particular, these tools use the data-driven evaluation strategy to make the stream-oriented, asynchronous, processing of very wide XML documents possible). These tools could be used as a sort of CASE support during the development process model promoted in this paper, which in turn could imply the provision of some roundtrip support (see the future work description in the next section).

The coupling of attribute evaluation and parsing has been extensively addressed as a way of implementing restricted classes of attribute grammars (see, for instance, [3] for a tutorial introduction). The works in [2][3] show how l-attributed grammars with underlying LL grammars can be implemented during top-down predictive descent parsing. In addition, different classes of LR-attributed grammars have been identified, which allow semantic evaluation to be implemented using straightforward extensions of LR parsers [4]. In the marriage of attribute grammars and logic programming, the class of *logical one-pass logical* attribute grammars shows how some kinds of right dependencies can also be managed during conventional top-down parsing [34][36]. Contrary to the work presented in this paper, all these approaches constrain the classes of allowed grammars to strict subclasses of non-circular attribute grammars. In contrast, our approach is able to deal with arbitrary non-circular attribute grammars. If the grammars are of certain types (e.g., l-attributed grammars with an LL(1) underlying context-free grammar), and a suitable semantic evaluation approach is used (e.g., a data-driven strategy), our implementations produce artifacts comparable in performance and memory footprint to those promoted by the abovementioned works. In other cases, the approach is still able to produce running implementations, which can adapt the memory footprint to that required for performing semantic evaluation.

The development of some attribute grammar-based systems has exploited the marriage between attribute grammars and parser generation tools. A common strategy is to build a preprocessor by translating an attribute grammar-based specification language into a running implementation written in terms of a parser generator. In [23] one of these systems is described, which takes an attribute grammar-like specification as input, and it turns it into a YACC implementation. However, since the resulting implementation evaluates attributes during parsing, the class of supported grammars is restricted to a subset of the LR-attributed ones. The Ox system [8] follows a similar approach, but it supports arbitrary non-circular attribute grammars. For this purpose, the processors generated decouple parsing and semantic evaluation by using an optimized implementation of the processing models behind attribute grammars (i.e., to build the parse tree, to arrange attribute instances in topological order, and then to perform evaluation according to this order). XLOP [43], a system developed by us to describe XML processing tasks as attribute grammars, also translates attribute grammar specifications into inputs to a parser generation tool (in this case, CUP). RIE [44], a system that supports a very general class of LR-attributed grammars (ECLR-

attributed grammars [4]) adopts a different implementation approach, by basing the metagenerator on an explicit modification of the Bison parser generation tool. Regardless of the implementation strategy followed (in these examples, based on preprocessors for / extensions to parser generation tools), they ultimately fall in the category of attribute grammar-based tools. Therefore, the general considerations made above concerning the relationships between our approach and attribute grammar – based tools also applies here.

Concerning parser generators, there is a plethora of systems available that can be used during the development of a language processor. A basic feature differentiating them is whether they generate top-down parsers (e.g., the aforementioned tools JavaCC [26] and ANTLR [38], as well as classic tools like COCO/R [32]), or bottom-up ones (e.g., the aformentioned YACC [45],Bison [27] and CUP [5], as well as tools like Tatoo [11], SableCC [13], Beaver[7], Copper [49] or YaJco[8]). Also, these tools differ in the class of grammars allowed (e.g., JavaCC supports LL(k) grammars, while ANTLR supports the aforementioned LL(*) parsing method, able to deal with unbounded look-ahead; additionally tools like Elkhound [30], SDF [10] or, under certain settings, Bison, provide support to arbitrary context-free grammars via the GLR parsing method [46]), by the expressiveness of its specification language (e.g., ANTLR or Tatoo support very sophisticated features, like grammar modularization, rule inheritance, etc.), by whether they include support for lexical specification (e.g., JavaCC, ANTLR) or whether it must be made by using a separating tool (e.g., CUP), and by many other features whose detailed analysis is beyond the scope of the present work. As was indicated, the patterns presented in this paper are applicable to most of these parser generators (in particular in those tools that support deterministic grammars; in tools like SDF, whose outcome is parse forests that must be subsequently disambiguated, the applicability of these patterns vanishes). Also, it is important to notice that, while many of these parser generation tools support the concept of *semantic attribute*, like attribute grammars (e.g., this terminology is explicitly included in ANTLR), it does not mean that these tools give direct support for attribute grammars. Indeed, in addition to managing semantic attributes, the essential aspect of attribute grammars is the support for a dependency-driven execution style: semantic evaluation is not necessarily coupled with parsing, but emerges as a consequence of the dependencies among attributes. In this way, the patterns introduced in this work make it possible to incorporate this computation style into specifications for parser generation tools, and, in consequence, to facilitate the subsequent refinement into more efficient implementations.

Finally, as the implementations of our attribution operations make apparent, we avoid the explicit construction of the parse tree. While this construction is necessary in order to support more sophisticated evaluation

[7] http://beaver.sourceforge.net/
[8] http://code.google.com/p/yajco/

strategies (see, for instance [1]), our simple coding patterns make it unnecessary, since it is centered directly on the construction of dependency graph-like structures. A similar technique is followed in [6], an implementation of circular attribute grammars in Prolog whose semantic equations are described by using λ-expressions. The execution model of the resulting artifact works in two stages: (i) construction of λ-expressions for the root's synthetized attributes, and (ii) interpretation of these expressions according to a least fixpoint semantics to yield the final values. Thus, the resulting approach resembles our demand-driven implementation. In [50], Prolog is also used to implement attribute grammars, and two evaluation strategies are proposed. The first one supposes building terms representing semantic expressions for the root's synthetized attributes, which are subsequently interpreted with a separate interpreter. The second one promotes the use of Prolog co-routine facilities to delay evaluation of arguments until they are instantiated. Thus, the first strategy is analogous to our demand-driven implementation (nevertheless, our implementation is optimized to avoid duplicated evaluations; see [47] for a similar implementation in Prolog that also avoids redundant evaluations). The second one is a Prolog implementation of a data-driven strategy.

7. Conclusions and future work

This paper has shown how to systematically code arbitrary non-circular attribute grammars in the input languages of bottom-up, LALR(1) parser generation tools like YACC, BISON or CUP, as well as top-down, LL parser generation tools like JavaCC or ANTLR. It is done by using a small set of attribution operations. These operations, in turn, can be implemented in different ways in order to enable different semantic evaluation styles. In particular, this paper has illustrated two alternative implementations: one supporting a demand-driven style, and another supporting a data-driven one. The results of this work can be useful to promote a systematic method of using conventional parser generation tools to yield final implementations. This method starts with the initial coding of an attribute grammar-based specification, and then it evolves it in a final implementation by applying systematic implementation patterns and techniques. Thus, by applying and documenting systematic refinements, it is possible, on one hand, to yield efficient implementations and, on the other hand, to track the refinement chain from these final implementations to the original attribute grammar-based specifications. Besides, the method facilitates the incremental introduction of new language features, since they can be described according to attribute grammar conventions, then readily coded in the implementation, and finally optimized according to implementation-dependent criteria. Therefore, the method transports the attribute grammar amenability to doing modular and extensible specifications incrementally to an implementation process based on parser generation tools.

Currently we have successfully tested our method with several small examples, and we are applying it to the development of a non-trivial translator for a Pascal-like language. From these experiences, we have realized how the encoding patterns are simple enough to being applied without specific tooling support (although, of course, this support could be a very valuable facility in our methodology). Also, we have gained further evidence on the feasibility and usefulness of our method with its application in an introductory compiler construction course during the first period of the 2011-2012 academic year at the Complutense University. Indeed, we proposed that our students produce initial implementations of language processors by taking attribute grammar specifications as a guide, and using the method described in this paper. We observed that they didn't find it more difficult to apply than students of previous courses found while hand-coding conventional recursive descent translators. In addition, the quality of the final programs was substantially better than in previous years, since the method encouraged rigorous adherence to the original specification. Thus, we plan to further apply it as a systematic learning method in future editions of the course. Also, as future work, we plan to provide the aforementioned tooling support in order to facilitate the application of the method: automatic application of the coding patterns to produce the initial translation schemes, support for some of the transformations and refinements described in this paper, roundtrip support and support for tracking successive refinements, and support for profiling and debugging the semantic evaluation processes.

Acknowledgements. Thanks are due to project grants TIN2010-21288-C02-01 and Santander-UCM GR 42/10, group reference 962022. Also, Daniel Rodriguez-Cerezo was supported by the Spanish University Teacher Training Program (EDU/3445/2011).

References

1. Ablas, H. Attribute Evaluation Methods. In Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science Vol. 545, Springer, 48-113. (1991)
2. Aho A.V, Lam M.S, Sethi R, Ullman J.D.: Compilers: principles, techniques and tools (2nd Edition). Addison-Wesley. (2006)
3. Akker, R., Melichar, B., Tarhio, J. Attribute Evaluation and Parsing. In Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 187-214. (1991)
4. Akker, R., Melichar, B., Tarhio, J.: The Hierarchy of LR-attributed grammars. In Deransart, P., Jourdan, M (eds.): Attribute Grammars and their Applications – Proceedings of the International Workshop on Attribute Grammars and their Applications (WAGA'90), Paris, France, Lecture Notes in Computer Science 461, Springer, 13-28. (1990)

5. Appel, A.W. Modern Compiler Implementation in Java. Cambridge University Press. (2002)
6. Arbab, B. Compiling Circular Attribute Grammars into Prolog. IBM Journal of Research and Development, Vol. 30, No. 3, 294-309. 1986
7. Augusteijn, L. The Elegant Compiler Generator System. In Deransart, P., Jourdan, M (eds.): Attribute Grammars and their Applications – Proceedings of the International Workshop on Attribute Grammars and their Applications (WAGA'90), Paris, France, Lecture Notes in Computer Science 461, Springer, 238-254. (1990)
8. Bischoff, K.M. Design, Implementation, Use and Evaluation of Ox: An Attribute-Grammar Compiling System based on Yacc, Lex and C. TR #92-31, Dp. Of Computer Science, Iowa State University, (1992)
9. Bochmann, G.V.: Semantic Evaluation from Left to Right. Communications of the ACM, Vol. 19, No. 2, 55-62. (1976)
10. Brand, M.G.J v.d., Deursen, A, v., Heering, J., Jong, H.A.d., Jonge, M.d., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, JJ., Visser, E., Visser, J. The Asf +Sdf Meta-environment: A Component-Based Language Development Environment. In Wilhelm, R (ed.): Compiler Construction - Proceedings of the 10th International Conference on Compiler Construction CC'01, Genova, Italy, Lecture Notes in Computer Science, 2027, Springer, 365-370. (2001)
11. Cervelle, J., Forax, R., Roussel, G. Tatoo: an innovative parser generator. 4th International Symposium on Principles and Practice of Programming in Java PPPJ'06, Mannheim, Germany, ACM, 13-20. (2006)
12. Ekman, T., Hedin, G. The JastAdd system - modular extensible compiler construction. Science of Computer Programming, Vol. 69, No. 1-3, 14-26. (2007)
13. Gagnon, E.M., Hendren, L.J. SableCC, an Object-Oriented Compiler Framework. International Conference on Technology of Object-Oriented Languages TOOLS'98, Sta Barbara, CA, USA, IEEE, 140-154. (1998)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. (1995)
15. Gray, R.W., Heuring, V.P., Levi, S.P., Sloane, A.M., Waite, W.M.: Eli: A Complete, Flexible Compiler Construction System. Communications of the ACM, Vol. 35, 121-131. (1992)
16. Hedin, G. An Object-Oriented Notation for Attribute Grammars. 3rd European Conference on Object-Oriented Programming, Nottingham, UK, Cambridge University Press, 329-345. (1989)
17. Henriques, P.R., Varanda-Pereira, M.J., Mernik, M., Lenic, M., Gray, J.G., Wu, H. Automatic Generation of Language-Based Tools using the LISA System. IEE Proceedings – Software, Vol. 152, No. 2, 54-69. (2005)
18. Jalili, F.: A general linear-time evaluator for attribute grammars. ACM SIGPLAN Notices, Vol. 18, No. 9, 35-44. (1983)
19. Jones, L.G.: Efficient Evaluation of Circular Attribute Grammars. ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, 429-462. (1990)
20. Jourdan, M., Parigot, D.: Internals and Externals of the FNC-2 Attribute Grammar System. In Ablas, H., Melichar, B (eds.).: Attribute Grammars, Applications and Systems, Lecture Notes in Computer Science 545, Springer, 485-504. (1991)
21. Kastens, U., Waite, W.M.: Modularity and Reusability in Attribute Grammars. Acta Informatica, Vol. 31, No. 7, 601-627. (1994)

22. Kastens, U.: GAG: A Practical Compiler Generator. Lecture Notes in Computer Science 141, Springer. (1982)
23. Katwijk, J.: A preprocessor for YACC or a poor man's approach to parsing attributed grammar. ACM SIGPLAN Notices, Vol. 18, No. 10, 12-15. (1983)
24. Kennedy, K., Ramanathan, J.: A Deterministic Attribute Grammar Evaluator Based on Dynamic Sequencing. ACM Transaction of Programming Languages and Systems, Vol. 1, No. 1, 142-160. (1979)
25. Knuth, D. E.: Semantics of Context-free Languages. Mathematical System Theory, Vol. 2, No. 2, 127–145. (1968). See also the correction published in Mathematical System Theory, Vol. 5, No. 1, 95–96.
26. Kodaganallur, V. Incorporating language processing into Java applications: a JavaCC tutorial. IEEE Software, Vol. 21, No. 4, 70-77. (2004)
27. Levine, J. Flex & Bison: Text Processing Tools. O'Reilly Media. (2009)
28. Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E.: Attributed Translations. Journal of Computer and System Sciences, Vol. 9, No. 3, 279-307. (1974)
29. Magnusson, E. Hedin, G.: Circular Reference Attributed Grammars—Their Evaluation and Applications. Science of Computer Programming, Vol. 68, No. 1, 21-37. (2007)
30. McPeak, S., Necula, G.C. Elkhound: A Fast, Practical GLR Parser Generator. International Conference on Compiler Construction (CC'04), Barcelona, Spain, Lecture Notes in Computer Science, Vol. 2985, 73-88. (2005)
31. Mernik, M., Lenic, M., Acdicausevic, E., Zumer, V.: LISA: An Interactive Environment for Programming Language Development. 11th International Conference on Compiler Construction (CC'02), Grenoble, France, Lecture Notes in Computer Science, Vol. 2304, Springer, 1-4. (2002)
32. Mössenböck, H. A Generator for Production Quality Compilers. 3rd intl. workshop on Compiler Compilers (CC'90), Schwerin, Lecture Notes in Computer Science Vol. 477, 42–55. (1990)
33. Oliveira, N., Varanda-Pereira, M.J., Henriques, P.R., da Cruz, D., Cramer, B.: VisualLISA: A Visual Environment to Develop Attribute Grammars. Computer Science and Information Systems Journal, Vol. 7, No. 2, 266-289. (2010)
34. Paakki, J. Prolog in Practical Compiler Writing. Computer Journal, Vol. 34, No. 1, 64-72. (1991)
35. Paakki, J.: Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computing Surveys, Vol. 27, No. 2, 196-255. (1995)
36. Paakki, J.: PROFIT: A System Integrating Logic Programming and Attribute Grammars. 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP'91), Passau, Germany, Lecture Notes in Computer Science Vol. 528, 243-254. (1991)
37. Parr, T., Fisher, K. LL(*): the Foundation of the ANTLR Parser Generator. ACM SIGPLAN Notices - PLDI '11, Vol. 46, No. 6, 425-436. (2011)
38. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf. (2007)
39. Rebernak, D., Mernik, M., Henriques, P.R., Carneiro, D., Varanda-Pereira, M.J. Specifying Languages Using Aspect-oriented Approach: AspectLISA. Journal of Computing and Information Technology, Vol. 4, 343-350. (2006)
40. Rebernak, D., Mernik, M., Henriques, P.R., Varanda-Pereira, M.J.: AspectLISA: An Aspect-oriented Compiler Construction System Based on Attribute Grammars. Electronics Notes in Theoretical Computer Science – LDTA'06, Vol. 164, 37-53. (2006)

41. Rodriguez-Cerezo, D., Sarasa, A., Sierra, J.L.: Implementing Attribute Grammars Using Conventional Compiler Construction Tools. 3rd Workshop on Advances in Programming Languages (WAPL'11), Szczezin, Poland, IEEE, 855-862. (2011)
42. Saraiva, J., Swiestra, D.: Generic Attribute Grammars. 2nd Workshop on Attribute Grammars and Their Applications (WAGA'99), Amsterdam, The Netherlands. (1999)
43. Sarasa, A., Temprado-Battad, B., Sierra, J.L, Fernández-Valmayor, A.: XML Language-Oriented Processing with XLOP. 5th International Symposium on Web and Mobile Information Services, Bradford, UK, Proceedings of AINA'09 Workshops, IEEE, 322-327. (2009)
44. Sassa, M., Ishizuka, H., Nakata, I. Rie, a compiler generator based on a one-pass-type attribute grammar. Software – Practice & Experience, Vol. 25, No. 3, 229-250, (1995)
45. Schreiner, A.T., Friedman, H.G. Introduction to Compiler Construction with Unix. Prentice-Hall. (1985)
46. Scott, E., Johnstone, A. Right nulled GLR parsers. ACM Transactions on Programming Languages and Systems, Vol. 28, No. 4, 577-618. (2006)
47. Sierra, J.L., Fernández-Valmayor, A. A Prolog Framework for the Rapid Prototyping of Language Processors with Attribute Grammars. Electronics Notes in Theoretical Computer Science – LDTA'06, Vol. 164, 19-36. (2006)
48. Vogt, H.H., Swierstra, S.D., Kuiper, M.F.: Higher-Order Attribute Grammars. ACM SIGPLAN Notices Vol. 24, No. 7. (1989)
49. Vyk, E.R.v., Schwerdfeger, A.C. Context-aware scanning for Parsing Extensible Languages. 6th International Conference on Generative Programming and Component Engineering GPCE'06, Portland, Oregon, USA, ACM, 63-72. (2006)
50. Walsteijn, M.J., Kuiper, M.F.: Attribute Grammars in Prolog. Technical Report, RU-CS-86-14, Utrecht University. (1986)
51. Wyk, E.V., Bodin, D., Gao, J., Krishnan, L.: Silver: An Extensible Attribute Grammar System. Science of Computer Programming, Vol. 75, No. 1-2, 39-54. (2010)

Daniel Rodríguez-Cerezo is a PhD student in the Computer Science School at UCM, and a member of the research group ILSA (Implementation of Language-Driven Software and Applications: http://ilsa.fdi.ucm.es). His research is focused on the use of several e-Learning techniques (simulations, interactive prototyping tools, recommendation systems for learning object repositories, etc.) to improve teaching and learning of the Software Language Engineering discipline. Besides, he is interested in the development and improvement of software language engineering techniques.

Antonio Sarasa-Cabezuelo is a full-time Lecturer in the Computer Science School at Complutense University of Madrid, Spain (UCM). His research is focused on the language-oriented development of XML-processing applications, and on the development of applications in the fields of digital humanities and e-Learning. He was one of the developers of the Agrega project on digital repositories (a pioneer project in this field in Spain). He is a member of ILSA. He has participated in several research projects in the fields

of software language engineering, digital humanities and e-learning, and he has published over 50 research papers in national and international conferences.

José-Luis Sierra is an Associate Professor at the UCM's Computer Science School, where he leads the ILSA Research Group. His research is focused on the development and practical uses of computer language description tools and on the language-oriented development of interactive and web applications in the fields of digital humanities and e-Learning. Prof. Sierra has leaded and participated in several research projects in the fields of digital humanities, e-learning and software language engineering, the results of which have been published in over 100 research papers in international journals, conferences and book chapters. He serves regularly as reviewer / PC Member for several international reputed journals and conferences.

Received: December 23, 2011 Accepted: June 1, 2012.

DOI: 10.2298/CSIS111110021F

Implementation of EasyTime Formal Semantics using a LISA Compiler Generator

Iztok Fister Jr.[1], Marjan Mernik[1], Iztok Fister[1], Dejan Hrnčič[1]

University of Maribor
Faculty of electrical engineering and computer science
Smetanova 17
2000 Maribor
Slovenia
iztok.fister@guest.arnes.si,
marjan.mernik@uni-mb.si,
iztok.fister@uni-mb.si,
dejan.hrncic@uni-mb.si

Abstract. A manual measuring time tool in mass sporting competitions would not be imaginable nowadays, because many modern disciplines, such as IRONMAN, last a long-time and, therefore, demand additional reliability. Moreover, automatic timing-devices based on RFID technology, have become cheaper. However, these devices cannot operate as standalone because they need a computer measuring system that is capable of processing incoming events, encoding the results, assigning them to the correct competitor, sorting the results according to the achieved times, and then providing a printout of the results. This article presents the domain-specific language EasyTime, which enables the controlling of an agent by writing the events within a database. It focuses, in particular, on the implementation of EasyTime with a LISA tool that enables the automatic construction of compilers from language specifications, using Attribute Grammars.

Keywords: domain-specific language, compiler, code generator, measuring time.

1. Introduction

In the past, timekeepers measured the time manually. The time given by a timer was assigned to competitors based on their starting number, and these competitors were then placed in order according to their achieved results and category. Later, manual timers were replaced by timers with automatic time-registers capable of capturing and printing out registered times. However, assigning the times to competitors based on their starting numbers, was still done manually. This work could be avoided by using electronic-measuring technology which, in addition to registering the time, also enables the registering of competitors' starting numbers. An expansion of RFID (Radio Frequency Identification) technology has helped this measuring-technology to become less expensive ([4,

23]), and accessible to a wider-range of users (e.g., sports clubs, organizers of sporting competitions). Moreover, they were also able to compete with time-measuring monopolies at smaller competitions.

In addition to measuring technology, a flexible computer system is also needed to monitor the results. The proposed computer system enables the monitoring of different sporting competitions using a various number of measuring devices and measuring points, the online recording of events, the writing of results, as well as efficiency and security. This measuring device is dedicated to the registration of events and is triggered either automatically, when the competitor crosses the measuring point that acts as an electromagnetic antenna fields with an appropriate RFID tag, or manually, when an operator presses the suitable button on a personal computer that acts as a timer. The control point is the place where the organizers want to monitor the results. Until now, each control point has required its own measuring device. However, modern electronic-measuring devices now allow for the handling of multiple control points, simultaneously. Moreover, each registered event can have a different meaning, depending on the situation within which it is generated. Therefore, an event is handled by the measuring system according to those rules that are valid for the control point. As a result, the number of control points (and measuring devices) can be reduced by using more complex measurements. Fortunately, the rules controlling events can be described easily with the use of a domain-specific language (DSL) [11, 17]. When using this DSL, measurements at different sporting competitions can be accomplished by an easy pre-configuration of the rules.

A DSL is suited to an application domain and has certain advantages over general-purpose languages (GPL) within a specific domain [17]. The GPL is dedicated to writing software over a wider-range of application domains. General problems are usually solved using these languages. However, a programmer is necessary for changing the behavior of a program written in a GPL. On the other hand, the advantages of DSL are reflected in its greater expressive power in a particular domain and, hence, increased productivity [14] , ease of use (even for those domain experts who are not programmers), and easier verification and optimization [17]. This article presents a DSL called EasyTime, and its implementation. EasyTime is intended for controlling those agents responsible for recording events from the measuring devices, into a database. Therefore, the agents are crucial elements of the proposed measuring system. To the best of the author's knowledge there is no comparable DSL of time measuring for sport events, whilst some DSLs for performance measurement of computer systems [2, 21] as well as on general measurement systems do indeed already exist [13]. Finally, EasyTime has been successfully employed in practice, as well. For instance, it measured times at the World Championship for the double ultra triathlon in 2009 [9], and at a National Championship in the time-trials for bicycle in 2010 [9].

The structure of the remaining article is as follows; In the second section, those problems are illustrated that accompany time-measuring at sporting com-

petitions. Focus is directed primarily on triathlon competitions, because they contain three disciplines that need to be measured, and also because of their lengthy durations. The design of DSL EasyTime is briefly shown in section three. The implementation of the EasyTime compiler is described in the fourth section, whilst the fifth section explains the execution of the program written in EasyTime. Finally, the article is concluded with a short analysis of the work performed, and a look at future work. This paper extends a previous workshop paper [10] by providing general guidelines on how to transform formal language specifications using denotational semantics into attribute grammars. The concreteness of these guidelines is shown on EasyTime DSL.

2. Measuring Time in Sporting Competitions

In practice, the measuring time in sporting competitions can be performed manually (classically or with a computer timer) or automatically (with a measuring device). The computer timer is a program that usually runs on a workstation (personal computer) and measures in real-time. Thereby, the processor tact is exploited. The processor tact is the velocity with which the processor's instructions are interpreted. A computer timer enables the recording of events that are generated by the competitor crossing those measure points (MP) in line with the measuring device. In that case, however, the event is triggered by an operator pressing the appropriate button on the computer. The operator generates events in the form of $\langle \#, MP, TIME \rangle$, where $\#$ denotes the starting number of a competitor, MP is the measuring point, and $TIME$ is the number of seconds since 1.1.1970 at 0:0:0 (timestamp). One computer timer represents one measuring-point.

Today, the measuring device is usually based on RFID technology [6], where identification is performed using electromagnetic waves within a range of radio frequencies, and consists of the following elements:

- readers of RFID tags,
- primary memory,
- LCD monitor,
- numerical keyboard, and
- antenna fields.

More antenna fields can be connected on to the measuring device. One antenna field represents one measuring point. Each competitor generates an event by crossing the antenna field using passive RFID tags that include an identification number. This number is unique and differs from the starting number of the competitor. The event from the measuring device is represented in the form of $\langle \#, RFID, MP, TIME \rangle$, where the identification number of the RFID tag is added to the previously mentioned triplet.

The measuring devices and workstations running the computer timer can be connected to the local area network. Communication with devices is performed by a monitoring program, i.e. an agent, that runs on the database server. This

agent communicates with the measuring device via the TCP/IP sockets, and appropriate protocol. Usually, the measuring devices support a $Telnet$ protocol that is character-stream oriented and, therefore, easy to implement. The agent employs the file transfer protocol (ftp) to communicate with the computer timer.

2.1. Example: Measuring Time in Triathlons

Special conditions apply for triathlon competitions, where one competition consists of three disciplines. This article, therefore, devotes most of its attention to this problem.

The triathlon competition is performed as follows: first, the athletes swim, then they ride a bicycle and finally run. In practice, all these activities are performed consecutively. However, the transition times, i.e. the time that elapses when a competitor shifts from swimming to bicycling, and from bicycling to running, are added to the summary result. There are various types of triathlon competitions that differ according to the lengths of various courses. In order to make things easier, organizers often employ round courses (laps) of shorter lengths instead of one long course. Therefore, the difficulty of measuring time is increased because the time for each lap needs to be measured.

Measuring time in triathlon competitions can be divided into nine control points (Fig. 1). The control point (CP) is a location on the triathlon course, where the organizers need to check the measured time. This can be intermediate or final. When dealing with a double triathlon there are 7.6 km of swimming, 360 km of bicycling, and 84 km of running. Hence the swimming course of 380 meters consists of 20 laps, the bicycling course of 3.4 kilometers contains 105 laps, and the running course of 1.5 kilometers has 55 laps (Fig. 1).

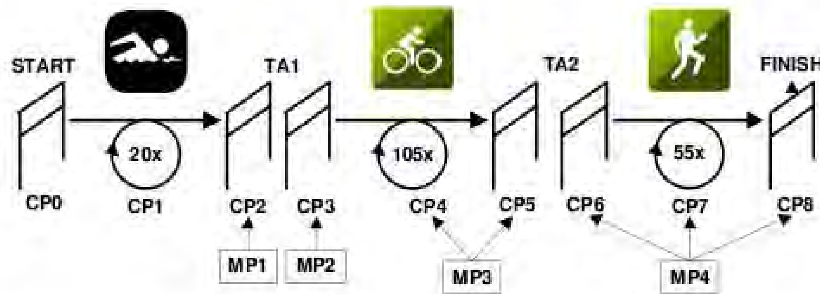


Fig. 1. Definition of control points in the triathlon

Therefore, the final result for each competitor in a triathlon competition (CP8) consists of five final results: the swimming time SWIM (CP2-CP0), the time for the first transition TA1 (CP3-CP2), the time spent bicycling BIKE (CP5-CP3), the time for the second transition TA2 (CP6-CP5), the time spent running RUN (CP8-CP6), and three intermediate results: the intermediate time for swimming

(CP1), the intermediate time for bicycling (CP4) and the intermediate time for running (CP7). However, the current time INTER_x and the number of remaining laps LAPS_x are measured by the intermediate results, where $x = \{1, 2, 3\}$ denotes the appropriate discipline (1=SWIM, 2=BIKE and 3=RUN).

The DSL EasyTime was developed in order to achieve this goal, and has been employed in practice by conducting measurements at the World Championship in the Double Triathlon in 2009. Note that the measurements were realized according to Fig. 1. The next sections presents the design, implementation, and operation of EasyTime.

3. The Design of the EasyTime Domain-Specific Language

Typically, the development of a DSL consists of the following phases [17]:

- a domain analysis,
- a definition of an abstract syntax,
- a definition of a concrete syntax,
- a definition of formal semantics, and
- an implementation of the DSL.

Domain analysis provides an analysis of the application domain, i.e. measuring time in sporting competitions. The results of this analysis define those concepts of EasyTime that are typically represented within a feature diagram [5, 25]. The feature diagram also describes dependencies between the concepts of DSL. Thus, each concept can be broken-down into features and sub-features. In the case of EasyTime, the concept $race$ consists of sub-features: $events$ (e.g., $swimming$, $bicycling$, and $running$), $control\ points$, $measuring\ time$, $transition\ area$, and $agents$. Each $control\ point$ is described by its $starting$ and $finish$ line and at least one lap. In addition, the feature $transition\ area$ can be introduced as the difference between the finish and start times. Both $updating\ time$ and $decrementing\ laps$ are sub-features of $measuring\ time$. However, an $agent$ is needed for the processing of events received from the measuring device. It can act either $automatically$ or $manually$. Note that during domain analysis not all the identified concepts are useful for solving actual problem. Hence, the identified concepts can be further classified into [16]:

- irrelevant concepts, those which are irrelevant to the actual problem;
- variable concepts, those which actually need to be described in the DSL program; and
- fixed concepts, those which can be built into the DSL execution environment.

Domain analysis identifies several variable and fixed concepts within the application domain that needs to be mapped into EasyTime syntax and semantics [17]. At first, the abstract syntax is defined (context-free grammar). Each variable concept obtained from the domain analysis is mapped to a non-terminal in the context-free grammar; additionally, some new non-terminal and

terminal symbols are defined. The translations of the EasyTime domain concepts to non-terminals are presented and explained in Table 1, whilst an abstract syntax is presented in Table 2. Note that, the concepts *Events* and *Transition* are irrelevant for solving actual problem and are not mapped into non-terminals' symbols (denoted as *none* in Table 1). Interestingly, a description of agents and measuring places cannot be found in other DSLs or GPLs. Whilst attribute declaration is similar to variable declaration in many other programming languages. However, note that there is the distinction that variables are actually database attributes allocated for every competitor. Some statements, such as assignment, conditional statement, and compound statement can be found in many other programming languages, whilst decrement attributes and update attributes are domain-specific constructs.

Table 1. Translation of the application domain concepts into a context-free grammar

Application domain concepts	Non-terminal	Formal sem.	Description
Race	P	$\mathcal{CP}$	Description of agents; control points; measuring places.
Events (swimming, cycling, running)	none	none	Measuring time is independent from the type of an event. However, good attribute's identifier in control points description will resemble the type of an event.
Transition area times	none	none	Can be computed as difference between events final and starting times.
Control points (start, number of laps, finish)	D	$\mathcal{D}$	Description of attributes where start and finish time will be stored as well as remaining laps.
Measuring places (update time, decrement lap)	M	$\mathcal{CM}$	Measuring place id; agent id, which will control this measuring place; specific actions (presented with new non-terminal S) which will be performed at this measuring place (e.g., decrement lap).
Agents (automatic, manual)	A	$\mathcal{A}$	Agent id; agent type (automatic, manual); agent source (file, ip).

Table 2. The abstract syntax of EasyTime

$P \in$ **Pgm**	$A \in$ **Adec**
$D \in$ **Dec**	$M \in$ **MeasPlace**
$S \in$ **Stm**	$b \in$ **Bexp**
$a \in$ **Aexp**	$n \in$ **Num**
$x \in$ **Var**	$file \in$ **FileSpec**
$ip \in$ **IpAddress**	
P	::= $A\ D\ M$
A	::= n **manual** $file \mid n$ **auto** $ip \mid A_1; A_2$
D	::= **var** $x := a \mid D_1; D_2$
M	::= **mp**$[n_1] \rightarrow$ **agnt**$[n_2]\ S \mid M_1; M_2$
S	::= **dec** $x \mid$ **upd** $x \mid x := a \mid (b) \rightarrow S \mid S_1; S_2$
b	::= **true** $\mid$ **false** $\mid a_1 == a_2 \mid a_1 != a_2$
a	::= $n \mid x$

Although a language designer can proceed after domain analysis with informal or formal design patterns [17] the formal design step is preferred since it can identify problems before the DSL is actually implemented [27]. Moreover, formal specifications can be implemented automatically by language development systems, thus significantly reducing the implementation effort [17]. The meaning of the EasyTime language constructs is prescribed during the formal semantics phase. Each language construct, belonging to the syntax domain, is mapped into an appropriate semantic domain (Table 3) by semantic functions $\mathcal{CP}$, $\mathcal{A}$, $\mathcal{D}$, $\mathcal{CM}$, $\mathcal{CS}$, $\mathcal{CB}$, and $\mathcal{CA}$ (Table 4).

Table 3. Semantic domains

Domain	Element
Integer=$\{\ldots-3,-2,-1,0,1,2,3\ldots\}$	$n \in$ **Integer**
Truth-Value=$\{true, false\}$	
State=**Var**→**Integer**	$s \in$ **State**
AType=$\{manual, auto\}$	
Agents=**Integer**→**AType** × $(FileSpec \cup IpAddress)$	$ag \in$ **Agents**
Runners=$(Id \times RFID \times LastName \times FirstName)^*$	$r \in$ **Runners**
DataBase=$(Id \times Var_1 \times Var_2 \times \ldots \times Var_n)^*$	$db \in$ **DataBase**
Code=**String**	$c \in$ **Code**

These semantic functions translate EasyTime constructs into the instructions of the simple virtual machine. The meaning of virtual machine instructions has been formally defined using operational semantics (Table 5) as the transition of configurations $< c,\ e,\ db,\ j >$, where c is a sequence of instructions, e is the evaluation stack to evaluate arithmetic and boolean expressions, db is the database, and j is the starting number of a competitor. More details of EasyTime syntax and semantics are presented in [9]. This article focuses on the implementation phase, as presented in the next section.

The sample program written in EasyTime that covers the measuring time in the double ultra triathlon is presented by Algorithm 1. In lines 1-2 two agents are defined. Agent no. 1 is manual and agent no. 2 is automatic. In lines 4-14 several variables, attributes in a database for each competitor, are defined and initialized appropriately. For example, from Figure 1 it can be seen that 20 laps are needed for the swimming course and $ROUND1$ is set to 20, 105 laps are needed for the bicycling course and $ROUND2$ is set to 105, and 55 laps are needed for the running course and $ROUND3$ is set to 55. Lines 16-19 define the first measuring place which is controlled by manual agent no. 1. At this measuring place the intermediate swimming time must be updated in the database ($upd\ SWIM$) and the number of laps must be decremented ($dec\ ROUND1$). Lines 20-22 define the second measuring place which is also controlled by manual agent no. 1. At this measuring place only transition time must be stored in the database ($upd\ TRANS1$). Lines 23-27 define the third measuring place which is controlled by automatic agent no. 2. At this measuring place we must update the intermediate result for bicycling ($upd\ INTER2$) and decrement the number of laps ($dec\ ROUND2$). If a competitor finished

Table 4. EasyTime formal semantics

$\mathcal{CP}$: **Pgm** → **Runners**	→ **Code** × **Integer** × **DataBase**
$\mathcal{CP}[\![A\ D\ M]\!]r$	= let $s = \mathcal{D}[\![D]\!]\emptyset$: db =create&insertDB(s, r) in $(\mathcal{CM}[\![M]\!](\mathcal{A}[\![A]\!]\emptyset), db)$
$\mathcal{A}$: **Adec** → **Agents**	→ **Agents**
$\mathcal{A}[\![n$ **manual** $file]\!]ag$	= $ag[n \to (manual, file)]$
$\mathcal{A}[\![n$ **auto** $ip]\!]ag$	= $ag[n \to (auto, ip)]$
$\mathcal{A}[\![A_1; A_2]\!]ag$	= $\mathcal{A}[\![A_2]\!](\mathcal{A}[\![A_1]\!]ag)$
$\mathcal{D}$: **Dec**→**State**	→ **State**
$\mathcal{D}[\![$**var** $x := a]\!]s$	= $s[x \to a]$
$\mathcal{D}[\![D_1, D_2]\!]s$	= $\mathcal{D}[\![D_2]\!](\mathcal{D}[\![D_1]\!]s)$
$\mathcal{CM}$: **MeasPlace** → **Agents**	→ **Code** × **Integer**
$\mathcal{CM}[\![$**mp**$[n_1] \to$ **agnt**$[n_2]S]\!]ag$	= (WAIT $i : \mathcal{CS}[\![S]\!](ag, n_2), n_1)$
$\mathcal{CM}[\![M_1; M_2]\!]ag$	= $\mathcal{CM}[\![M_1]\!]ag : \mathcal{CM}[\![M_2]\!]ag$
$\mathcal{CS}$: **Stm**→ **Agents** × **Integer**	→ **Code**
$\mathcal{CS}[\![$ **dec** $x]\!](ag, n)$	= FETCH x:DEC:STORE x
$\mathcal{CS}[\![$ **upd** $x]\!](ag, n)$	= FETCH y:STORE x where $y = \begin{cases} \text{accessfile}(ag(n) \downarrow 2) & \text{if } ag(n) \downarrow 1 = manual \\ \text{connect}(ag(n) \downarrow 2) & \text{if } ag(n) \downarrow 1 = automatic \end{cases}$
$\mathcal{CS}[\![x := a]\!](ag, n)$	= $\mathcal{CA}[\![a]\!]$:STORE x
$\mathcal{CS}[\![(b) \to S]\!](ag, n)$	= $\mathcal{CB}[\![b]\!]$:BRANCH($\mathcal{CS}[\![S]\!](ag, n), NOOP$)
$\mathcal{CS}[\![S_1; S_2]\!](ag, n)$	= $\mathcal{CS}[\![S_1]\!](ag, n) : \mathcal{CS}[\![S_2]\!](ag, n)$
$\mathcal{CB}$: **Bexp**	→ **Code**
$\mathcal{CB}[\![$**true**$]\!]$	= TRUE
$\mathcal{CB}[\![$**false**$]\!]$	= FALSE
$\mathcal{CB}[\![a_1 == a_2]\!]$	= $\mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!]$:EQ
$\mathcal{CB}[\![a_1 != a_2]\!]$	= $\mathcal{CA}[\![a_2]\!] : \mathcal{CA}[\![a_1]\!]$:NEQ
$\mathcal{CA}$: **Aexp**	→ **Code**
$\mathcal{CA}[\![n]\!]$	= PUSH n
$\mathcal{CA}[\![x]\!]$	= FETCH x

all the requested 105 laps ($ROUND2 == 0$) then time spent on the bicycle must be stored in the database ($upd\ BIKE$). Lines 28-33 define the fourth measuring place which is also controlled by automatic agent no. 2. At this measuring place we must first check if a competitor has just started running ($ROUND3 == 55$). If this is the case, we must record the transition time between bicycling and running ($upd\ TRANS2$). At this measuring place we also must update the intermediate result for running ($upd\ INTER3$) and decremented number of laps ($dec\ ROUND3$). If a competitor finished all the requested 55 laps ($ROUND3 == 0$) then the final time must be stored in the database ($upd\ RUN$).

Algorithm 1 EasyTime program for measuring time in a triathlon competition as illustrated in Fig. 1

```
1: 1 manual ”abc.res”;
2: 2 auto 192.168.225.100;
3:
4: var ROUND1 := 20;
5: var INTER1 := 0;
6: var SWIM := 0;
7: var TRANS1 :=0;
8: var ROUND2 := 105;
9: var INTER2 :=0;
10: var BIKE := 0;
11: var TRANS2 :=0;
12: var ROUND3 := 55;
13: var INTER3 := 0;
14: var RUN := 0;
15:
16: mp[1] → agnt[1] {
17:   (true) → upd SWIM;
18:   (true) → dec ROUND1;
19: }
20: mp[2] → agnt[1] {
21:   (true) → upd TRANS1;
22: }
23: mp[3] → agnt[2] {
24:   (true) → upd INTER2;
25:   (true) → dec ROUND2;
26:   (ROUND2 == 0) → upd BIKE;
27: }
28: mp[4] → agnt[2] {
29:   (ROUND3 == 55) → upd TRANS2;
30:   (true) → upd INTER3;
31:   (true) → dec ROUND3;
32:   (ROUND3 == 0) → upd RUN;
33: }
```

4. Implementation of the Domain-Specific Language EasyTime

4.1. A LISA Compiler-Generator

One of the benefits of formal language specifications is the unique possibility for automatic language implementation. Although some compiler generators accept denotational semantics [22], the generated compilers are mostly inefficient. Although many compiler-generators based on attribute grammars [12, 20] exist today, we selected a LISA compiler-compiler that was developed at the University of Maribor in the late 1990s [18]. The LISA tool produces a highly efficient source code for: the scanner, parser, interpreter or compiler, in Java. The lexical and syntactical parts of the language specification in LISA supports various well-known formal methods, such as regular expressions and BNF [1]. LISA provides two kinds of user interfaces:

– a graphic user interface (GUI) (Fig. 2), and
– a Web-Service user interface.

The main features of LISA are as follows:

Table 5. The virtual machine specification

$\langle$PUSH $n : c, e, db, j\rangle$	$\triangleright \langle c, n : e, db, j\rangle$	
$\langle$TRUE $: c, e, db, j\rangle$	$\triangleright \langle c, true : e, db, j\rangle$	
$\langle$FALSE $: c, e, db, j\rangle$	$\triangleright \langle c, false : e, db, j\rangle$	
$\langle$EQ $: c, z_1 : z_2 : e, db, j\rangle$	$\triangleright \langle c, (z_1 == z_2) : e, db, j\rangle$	if $z_1, z_2 \in$ **Int**
$\langle$NEQ $: c, z_1 : z_2 : e, db, j\rangle$	$\triangleright \langle c, (z_1 != z_2) : e, db, j\rangle$	if $z_1, z_2 \in$ **Int**
$\langle$DEC $: c, z : e, db, j\rangle$	$\triangleright \langle c, (z - 1) : e, db, j\rangle$	if $z \in$ **Int**
$\langle$WAIT $i : c, e, db, j\rangle$	$\triangleright \langle c, e, db, i\rangle$	
$\langle$FETCH $x : c, e, db, j\rangle$	$\triangleright \langle c,$ select x from db where $Id = j : e, db, j\rangle$	
$\langle$FETCH $accessfile(fn) : c, e, db, j\rangle$	$\triangleright \langle c, time : e, db, j\rangle$	
$\langle$FETCH $connect(ip) : c, e, db, j\rangle$	$\triangleright \langle c, time : e, db, j\rangle$	
$\langle$STORE $x : c, z : e, db, j\rangle$	$\triangleright \langle c, e,$ update db set $x = z$ where $Id = j, j\rangle$	if $z \in$ **Int**
$\langle$NOOP $: c, e, db, j\rangle$	$\triangleright \langle c, e, db, j\rangle$	
$\langle$BRANCH$(c_1, c_2) : c, t : e, db, j\rangle$	$\triangleright \begin{cases}\langle c_1 : c, e, db, j\rangle \\ \langle c_2 : c, e, db, j\rangle\end{cases}$	if $t = true$ otherwise

- since it is written in Java, LISA works on all Java platforms,
- a textual or a visual environment,
- an Integrated Development Environment (IDE), where users can specify, generate, compile and execute programs on the fly,
- visual presentations of different structures, such as finite-state-automata, BNF, a dependency graph, a syntax tree, etc.,
- modular and incremental language development [19].

LISA specifications are based on Attribute Grammar (AG) [20] as introduced by D.E. Knuth [12]. The attribute grammar is a triple $AG = \langle G, A, R\rangle$, where G denotes a context-free grammar, A a finite set of attributes, and R a finite set of semantic rules. In line with this, the LISA specifications (Table 6) include:

- lexical regular definitions (lexicon part in Table 6),
- attribute definitions (attributes part in Table 6),
- syntax rules (rule part before compute in Table 6),
- semantic rules, (rule part after compute in Table 6) and
- operations on semantic domains (method part in Table 6).

Lexical specifications for EasyTime in LISA (Fig. 2) are similar to those used in other compiler-generators, and are obtained from EasyTime concrete syntax (Table 7). Note that in the rule part of LISA specifications the terminal symbols that are defined by regular expressions in the lexical part are denoted with symbol # (e.g., #Id, #Int). EasyTime concrete syntax is derived from EasyTime abstract syntax (Table 2). The process of transforming abstract syntax into concrete syntax is straightforward, and presented in [9]. Semantic rules are written in LISA as regular Java assignment statements and are attached to a particular syntax rule. Hence, the rule part in LISA (Table 6) specifies the BNF production as well as the attribute computations attached to this production. Since the theory about attribute grammars is a standard topic of compiler science, it is assumed that a reader has a basic knowledge about attribute grammars [12, 20].

```
C:\LISA2-1\Examples\Easytime\EasyTime.lisa
EasyTime.lisa
language EasyTime {
  lexicon  {
        Int         [0-9]+
        Id          [a-zA-Z][a-zA-Z0-9]*
        Keyword1    mp | agnt
        Keyword2    dec
        Keyword4    upd
        Keyword3    true | false
        file        \"[a-z]+\.[a-z][a-z][a-z]\"
        ip          [0-9][0-9][0-9]\.[0-9][0-9][0-9]\.[0-9][0-9][0-9]\.[0-9][0-9][0-9]
        Operator    == | != | := | \+ | \* | \- | \= | <= | \-\>
        Separator   \; | \( | \) | \[ | \] | \{ | \}
        ignore      [\0x09\0x0A\0x0D\  ]+
  }
  attributes String *.code, *.type, *.y, *.file_ip, *.name;
             Hashtable *.outAG,*.inAG, *.inState, *.outState;
             int *.number, *.value,  *.n;
             boolean *.ok;

Lisa 2.1  LAB RAJ & LSPO
          University of Maribor
Starting compilation of 'C:\LISA2-1\Examples\Easytime\EasyTime.lisa'
Compiling EasyTime
Start waeving...
Weaving language "EasyTime"
Done
Generating lexical definitions for language 'EasyTime'
Generating BNF definitions for language 'EasyTime'
Generating semantics function definitions for language 'EasyTime'
```

Fig. 2. LISA GUI

4.2. Translation scheme from denotational semantics to attribute grammars

The most difficult part of transforming formal EasyTime specifications into LISA specifications, consists of mapping denotational semantics into attribute grammars. This mapping can be described in a systematic manner, and can also be used for the implementation of other DSLs (e.g., [15]). It consist of the following steps similar to the translation scheme from natural semantics into attribute grammars [3]:

1. Identification of syntactic and semantic domains in each semantic function of denotational semantics. Identified syntactic domains must have their counterparts in non-terminals of concrete syntax. Identified semantic domains must be represented appropriately, with suitable data structures (types) in chosen programming language.
2. Identification of inherited and synthesized attributes for each non-terminal derived in step 1. Semantic argument, which is an input parameter in semantic function, is represented as inherited attribute, while an output parameter is represented as synthesized attribute. According to [12], the starting

Table 6. LISA specifications

```
language L1 [ extends L2, ..., LN ] {
  lexicon {
    [[P] overrides | [P] extends] R regular expr.
      ⋮
  }
  attributes type A1, ..., AM
    ⋮
 rule [[Y] extends | [Y] overrides] Z {
    X ::= X11 X12 ... X1p compute {
        semantic functions }
      ⋮
    |
      Xr1 Xr2 ... Xrt compute {
        semantic functions }
    ;
    }
    ⋮
  method [[N] overrides | [N] extends] M {
    operations on semantic domains
    }
  ⋮
}
```

non-terminal should not have inherited attributes. Whilst LISA automatically infers whether an attribute is inherited or synthesized [12], the type of attribute must be specified (Fig. 2).

3. For all identified attributes attached to a particular non-terminal's, semantic equations need to be developed that are in conformance to semantic equations from denotational semantics. In particular, semantic equations need to be written for each synthesized attribute of the left-hand side non-terminal and for each inherited attribute attached to non-terminals of the right-hand side. This rule is applied to every production of a concrete syntax. In this step the whole semantic equation is not yet written, only the existence of such an equation is identified.
4. In the productions of concrete syntax certain new non-terminals appear, which are consequences of transformation of abstract syntax into concrete syntax. These non-terminals also carry information that are needed for computations. In this step such non-terminals are identified and attached attributes are classified into inherited and synthesized.
5. Finalizing semantics for all identified semantic equations. These semantic equations need to be in conformance to denotational semantics, and require careful examination of semantic functions of denotational semantics (e.g., $\mathcal{CP}$, $\mathcal{A}$, $\mathcal{D}$, $\mathcal{CM}$, $\mathcal{CS}$, $\mathcal{CB}$, and $\mathcal{CA}$ from Table 4). This step is most demanding.
6. In code generation, certain additional tests are usually performed, which are sometimes non-described in formal semantics, in order to be on a proper abstraction level. For example, only declared variables can be used in ex-

Table 7. The concrete syntax of EasyTime

PROGRAM	::= AGENTS DECS MES_PLACES
AGENTS	::= AGENTS AGENT | ε
AGENT	::= #Int auto #ip ; | #Int manual #file ;
DECS	::= DECS DEC | ε
DEC	::= var #Id := #Int ;
MES_PLACES	::= MES_PLACE MES_PLACES | MES_PLACE
MES_PLACE	::= mp[#Int] − > agnt [#Int] { STMTS }
STMTS	::= STMT STMTS | STMT
STMT	::= dec #Id ; | upd #Id ; | #Id := EXPR ; | (LEXPR) − > STMT
LEXPR	::= true | false | EXPR == EXPR | EXPR != EXPR
EXPR	::= #Int | #Id

pressions and commands of a language under development. Such additional tests require that new attributes are defined to carry the results of tests, as well as existing attributes being propagated to appropriate constructs (e.g., expressions, commands). An attribute grammar is finalized during this step.

Note that the presented guidelines are general and not restricted to a particular class of attribute grammars [12, 20] (e.g., S-attributed, L-attributed, ordered attribute grammar, absolutely non-circular attribute grammar). Actually, the class of obtained attribute grammar can be identified only after the translation has been completely performed.

4.3. Translation scheme from EasyTime formal semantics to LISA

When applying the aforementioned rules to EasyTime, the following results are obtained after each step.

Step 1:

The following non-terminals from Table 7 represent syntactic domains (Table 2): PROGRAM ∊ **Pgm**, MES_PLACES ∊ **MeasPlace**, DECS ∊ **Dec**, AGENTS ∊ **Adec**, STMTS ∊ **Stm**, etc. Semantic domains (Table 3) such as **Integer**, **Truth-Value**, **Code** have direct counterparts with Java types: int, boolean, and String. While semantic domains which are functions (e.g., **State**, **Agents**) can be modeled with Java Hashtable type. For example, from Figure 2 we can notice that attribute $inState$, which represents function **State**, is of type Hashtable. Using methods such as $put()$, $get()$, and $containsKey()$ we can respectively insert a new variable, obtain a variable's value, and check if the variable is declared. Other semantic domains (e.g., cartesian product) can be modeled easily with a Java rich type system. Hence, in LISA the type of attributes regarding an attribute grammar can be any valid pre-defined or user-defined Java type. An example of auxiliary operations on semantic domains (e.g., Hashtable), is presented in [10].

Step 2:

From $\mathcal{CP}$: **Pgm** → **Runners** → **Code** × **Integer** × **DataBase** (Table 4) it can be concluded that to non-terminal PROGRAM one inherited (representing a parameter of type **Runners**) and three synthesized attributes (representing parameters of **Code**, **Integer**, and **DataBase**) need to be attached. However, the starting non-terminal should not have inherited attributes [12, 20]. From the definition of semantic function $\mathcal{CP}$ (Table 4) it can be noticed that the input parameter of type **Runners** are only needed to create a database. Hence, both parameters (of type **Runners** and **DataBase**) can be omitted from LISA specifications, and its functionality can be externally implemented. Moreover, it was decided to represent both the generated code and the identification number of the virtual machine, where the code is going to be executed, as a string "(Code, Integer)". Hence, only one synthesized attribute, PROGRAM.code, is attached to starting non-terminal PROGRAM.
From $\mathcal{A}$: **Adec** → **Agents** → **Agents** (Table 4) it can be concluded that one inherited and one synthesized attribute need to be attached to non-terminal AGENTS. For this purpose AGENTS.inAG is an inherited attribute, and AGENTS.outAG a synthesized attribute. Both attributes are of type Hashtable since semantic domain **Agents** is a function, which can be modeled as a Hashtable.
From $\mathcal{D}$: **Dec** → **State** → **State** (Table 4) it can be concluded that one inherited and one synthesized attributes need to be attached to non-terminal DECS. For this purpose DECS.inState is inherited attribute, and DECS.outState a synthesized attribute. Both attributes are of type Hashtable since semantic domain **State** is a function, which can be modeled as a Hashtable.
From $\mathcal{CM}$: **MeasPlace** → **Agents** → **Code** × **Integer** (Table 4) it can be concluded that one inherited and two synthesized attributes need to be attached to non-terminal MES_PLACES. Again, it was decided to represent both, a generated code and the identification number of virtual machine, as a string. For this purpose MES_PLACES.inAG is an inherited attribute and MES_PLACES.code is a synthesized attribute.
From $\mathcal{CS}$: **Stm** → **Agents** × **Integer** → **Code** (Table 4) it can be concluded that two inherited and one synthesized attribute need to be attached to non-terminal STMTS. For this purpose STMTS.inAG and STMTS.n are inherited attributes of type Hashtable and int, respectively. The attribute STMTS.code is a synthesized attribute of type String. The attributes, inherited and synthesized, attached to the appropriate non-terminals are collated in Table 8.

Table 8. Attributes of non-terminals representing syntactic domains from EasyTime formal semantics

X	Inherited(X)	Synthesized(X)
PROGRAM		code
AGENTS	inAG	outAG
DECS	inState	outState
MES_PLACES	inAG	code
STMTS	inAG, n	code

Step 3:

In this step semantic equations are given for each synthesized attribute of the left-hand side non-terminal, and for each inherited attribute for the right-hand side non-terminal. This procedure is applied to each production in the context-free grammar (Table 7). The LISA specification fragment as illustrated in Table 9 indicates, which semantic equations need to be developed. Let us explain the process for the first production. Since the non-terminal PROGRAM, left-hand side non-terminal, has only one synthesized attribute $code$ (Table 8) only one semantic equation must be defined (PROGRAM.code = ...;). Other non-terminals (AGENTS, DECS, MES_PLACES) in the first production are on the right hand side and hence only inherited attributes attached to those non-terminals must be defined (AGENTS.inAG = ...; DECS.inState = ...; MES_PLACES.inAG = ..;). Note that the order of these semantic equations is irrelevant [12, 20].

Table 9. Semantic equations under development that are obtained after Step 3

```
PROGRAM    ::= AGENTS DECS MES_PLACES compute {
                  AGENTS.inAG = ...;
                  DECS.inState = ...;
                  MES_PLACES.inAG = ...;
                  PROGRAM.code = ...; };

AGENTS     ::= AGENTS AGENT compute {
                  AGENTS[1].inAG = ...;
                  AGENTS[0].outAG = ...; };

DECS       ::= DECS DEC compute {
                  DECS[1].inState = ...;
                  DECS[0].outState = ...; };

MES_PLACES ::= MES_PLACE MES_PLACES compute {
                  MES_PLACES[1].inAG = ...;
                  MES_PLACES[0].code = ...; };

STMTS      ::= STMT STMTS compute {
                  STMTS[1].n = ...;
                  STMTS[1].inAG = ...;
                  STMTS[0].code = ...; };
```

Step 4:

From step 3, it can be identified the following non-terminals, which appears in concrete syntax (Table 7) and were unidentified in steps 1 - 3: AGENT, DEC, MES_PLACE, and STMT (Table 10). If the structure of these non-terminals is simple (e.g., AGENT, DEC) then attributes attached to these non-terminals carried only synthesized attributes representing mostly lexical values (Table 11). Semantic equations can be derived immediately for those attributes. On the other hand, some non-terminals might be complex (e.g., MES_PLACE, STMT) and inherited attributes attached to these non-terminals are also needed. The attributes might be similar to those attributes attached to other non-terminals in productions, where new non-terminals appear (Table 8). Moreover, semantic equations may no longer be simple (Table 11). For example, attributes attached to non-terminals MES_PLACE and STMT (Table 10) are the same as those attached to non-terminals STMTS and MES_PLACES, respectively (Table 8). However, due to the semantics of the update statement (Table 4) another attribute y is attached to the non-terminal STMT (Table 10).

Table 10. Attributes for additional non-terminals

X	Inherited(X)	Synthesized(X)
AGENT		number, type, file_ip
DEC		name, value
MES_PLACE	inAG	code
STMT	inAG, n	code, y

Step 5:

The reasoning of this step is only explained for semantic functions $\mathcal{A}$ and $\mathcal{CM}$ (Table 4), which are translated into attributes for non-terminals AGENTS, AGENT, MES_PLACES, and MES_PLACE (Tables 8 and 10). For other semantic functions the reasoning is similar. The semantic equation $\mathcal{A}[\![A_1; A_2]\!]ag$ = $\mathcal{A}[\![A_2]\!]$ $(\mathcal{A}[\![A_1]\!]ag)$ (Table 4) constructs $ag \in Agents$, which is a function from an integer, denoting an agent, into an agent's type (manual or auto), and an agent's ip or agent's file. This function is described in LISA as presented in Table 12. From Table 12 it can be noticed how the attribute $outAG$, which represents the $ag \in Agents$, is constructed simply by the calling method $insert()$. The method $insert()$ will insert a new agent with a particular number, type, and file_ip into the Hashtable. Note also, how the missing equations from Step 3 have been developed. The net effect is that we are constructing a list, more precisely a hash table, of agents where we are recording the agent's number ($AGENT.number$), the agents's type ($AGENT.type$), and the agent's ip or file ($AGENT.file_ip$) (see Step 4). The complete LISA specifications for semantic function $\mathcal{A}$, is shown in Algorithm 2.

The reasoning for the semantic function $\mathcal{CM}$ is done in a similar manner. The semantic equation $\mathcal{CM}[\![M_1; M_2]\!]ag$ = $\mathcal{CM}[\![M_1]\!]ag$: $\mathcal{CM}[\![M_2]\!]ag$ (Table 4) translates the first construct M_1 into code before performing the translation of the second construct M_2. This function is described in LISA, as repre-

Table 11. Semantic equations for additional non-terminals

```
AGENT        ::= #Int auto #ip   compute {
                    AGENT.number = Integer.valueOf(#Int[0].value()).intValue();
                    AGENT.type = "auto";
                    AGENT.file_ip = #ip.value(); };

DEC          ::= var #Id  #Int   compute {
                    DEC.name = #Id.value();
                    DEC.value = Integer.valueOf(#Int.value()).intValue(); };

MES_PLACES ::= MES_PLACE MES_PLACES compute {
                    MES_PLACE.inAG = ...; };

MES_PLACE   ::= mp [ #Int ] − > agnt [ #Int ] { STMTS } compute {
                    MES_PLACE.code= ...; };

STMTS        ::= STMT STMTS compute {
                    STMT.n = ...;
                    STMT.inAG = ...; };

STMT         ::= upd #Id   compute {
                    STMT.y = ...;
                    STMT.code = ...; };
```

Table 12. Semantic equation for AGENTS

```
AGENTS ::= AGENTS AGENT compute {
                AGENTS[1].inAG = AGENTS[0].inAG;
                AGENTS[0].outAG = insert(AGENTS[1].outAG,
                new Agent(AGENT.number, AGENT.type, AGENT.file_ip));
                }
              | epsilon compute {
                AGENTS.outAG = AGENTS.inAG;
                };
```

sented in Table 13, with the following meaning: The code for the first construct MES_PLACE is simply concatenated with the code from the second construct $MES_PLACES[1]$.

The semantic equation $\mathcal{CM}[\![\textbf{mp}[n_1] \rightarrow \textbf{agnt}[n_2]S]\!]ag = (WAIT\ i : \mathcal{CS}[\![S]\!](ag, n_2), n_1)$ (Table 4) is described in LISA, as presented in Table 14.

However, in this step the undefined semantic equations from steps 3 and 4 also need to be developed (Table 15). For example, a list of agents ($inAG$) needs to be propagated.

Step 6:

Easytime also uses variables in statements, and additional checks must be performed if only declared variables appear in expressions and statements. For this reason an additional attribute ok of type boolean has been introduced into the specifications. Moreover, to be able to check if a variable is declared, it

Algorithm 2 Translation of Agents into LISA specifications

```
1: rule Agents {
2:   AGENTS ::= AGENTS AGENT compute {
3:     AGENTS[1].inAG = AGENTS[0].inAG;
4:     AGENTS[0].outAG = insert(AGENTS[1].outAG,
5:      new Agent(AGENT.number, AGENT.type, AGENT.file_ip));
6:   }
7:   | epsilon compute {
8:     AGENTS.outAG = AGENTS.inAG;
9:   };
10: }
11: rule AGENT {
12:   AGENT ::= #Int manual #file  compute {
13:     AGENT.number = Integer.valueOf(#Int[0].value()).intValue();
14:     AGENT.type = "manual";
15:     AGENT.file_ip = #file.value();
16:   };
17:   AGENT ::= #Int auto #ip  compute {
18:     AGENT.number = Integer.valueOf(#Int[0].value()).intValue();
19:     AGENT.type = "auto";
20:     AGENT.file_ip = #ip.value();
21:   };
22: }
```

Table 13. Semantic equation for MES_PLACES

MES_PLACES ::= MES_PLACE MES_PLACES compute {	
	MES_PLACES[0].code = MES_PLACE.code +
	"\ n" + MES_PLACES[1].code; };
MES_PLACES ::= MES_PLACE compute {	
	MES_PLACES.code = MES_PLACE.code };

is necessary to propagate attribute $inState$ into the measuring places, statements, and expressions. The complete LISA specifications for MES_PLACE are shown in Algorithm 3 also using attributes ok and $inState$.

Semantic equations for other production are obtained in a similar manner. Let us conclude this example by finalizing semantic equations for the starting production (see also Table 9). The initial hash table for agents ($AGENTS.inAG$) and declarations ($DECS.inState$) are empty (Table 16). Agents and declarations are constructed after visiting the subtrees represented by the non-terminals $AGENTS$ and $DECS$, and stored into attributes $AGENTS.outAG$ and $DECS.outState$, that are passed to the subtree represented by the non-terminal MES_PLACES. If all the syntactic constraints are satisfied ($MES_PLACES.ok == true$), then the generated code is equal to a code produced by the subtree represented by the non-terminal MES_PLACES.

Table 14. Semantic equation for MES_PLACE

MES_PLACE ::= mp [#Int] − > agnt [#Int] { STMTS } compute {	
	MES_PLACE.code= "(WAIT i " + STMTS.code +
	", " + #Int[0].value() + ")"; };

Table 15. Developing undefined semantic equations for MES_PLACES

```
MES_PLACES ::= MES_PLACE MES_PLACES compute {
                    MES_PLACE.inAG = MES_PLACES[0].inAG;
                    MES_PLACES[1].inAG = MES_PLACES[0].inAG;
                    ... };
MES_PLACES ::= MES_PLACE compute {
                    MES_PLACE.inAG = MES_PLACES.inAG;
                    ... };
```

Table 16. Semantic equations for the starting production

```
PROGRAM ::= AGENTS DECS MES_PLACES compute {
                    AGENTS.inAG = new Hashtable();
                    DECS.inState = new Hashtable();
                    MES_PLACES.inAG = AGENTS.outAG;
                    MES_PLACES.inState = DECS.outState;
                    PROGRAM.code = MES_PLACES.ok ? "\ n" +
                    MES_PLACES.code + "\ n" : "ERROR"; };
```

5. Operation

Local organizers of sporting competitions were faced with two possibilities before developing EasyTime:

- to rent a specialized company to measure time,
- to measure time manually.

The former possibility is expensive, whilst the latter can be very unreliable. However, both objectives (i.e. inexpensiveness and reliability), can be fulfilled by EasyTime. On the other hand, producers of measuring devices usually deliver their units with software for the collecting of events into a database. Then these events need to be post-processed (batch processed) to get the final results of the competitors. Although this batch-processing can be executed whenever the organizer desires, each real-time application requests online processing. Fortunately, EasyTime enables both kinds of event processing.

In order to use the source program written in EasyTime by the measuring system, it needs to be compiled. Note that the code generation [1] of a program in EasyTime is performed only if the parsing is finished successfully. Otherwise the compiler prints out an error message and stops. For each of measuring places individually, the code is automatically generated by strictly following the rules, as defined in Section 3. An example of the generated code from the Algorithm 1 for the controlling of measurements, as illustrated by Fig. 1, is presented in Table 17. Note that the generated code is saved into a database. The meaning of the particular instructions of virtual machine (e.g., WAIT, FETCH, STORE), is explained in Table 5.

As a matter of fact, the generated code is dedicated to the control of an agent by writing the events received from the measuring devices, into the database. Normally, the program code is loaded from the database only once. That

Algorithm 3 Translation of MES_PLACE into LISA specifications

```
1: rule Mes_places {
2:   MES_PLACES ::= MES_PLACE MES_PLACES compute {
3:     MES_PLACE.inAG = MES_PLACES[0].inAG;
4:     MES_PLACES[1].inAG = MES_PLACES[0].inAG;
5:     MES_PLACE.inState = MES_PLACES[0].inState;
6:     MES_PLACES[1].inState = MES_PLACES[0].inState;
7:     MES_PLACES[0].ok = MES_PLACE.ok && MES_PLACES[1].ok;
8:     MES_PLACES[0].code = MES_PLACE.code + "\n" + MES_PLACES[1].code;
9:   };
10: MES_PLACES ::= MES_PLACE compute {
11:     MES_PLACE.inAG = MES_PLACES.inAG;
12:     MES_PLACE.inState = MES_PLACES.inState;
13:     MES_PLACES.ok = MES_PLACE.ok;
14:     MES_PLACES.code = MES_PLACE.code;
15:   };
16: }
17: rule MES_PLACE {
18:   MES_PLACE ::= mp \[ #Int \] \- \> agnt \[ #Int \] \{ STMTS \} compute {
19:     STMTS.inAG = MES_PLACE.inAG;
20:     STMTS.inState = MES_PLACE.inState;
21:     STMTS.n = Integer.valueOf(#Int[1].value()).intValue();
22:     MES_PLACE.ok = STMTS.ok;
23:     MES_PLACE.code = "(WAIT i " + STMTS.code + ", " + #Int[0].value() + ")";
24:   };
25: }
```

is, only an interpretation of the code could have any impact on the performance of a measuring system. Because this interpretation is not time consuming, it cannot degrade the performance of the system. On the other hand, the precision of measuring time is handled by the measuring device and is not changed by the processing of events. In fact, the events can be processed as follows:

– batch: manual mode of processing, and
– online: automatic mode of processing.

The agent reads and writes the events that are collected in a text file, when the first mode of processing is assumed. Typically, events captured by a computer timer are processed in this mode. Here, the agent looks for an existence of the event text file that is configured in the agent statement. If it exists, the batch processing is started. When the processing is finished, the text file is archived and then deleted. The online processing is event oriented, i.e. each event generated by the measuring device is processed in time. In both modes of processing, the agent works with the program PGM, the runner table RUNNERS, and the results table DATABASE, as can be seen in Fig. 3. An initialization of the virtual machine is performed when the agent starts. The initialization consists of loading the program code from PGM. That is, the code is loaded only once. At the same time, the variables are initialized on starting values.

In order to ensure the reliability of Easytime in practice, competitors are not allowed to go directly from swimming to running, because the course is complex and the competitor must to go through both transition areas. In the case that a competitor skips over the next discipline, the referees disqualify

Table 17. Translated code for the EasyTime program in Algorithm 1

```
(WAIT i FETCH accessfile(”abc.res”) STORE SWIM
FETCH ROUND1 DEC STORE ROUND1, 1)

(WAIT i FETCH accessfile(”abc.res”) STORE TRANS1, 2)

(WAIT i FETCH connect(192.168.225.100) STORE INTER2
FETCH ROUND2 DEC STORE ROUND2
PUSH 0 FETCH ROUND2 EQ BRANCH( FETCH
connect(192.168.225.100) STORE BIKE, NOOP), 3)

(WAIT i FETCH connect(192.168.225.100) STORE INTER3
PUSH 55 FETCH ROUND3 EQ BRANCH( FETCH
connect(192.168.225.100) STORE TRANS2, NOOP)
FETCH ROUND3 DEC STORE ROUND3
PUSH 0 FETCH ROUND3 EQ BRANCH( FETCH
connect(192.168.225.100) STORE RUN, NOOP), 4)
```

him/her immediately. Actually, EasyTime is only of assistance to referees. All misuses of the triathlons rules do not have any impact on its operation.

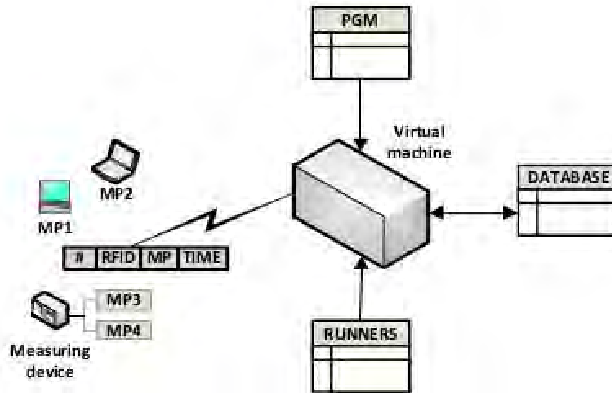


Fig. 3. Executable environment of a program in EasyTime

After the development of EasyTime another demand has arisen - drafting detection in triathlons. This problem is especially expressive in cycling, where competitors wishing to improve their results ride their cycles within close-knit groups. In this way, competitors achieve a higher speed and save energy for later efforts. Typically, within such groups of competitors the hardest work is performed by the leading competitor because he needs to overcome on air resistance. At the same time, other competitors may take a rest. Actually, the drafting violation arises when one competitor rides behind the other closer than 7 me-

ters for more than 20 seconds. Interestingly, this phenomenon is only pursued during long-distance triathlons, whilst drafting is allowed over short-distances. Any competitor who violates this drafting rule is punished by the referees with 5 minutes of elimination from the cycling race. The referees observe the race from motorcycles and determine the drafting violations according to their feelings. In this sense only, this assessment is very subjective. On the other hand, the referees can control one competitor a time. Consequently, an automatic system is needed for detecting drafting violations during triathlons. A drafting detection system is proposed in order to track this violation. This system is based on smart-phones because these incorporate the following features: information access via wireless networks and GPS navigation. Smart-phones need to be borne by competitors on their bicycles (Fig. 4). These determine information about competitor current GPS positions and transmit these over wireless modems to a web-service. From the positions of all competitors the web-service calculates whether a particular competitor is violating the drafting rule. In addition, these violations can be tackled by the referees on motorcycles using smart-phones.

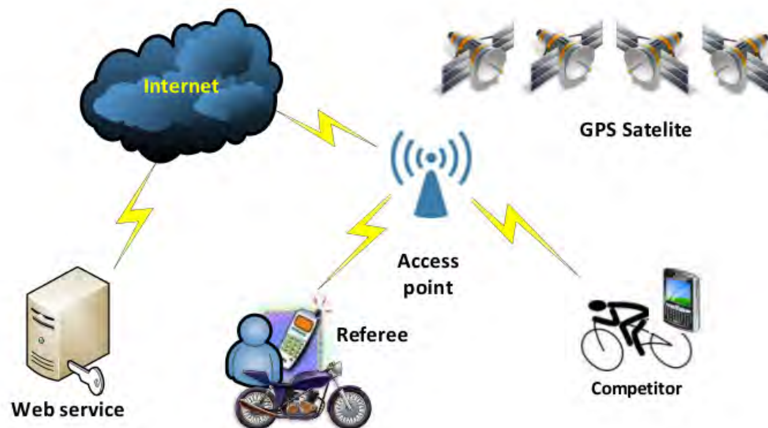


Fig. 4. Proposed system for drafting detection in triathlons

Normally, the organizers of triathlons demand the integration of EasyTime within the system for drafting violation. At a glance, this integration can be performed at the computer-system level, i.e., the mobile agent is added to the existing EasyTime agents. This mobile agent acts as a web-service and runs on an application server. Like EasyTime, it uses its own database. Each record in this database represents a competitor's current GPS position that can be defined as tuple $\langle \#, x, y, z, t, l \rangle$, where $\#$ denotes the competitor's starting number, x, y, z his current position within the coordinate system UTM, t the registration time in the mobile device, and l the calculated path-length. This length l is ob-

tained by projecting the current position of the #-th competitor on the line that connects the points gained by tracking the cycling course with a precise GPS device, at each second. This has an impact on the competitor's current position, from which the distance is calculated to the competitor in front of him. At the moment, both systems run on the same server separately. However, further development of a wireless technology and pervasive computing [29] indicates that EasyTime should have the ability to run on an application server as well.

Interestingly, the measuring time in biathlons represents another great challenge for EasyTime. Here, competitors ski on cross-country skiis and stop at certain places to shoot at targets with rifles carried by them. In order to measure time during biathlons, EasyTime needs to be modified slightly. In line with this, two measuring devices are need, and a special measuring device for counting hits. The first measuring device is dedicated to measuring the four laps of skiing, whilst the second is applied for counting the penalty laps. Each missed shot attracts one additional penalty lap. The measuring device for counting hits is described in EasyTime as a new agent. This agent is responsible for setting the number of additional penalty laps to be measured using the second measuring device. In contrast to the static initialization of the laps counter in EasyTime, a new request is demanded, i.e, a dynamic initialization of this laps counter needs to be implemented.

EasyTime could also be extended and used in some other application domains. For example, EasyTime could be employed as an electric shepherd for tracking livestock (cows, sheep, etc.) in the mountains. In this case, each animal would be labeled with a RFID tag that is controlled by crossing the measuring place twice a day. First, in the morning, when the animals go from their stalls and, second, in the evening, when they return to their stalls. Each crossing of the measuring place by the animal decrements a counter of herd-size for one. Essentially, the EasyTime tracking system reports an error, when the counter is not decreased to zero within a specified time interval. In order for this tracking system to work properly, the herd-size counter has to be initialized twice a day (for example, at 12:00 am and 12:00 pm). Additionally, EasyTime could be used in the clothing industry for tracking cloth through the production. Clothing production consists of the following phases: preparing, sewing, ironing, adjusting, quality-control and packing [7, 8]. The particular cloth origins during the preparation stage, where the parts of cutting patterns are collected into bundles, labeled with the RFID tags, and delivered for sewing. This transition of the bundle into the sewing room presents a starting point for the EasyTime tracking system. The other control points are, as follows: transition from sewing room into ironing, transition from ironing into adjusting, transition from adjusting into quality-control, and transition from quality-control into packing room that represents the finishing point of the cloth production. Note that these transitions act similarly to those transition areas in Ironman competitions. Usually, the cloth does not traverse through the production in any one-way because quality-control can return it to any of the past production phases. In this case, EasyTime could be used for tracking errors during clothing production.

6. Conclusion

The flexibility of the measuring system is a crucial objective in the development of universal software for measuring time in sporting competitions. Therefore, the domain-specific language EasyTime was formally designed, which enables the quick adaptation of a measuring system to the new requests of different sporting competitions. Preparing the measuring system for a new sporting competition with EasyTime requires the following: changing a program's source code that controls the processing of an agent, compiling a source code and restarting the agent. Using EasyTime in the real-world has shown that when measuring times in small sporting competitions, the organizers do not need to employ specialized and expensive companies any more. On the other hand, EasyTime can reduce the heavy configuration tasks of a measuring system for larger competitions, as well. In this paper, we explained how the formal semantics of EasyTime are mapped into LISA specifications from which a compiler is automatically generated. Despite the fact that mapping is not difficult, it is not trivial either, as some additional rules must be defined for attribute propagation. Moreover, we need to take care of error reporting (e.g., multiple declarations of variables). In future work, EasyTime could be replaced by the domain-specific modeling language (DSML) [24, 26, 28] that could additionally simplify the programming of a measuring system.

References

1. A.V. Aho and J.D. Ullman. *The theory of parsing, translation, and compiling*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1972.
2. P. Arpaia, L. Fiscarelli, G. La Commara, and C. Petrone. A model-driven domain-specific scripting language for measurement-system frameworks. *IEEE Transactions on Instrumentation and Measurement*, 60(12):3756–3766, 2011.
3. I. Attali and D. Parigot. Integrating natural semantics and attribute grammars: the minotaur system. Technical Report 2339, INRIA, 1994.
4. Championchip website, 2010.
5. A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10:1–17, 2002.
6. K. Finkenzeller. *RFID Handbook*. John Wiley & Sons, Chichester, UK, 2010.
7. I. Fister, M. Mernik, and B. Filipič. Optimization of markers in clothing industry. *Engineering Application of Artificial Intelligence*, 21(4):669–678, 2008.
8. I. Fister, M. Mernik, and B. Filipič. A hybrid self-adaptive evolutionary algorithm for marker optimization in the clothing industry. *Applied Soft Computing*, 10(2):409–422, 2010.
9. I. Jr. Fister, I. Fister, M. Mernik, and J. Brest. Design and implementation of domain-specific language Easytime. *Computer Languages, Systems & Structures*, 37(4):276–304, 2011.
10. I. Jr. Fister, M. Mernik, I. Fister, and D. Hrnčič. Implementation of the domain-specific language easy time using a LISA compiler generator. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 809–816, Szczecin, Poland, 2011.

11. P. Hudak. Building domain-specific embedded languages. *ACM computing surveys*, 28, 1996.
12. D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
13. T. Kos, T. Kosar, and M. Mernik. Development of data acquisition systems by using a domain-specific modeling language. *Computers in industry*, 63(3):181–192, 2012.
14. T. Kosar, M. Mernik, and J.C. Carver. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering*, 17(3):276–304, 2012.
15. I. Lukovič, M.J. Varanda Pereira, N. Oliveira, D. da Cruz, and P.R. Henriques. A DSL for PIM specifications: Design and attribute grammar based implementation. *Computer Science and Information Systems*, 8(2):379–403, 2011.
16. S. Mauw, W. Wiersma, and T. Willemse. Language-driven system design. *International Journal of Software Engineering and Knowledge Engineering*, 14:1–39, 2004.
17. M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM computing surveys*, 37(4):316–344, 2005.
18. M. Mernik, M. Lenič, E. Avdičauševič, and V. Žumer. Lisa: an interactive environment for programming language development. In *11th International Conference Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 1–4, 2002.
19. M. Mernik and V. Žumer. Incremental programming language development. *Computer Languages, Systems and Structures*, 31(1):1–16, 2005.
20. J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.
21. S. Pakin. The design and implementation of a domain-specific language for network performance testing. *IEEE Transactions on Parallel and Distributed Systems*, 18(10):1436–1449, 2007.
22. L. Paulson. A semantics-directed compiler generator. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 224–233, 1982.
23. Rfid timing system website, 2010.
24. J. Sprinkle, M. Mernik, J-P. Tolvanen, and D. Spinellis. What kinds of nails need a domain-specific hammer? *IEEE Software*, 26(4):15–18, 2009.
25. V. Štuikys and R. Damaševicius. Measuring complexity of domain models represented by feature diagrams. *Information Technology And Control, Kaunas, Technologija*, 38(3):179–187, 2009.
26. V. Štuikys, R. Damaševicius, and A. Targamadze. A model-driven view to metaprogram development process. *Information Technology And Control, Kaunas, Technologija*, 39(3):89–99, 2010.
27. M. Viroli, J. Beal, and M. Casadei. Core operational semantics of proto. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 1325–1332, New York, NY, USA, 2011. ACM.
28. R. Vitiutinas, D. Silingas, and L. Telksnys. Model-driven plug-in development for uml based modeling systems. *Information Technology And Control, Kaunas, Technologija*, 40(3):191–201, 2011.
29. M. Weiser. The computer for the 21st century. *Scientific American*, 3:94–104, 1991.

Iztok Fister Jr. is a first-year post-graduate student in Computer Science and Information Technologies at the Faculty of Electrical Engineering and Computer Science, University of Maribor. Besides his study and research activities, especially in the field of web-oriented programming, he is an enthusiastic competitor in triathlons. He is a student member of IEEE.

Mernik Marjan received his M.Sc., and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also a visiting professor at the University of Alabama in Birmingham, Department of Computer and Information Sciences, and at the University of Novi Sad, Faculty of Technical Sciences. His research interests include programming languages, compilers, domain-specific (modeling) languages, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Iztok Fister graduated in computer science from the University of Ljubljana in 1983. In 2007, he received his Ph.D. degree from the Faculty of Electrical Engineering and Computer Science, University of Maribor. Since 2010, he has worked as a Teaching Assistant in the Computer Architecture and Languages Laboratory at the same faculty. His research interests include computer architectures, program languages, operational research, artificial intelligence, and evolutionary algorithms. He is a member of IEEE.

Dejan Hrnčič received his B.Sc. degree from the Faculty of Electrical Engineering and Computer Science, University of Maribor, in 2007. Currently he is working on his Ph.D. thesis in computer science. His research interests include evolutionary computation, grammatical inference, and optimization techniques.

Received: November 10, 2011; Accepted: March 6, 2012.

DOI:10.2298/CSIS111216033D

Using Aspect-Oriented State Machines for Detecting and Resolving Feature Interactions

Tom Dinkelaker[1], Mohammed Erradi[2], and Meryeme Ayache[2]

[1] Ericsson R&D, Frankfurt, Germany
tom.dinkelaker@ericsson.com
[2] Networking & Distributed Systems Research Group, TIES, SIME Lab, ENSIAS, Mohammed V-Souissi University, Rabat, Morocco
erradi@ensias.ma, meryemeayache@gmail.com

Abstract. Composing different features in a software system may lead to conflicting situations. The presence of one feature may interfere with the correct functionality of another feature, resulting in an incorrect behavior of the system. In this work we present an approach to manage feature interactions. A formal model, using Finite State Machines (FSM) and Aspect-Oriented (AO) technology, is used to specify, detect and resolve features interactions. In fact aspects can resolve interactions by intercepting the events which causes troubleshoot. Also a Domain-Specific Language (DSL) was developed to handle Finite State Machines using a pattern matching technique.

Keywords: feature interactions, aspect interactions, aspect-oriented programming, state machines, conflict detection, conflict resolution, object-oriented programming, formal methods, domain-specific aspect languages.

1. Introduction

An important problem in modeling and programming languages is handling *Feature Interactions*. When composing different features in a software system, these may interact with each other. This can lead to a conflicting situation, where the presence of one feature may interfere with the correct functionality of another feature, resulting in an incorrect behavior of the system. Various techniques have been explored to overcome this problem. Among them, formal approaches have received much attention as a means for detecting feature interactions in communication service specifications.

In Software Product-Line (SPL) engineering [1], [2], the designer decomposes a software system into functional features by creating a feature model [1], [3]. But a feature model can only define a set of features and known interactions between them. Feature models do not help, when the designer overlooks a feature interaction – especially at the implementation level.

Aspect-Oriented Programming (AOP) [4] uses a special kind of modules called aspects that supports localization of code from crosscutting features. AOP has been extended with special language concepts for controlling aspect interactions [5], [6], but AOP does not support controlling feature interactions with modules that are not aspects in particular objects.

To address the above problems, in this work we propose a formal approach which uses an extension to finite state machines as the formalism for behavioral specification. The central idea behind using finite state machines as specification models is to have a strong mean to envision feature interactions. The formalism defines a process, which consists of the following steps: First, the developer gives a formal specification of each feature that extends the system's core feature, even partial specifications are allowed. Second, using a suitable composition mechanism for FSMs (e.g., the FSM's synchronized cross-product [7]), the developer makes a parallel composition of the selected feature specifications and analyzes this composition. Third, the developer can identify conflicting states by analyzing the composed specification of the global system. Forth, to resolve feature interactions, the approach uses aspect-oriented state machines to intercept, prevent, and manipulate events that cause conflicts. We suggest a new formalism for aspect oriented state machines (AO-FSM) where pointcut and advice are used to adopt Domain-Specific Language (DSL) [8] state machine artifacts. The advice defines a state and transition pattern that it applies at the selected points, i.e. it may insert new states and transitions as well as it may delete existing ones.

2. Case Study: Telecommunication Systems

2.1. Plain Old Telephone Service (POTS)

Features in Telecommunication systems are packages providing services to subscribers. The Plain Old Telephone System (POTS) is considered as a feature providing basic means to set up a conversation between subscribers. In the following we provide the design and the specification of the basic service of a telephone system (POTS). We assume that a phone is identified by a unique number, and it can be either calling or being called.

In this specification, there are three objects that constitute the telephone system: the "user", the "agent" and the "call" as shown in Fig. 1. According to our semantics, the instantiation of these objects provides three objects running in parallel. The communication between objects is based on operation calls using a rendezvous mechanism. Note that the behavior part of these objects is specified using a finite state machine model.

Fig. 1 partially specifies the behavior of the system using finite state machines (see section 4 for a more detailed presentation of the formalism). This system works as follows: Once the caller (user-1, an instance of the User

behavior of Fig. 1) picks up (offhook) his phone (Agent-1, an instance of the Agent automaton), the network (designated by the object "call") responds by sending a tone. This user is then ready to dial the telephone number of the called party (using the operation "dial") using a standard telephone interface (Fig. 2). Then the network sends back a signal (operation "Ring") which causes a ring on the called phone (Agent-2, another instance of Agent). An Echo_ring is then sent to the caller (operation Echo_ring). We assume that the called user is always ready to answer a call. When the called user picks up (offhook) his phone, the ring is then interrupted and the two users engage in a conversation.

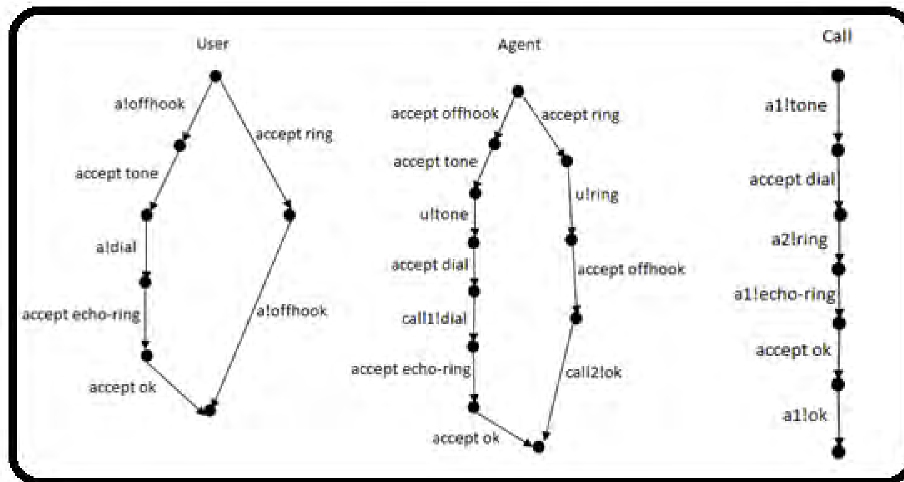


Fig. 1. Partial automata specifying the three objects

Flash-hook button

Fig. 2. Standard telephone interface with a flash-hook button (labeled with “R”)

2.2. Features available for User Selection (User Services)

According to the definition provided by Pamela Zave [9]: "in a software system, a feature is an increment of functionality, usually with a coherent purpose. If a system description is organized by features, then it probably takes the form B + F_1 + F_2 + F_3 . . ., where B is a base description, each F_i is a feature module, and + denotes some feature-composition operation". Therefore, telecommunication software systems have been designed in terms of features. So different customers can subscribe to the features they need. Many features can be enabled or disabled dynamically by their subscribers. Among the telecommunications features provided by a telephone system we found: Call Waiting, Three Way Calling, Call Forwarding, and Originating Call Screening.

2.2.1 Call Waiting (CW)

A Call Waiting feature (CW) is a service added to the basic service POTS described earlier. It allows a subscriber *A* (having the service CW) already engaged in a communication with a user *B* to be informed if another user *C* tries to reach him. *A* can either ignore the call of *C*, or press a flash_hook button to get connected to *C*. In other words, if *C* makes a call to *A*, while *A* is in communication with *B*, then *C* receives an Echo_ring, as if *A* was available, and *A* receives an "on hold" signal. Then *A* could switch between *B* and *C* by pressing the flash_hook button. If *B* or *C* hangs up, then *A* will be in communication with the user still on line. The basic service POTS to which is added the Call Waiting feature is symbolically designated by POTS + CW.

2.2.2 Three Way Calling (TWC)

The Three Way Calling is a service which extends the basic service POTS. It allows three users *A*, *B* and *C* to communicate in the following way: Consider a subscriber *A* (having the TWC feature) who is communicating with *B*. *A* can then add *C* in the conversation. To reach this goal, *A* put first *B* on hold by pressing a button flash hook button. Then, establish a communication with *C*. And finally, press the flash hook button again, to get, *A*, *B* and *C* connected. *A* can remove *C* from the conversation by pressing the flash hook button. If *A* hangs up, *B* and *C* remain in communication. The basic service POTS to which is added the Three Way Calling feature is symbolically designated by POTS + TWC.

2.2.3 Call Forwarding on Busy (CFB)

Call forwarding on busy is a feature on some telephone networks that allows an incoming call to a called party, which would be otherwise unavailable, to be redirected to another telephone number where the desired called party is situated.

2.2.4 Originating Call Screening (OCS)

The OCS Feature allows a user to define a list of subscribers hoping to screen outgoing calls made to any number in this screening list. A user *A* (with the OCS feature) who registered user *B* on the list will no longer make a call to *B*, but *B* could call *A*.

2.3. Feature Interactions

Feature interactions could be considered as all interactions that interfere with the desired operation of a feature and that may occur between a feature and its environment, including other features. Therefore, a feature interaction may refer to situations where a combination of different services behaves differently than expected.

For instance, pressing a “tap” button can mean different things depending on which feature is anticipated. This is the case of a flash-hook signal (generated by pressing such button) issued by a busy party could mean adding a third party to an established call (Three Way Calling) or to accept a connection attempt from a new caller while putting the current conversation on hold (Call Waiting). Should the flash hook be considered the response of Call Waiting, or an initiation signal for Three-Way Calling?

Another feature interaction may occur if we consider a situation where a user *A* has subscribed to the Originating Call Screening (OCS) feature and screens calls to user *C*. Suppose that a user *B* has activated the service Call Forwarding (CF) to user *C*. In this situation, if *A* calls *B*, the intention of OCS not to be connected to *C* will be violated since the call will be established to *C* by way of *B*.

Usually, the causes of interactions may be due to the violation of assumptions related to the feature functionality, to the lack of a technical support from the network, or to problems related to the distributed implementation of a feature. Despite the lack of a formal definition of a feature interaction due to the diversity of the interactions types, the reader will find a detailed taxonomy of the features interactions in [10].

Our approach to process the feature interaction problem consists in two methods based on formal techniques. The first method is used to detect the interactions while the second resolves them. In the context of formal techniques, interactions are considered as "conflicting statements". This may

be a deadlock, a non-determinism, or constraints violation which may result from states incompatibility between two interacting features. The incompatibility between states can be detected using a "Model-Checking" technique.

3. Problem Statement

Feature interaction is considered a major obstacle to the introduction of new features and the provision of reliable services. In practical service development, the analysis of interactions has often been conducted in an ad hoc manner.

However, the feature interactions problem is not limited to the telecommunications domain. The phenomenon of undesirable interactions between components of a system can occur in any software system that is subject to changes. This is certainly the case for service-oriented architectures. First, we can observe that interaction is at the very basis of the web services concept. Web services need to interact, and useful web services will emerge from the interaction of more specialized services. Second, as the number of web services increases, their interactions will become more complex. Many of these interactions will be desirable, but others may be unexpected and undesirable, and we need to prevent their consequences from occurring.

There is a broad body of research that addresses the problem of feature interactions. However, as elaborated in the following, there are important limitations how the state of the art can detect and resolve feature interactions.

3.1. OOP cannot localize crosscutting Code of Features

Object-oriented programming (OOP) enables a hierarchical decomposition of the system into classes that can be extended by other classes. Using an OO language, developers can completely describe the system behavior in form of an implementation that can be executed. However, standard OO languages (such as Java or C++) do not provide special means to control feature interactions at the implementation level.

Furthermore, there are certain features for which OOP does not allow a good *Separation of Concerns* [12] because their implementation is scattered over several classes and tangled with the implementation of other features [4]. Examples for such features are non-functional components like tracing, billing calls, or feature interaction resolution.

Because OO languages do not have the right means to implement features and manage interactions among them, developers are left alone with implementing the logic that handles crosscutting and feature interactions, which results in code that is hard to understand and maintain.

3.2. Feature Models can only detect anticipated Feature Interactions

Feature models (FMs) [3] are a well-known technique to model the functionality of a system. They also allow prevent interactions between features that the developer is already aware of.

For example, Fig. 3 shows an FM for the telephone system, which does not only model features selected by the user, but choices made by the vendor. The telephone system has abstract features, such as a platform, a user interface, a receiving call indicator, and a set of user services, which can be implemented by a choice of features. The telephone platform can be either analog or digital (exclusive-or). If it is analog, then there can be a digital display. It must have a bell (mandatory feature), and in addition, it may have a LED (optional feature) that indicates receiving calls e.g. when the volume of the bell is low. The user may choose from the set of services from Section 2.2 (inclusive-or).

To model constraints of valid configurations that cannot be expressed using exclusive or inclusive or, the developer uses feature constraints. For example, the feature model in Fig. 3 defines that the features CW and TWC *requires* a flash_hook button has to be selected as well. In contrast, when selecting an analog platform, this *excludes* selecting a digital display.

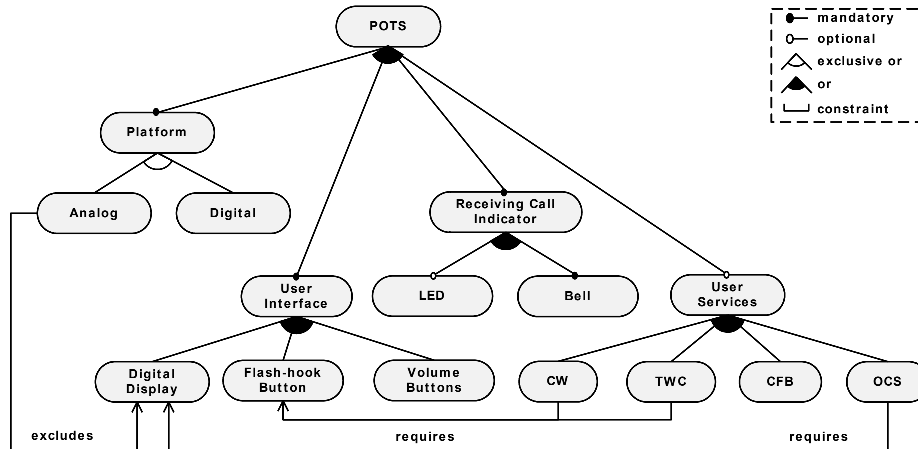


Fig. 3. A feature model of the telephone system (POTS)

An FM allows checking a particular selection of features, which is called a configuration, whereby a tool validates that all modeled feature constraints are met. However, an FM cannot guarantee that there is no feature interaction at the implementation-level. In case, the developer overlooks an interaction and does not model it correctly in the FM. In FMs, there is no support for formal behavioral modeling. Consequently, with FMs, developers cannot analyze the combined behavior of the features and for possible interactions between them.

3.3. AOP can only detect Aspect Interactions

Aspect-oriented programming (AOP) enables developers to modularize such non-functional concerns in OO languages. Important AOP concepts are *pointcut*, *join point model*, and *advice*. Pointcuts are predicates over program execution actions called join points. That is, a pointcut defines a set of join points related by some property; a pointcut is said to be triggered or to match at a join point, if the join point is in that set. It is also common to speak about join points intercepted by a pointcut. Such a join-point model (JPM) characterizes the kinds of execution actions and the information about them exposed to pointcuts (e.g. a method call). An Advice is a piece of code associated with a pointcut, it is executed whenever the pointcut is triggered, thus implementing crosscutting functionality. There are three types of advice, *before*, *after*, and *around*; relating the execution of advice to that of the action that triggered the pointcut the advice is associated with. The code of an around advice may trigger the execution of the intercepted action by calling the special method *proceed*.

However, there is a lack of a general approach to weave on code fragments of DSLs. The problem is that current AOP tools support only one JPM at a time, which is for most aspect-oriented (AO) languages one JPM for the events in the execution of an OO language [4]. Only for some DSLs, there is a domain-specific aspect language with a domain-specific JPM [13] (e.g. encompassing join points like a state transition in a state machine). Still, current AOP tools do not provide support for special quantifications for weaving aspects into programs written in several languages that have different kinds of join-point models.

For example, consider implementing a logging feature as an aspect that needs to be woven into the code of several languages for debugging, such as it need to be woven into code in Java with an Aspect-like JPM, code in SDL[1] that defines a JPM for FSMs, and code in LOTOS[2] that defines a JPM on top of protocols as communicating processes.

4. Characterization of Aspect-Oriented FSMs

In this paper, we propose a new formalism for aspect-oriented state machines (AO-FSM) which is based on finite-state machines and the Essential Behavioral Model. An AO-FSM defines a set of states and transitions like a FSM, but states and transitions do not need to be completely specified. Developers can selectively omit states, transitions, and labels, and therefore

[1] SDL: Specification and Definition Language: http://www.sdl-forum.org/SDL/index.htm

[2] LOTOS: Language Of Temporal Ordering Specification: http://language-of-temporal-ordering-specification.co.tv/

constitutes a partial FSM in which parts are missing so that it can be used as a pattern for matching against other FSMs and for manipulating them.

4.1. The Basic Finite State Machines (FSMs) Model

An automaton with a set of states, and its "control" moves from state to state in response to external "inputs" is called a Finite State Machine (FSM). A Finite State Machine provides the simplest model of a computing device. It has a central processor of finite capacity and it is based on the concept of state. It can also be given a formal mathematical definition. Finite State Machines are used for pattern matching in text editors, for compiler lexical analysis, for communication protocols specifications [15]. Another useful notion is the notion of the non-deterministic automaton. We can prove that deterministic finite State Machine, DFSM, recognize the same class of languages as Non-Deterministic Finite State Machine (NDFSM), i.e. they are equivalent formalisms.

Definition 1: *A non-deterministic Finite State Machine is defined by a quadruplet* $\langle Q, \Sigma, \delta, q_0 \rangle$ *where Q is a set of states, Σ is an alphabet, δ is the transition function, and* q_0 *is the initial state. The transition function is* $\delta: Q \times \Sigma \rightarrow 2^Q$ *where* 2^Q *is the set of subsets of Q.*

An event $\sigma \in \Sigma$ is accepted out from a state $q \in Q$ if the occurrence of σ is possible from the state q, i.e. if δ(q,σ) is not empty, we denote this by δ(q,σ)! When δ(q,σ) is empty, we write δ(q,σ)¬!. We consider a blocking state q (deadlock) if no transition is possible from this state. Formally: q is blocking $\Longleftrightarrow \forall \sigma \in \Sigma$, δ(q σ)¬!.

Definition 2: *A deterministic finite state machine is defined by a quadruplet* $\langle Q, \Sigma, \delta, q_0 \rangle$ *and corresponds to a particular case of the non-deterministic finite state machine where for any q and for any event σ, δ(q,σ) is either the empty set or a singleton. When δ(q,σ) is not empty, δ(q,σ) = {r} will be simply noted δ(q, σ) = r.*

The definitions introduced above refer to the basic formal model, but the actual notations used in our system modeling extend this model with other features in order to make it more practical and to support the requirements of our approach. Among these extensions we find: nested states, dependencies between states, and propositions. Therefore nested states, as shown in Fig.4, will be used to allow for partial automata modeling that hide parts of an automaton. Such partial specification hides the states and transitions which are not concerned by the composition and will not lead to an interaction. The dependencies between states allow indicating the order of occurrence of a given transition within different features. The propositions could be used as

guards to characterize a given state according to the value of defined variables within such state.

Fig.1 gave an example of FSM. In addition, we give here additional examples concerning the partial finite state machines corresponding to the Call Waiting (CW) and Three Way Calling (TWC) features:

A partial formal specification of *POTS+CW is a FSM* F_{CW} shown in Fig. 4. The shown states Q_i, for i=1 to 5, have the following semantics:

- Q_1: *A* and *B* are connected and start communicating.
- Q_2: *A* and *B* are communicating, then a call from *C* occurs on the switch of *A*.
- Q_3: *A* and *B* are communicating, and *A* receives the signal *call-waiting* indicating that someone is calling.
- Q_4: *B* is waiting, *A* and *C* are communicating.
- Q_5: *C* is waiting, *A* and *B* are communicating.

The events E_i, for i=1 to 3, have the following semantics:

- E_1: a call from *C* arrived on the switch of *A*.
- E_2: *A* receives the signal *call-waiting* indicating that someone else is calling.
- E_3: *A* pushes the *flash_hook* button.

A partial formal specification of POTS+TWC is the FSM F_{TWC} shown in Fig. 4. The states R_i, for i=1 to 4, have the following semantics:

- R_1: *A* and *B* are communicating.
- R_2: *B* is waiting.
- R_3: *B* is waiting, *A* and *C* are communicating.
- R_4: *A*, *B* and *C* are communicating.

The event E3 has already been defined for the specification POTS + CW. The event E_4 has as its semantics:

- E_4: *A* is communicating with *C*.

Note that the states "in bold" Q_1 and R_1 represent nested FSM. For instance this means that the state Q_1 corresponds to a FSM which is a portion of the global specification, nested in this state Q_1.

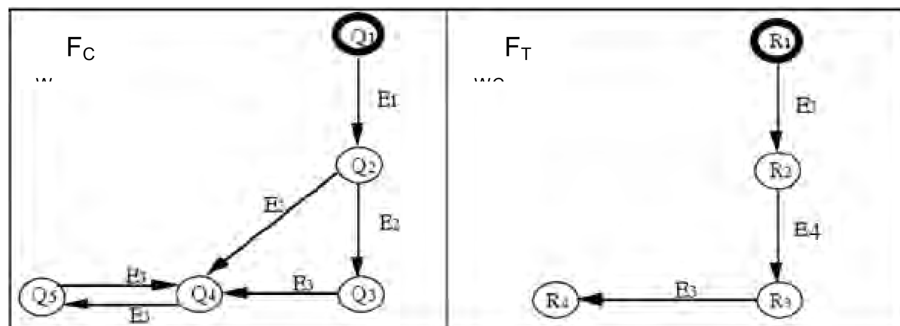


Fig. 4. Specification F_{CW} (left) and specification F_{TWC} (right)

4.2. Aspect-Oriented Finite State Machines (AO-FSM)

In an AO-FSM aspect, there are two parts: a *pointcut* and *advice* – like in other aspect-oriented languages for GPLs, but our pointcut and advice adapt DSL state machine artifacts. There is a composition model that defines how aspects at the meta-level are composed with base-level state machines, which is elaborated in the following.

An AO-FSM pointcut defines a state and transition pattern that selects all FSMs that the advice adapts. This pattern (at the meta-level) describes a model that needs to be observed on the execution of other (base-level) state machines. When the first part of the pattern is observed, in other words when a base-level state machine enters the initial state of the pointcut's state machine, our execution model creates a new meta-level instance of the state machine that describes the pattern and monitors the further execution of the base state machine. When observing new events at the base level, our composition model updates the meta-level instance in parallel to executing and updating the base-level instance. Finally, when the meta-level instance enters a *final state*, this means that the pattern has been recognized. In contrast, whenever the pointcut state machine does not define a matching transition for one of the observed events, the meta-level instance is deleted and garbage collected, since the pattern can no longer be fulfilled.

The advice defines a state and transition pattern that it applies at the selected points in the base-level state machine, i.e. it may insert new states and transitions as well as it may delete existing ones. As long as the meta-level instance remains in the final state, the advice is active, i.e. the changes are applied.

Fig. 5 shows visual models of all types of AO-FSMs. The upper row enumerates all pointcut types (alphabetic indices), in which only the shown parts define the pattern and omitted parts match like wildcards. The lower row enumerates all advice types (roman indices), in which only the bold parts adapt the corresponding parts of an FSM. When constructing an AO-FSM aspect, the different types of pointcut and advice types can be composed.

There are 6 different kinds of pointcuts: a) matches states with a particular label S_p, b) matches any state regardless of its label, c) matches a state in which a certain preposition p_1 is true, d) matches a state that has an incoming transition with label E_o, e) matches a state with an outgoing transition with a label E_q, and f) matches a sequence of two states with a transition that has the label E_r.

There are 8 different kinds of advice: i) inserts a new transition for event E_s, ii) inserts a new state S_t, iii) adds a new proposition p_2 to a state, iv) defines a dependency constraint c_2 between two states or two transitions, v) deletes the transition for event E_u, vi) deletes the state S_v, vii) deletes the property p_3, and finally, viii) defines a conflicting composition that results in an error message.

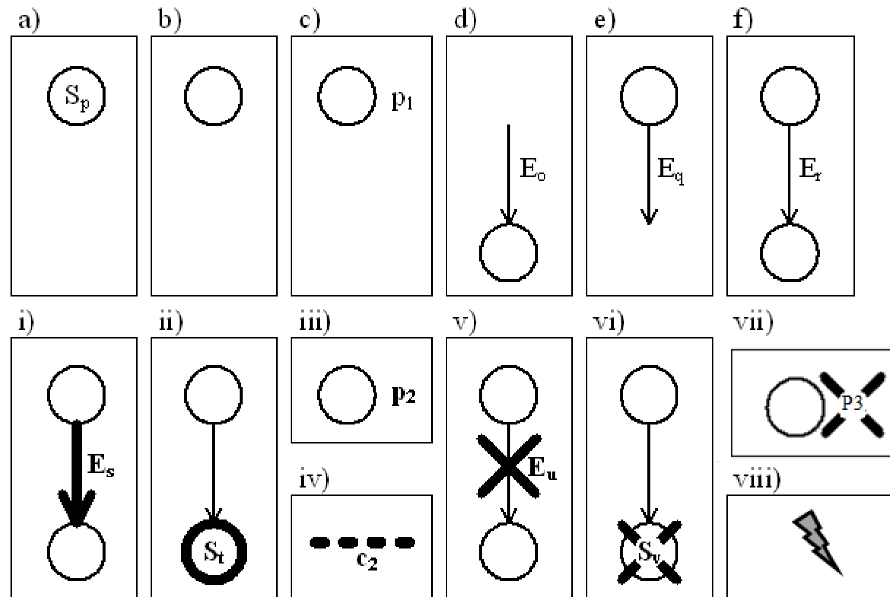


Fig. 5. Pointcut and advice types in aspect-oriented finite state machines

Usually, pointcut and advice compose the above basic patterns to more complex ones. For example, if we need a composite pointcut that matches a particular state Sp with an incoming transition with label Eo, then we would combine the basic patterns a) and f). For example, if we need a composite advice that adapts an existing state Sp by adding new transition Es to a new state St, then we would combine the basic patterns a), i) and ii). With these compositional semantics, rich adaption scenarios can be modeled.

To weave an aspect, we match all pointcuts and apply all advice for all FSMs. For a single FSM, the pointcut matches at every point in the FSM and applies the advice at each of these points. The adapted FSMs are then used for execution.

5. Resolving Feature Interactions with AO-FSMs

To control feature interactions, developers uses aspects to analyze and manipulate the behavior of a system that they compose from a set of modular feature specifications. In a nutshell, they compose specifications into a global behavior which we call an Essential Behavioral Model (EBM) of the system. The EBM consists of nested state machines that describe the composition of all features in the system. In the beginning, the EBM may expose feature interactions. To achieve a conflict-free composition of the features, developers use AO-FSM aspects to detect interactions that manifest singularities in the composed specification. There are three possible

singularities: 1) the composed EBM is non-deterministic, 2) the composed EBM has contradicting prepositions, or 3) the composed EBM has blocking states. The main advantage of our approach is that feature interactions can be directly identified from the model. Finally, the developer can resolve feature interactions by eliminating singularities using AO-FSM aspect.

5.1. The Composition Mechanisms

A composition mechanism models the way in which features are composed together to yield a single FSM model of the system. In this section we present two possible composition mechanisms: the synchronized product and the exclusive sum. Such operations are defined as follows[3]:

Definition 4: *Consider two FSMs* $A=\langle Q_A, \Sigma_A, \delta_A, q_{A0}\rangle$ *and* $B=\langle Q_B, \Sigma_B, \delta_B, q_{B0}\rangle$. *Let* Ω *be a subset of* Σ_A *and* Σ_B, *in other words* $\Omega \subseteq \Sigma_A \cap \Sigma_B$. ***The Synchronized Product of A and B***, *according to* Ω, *is a FSM represented by* $A*B[\Omega] = \langle Q, \Sigma, \delta, q_0\rangle$ *defined formally as follows:*

– $Q \subseteq Q_A \times Q_B$, $\Sigma = \Sigma_A \cup \Sigma_B$, $q_0 = (q_{A0}, q_{B0})$
– $\forall q=\langle q_A, q_B\rangle \in Q, \forall \sigma \in \Omega$:
  $(\delta(q,\sigma)!) \Longleftrightarrow (\delta_A(q_A,\sigma_A)! \wedge \delta_B(q_B,\sigma_B)!)$
  $(\delta(q,\sigma)!) \Rightarrow (\delta(q,\sigma)) = (\delta_A(q_A,\sigma) \times \delta_B(q_B,\sigma))$
– $\forall q = \langle q_A, q_B\rangle \in Q, \forall \sigma \notin \Omega$:
  $(\delta(q,\sigma)!) \Longleftrightarrow (\delta_A(q_A,\sigma_A)! \vee \delta_B(q_B,\sigma_B)!)$
  $(\delta(q,\sigma)!) \Rightarrow (\delta(q,\sigma) = (\delta_A(q_A,\sigma) \times \{q_B\}) \cup (\{q_A\} \times \delta_B(q_B,\sigma))$

Intuitively, if *A* and *B* specifies two processes, then $A*B[\Omega]$ is the global specification of the two processes composed in parallel and have to synchronize on Ω's actions. By $A \otimes B[\Omega]$ we will note the product of the automaton *A* and *B* obtained by removing the blocking states from the *Synchronized Product* $A*B[\Omega]$. When Ω is empty, the two processes are said to be independent and their product is called the cross-product of A and B. It is denoted by $A*B[]$. When $\Omega = \Sigma_A \cap \Sigma_B$, their product is denoted $A*B$.

Definition 5: *(Sum of two FSMs, the Extension Relationship)*
Consider two FSMs $A=\langle Q_A, \Sigma_A, \delta_A, q_{A0}\rangle$ *and* $B=\langle Q_B, \Sigma_B, \delta_B, q_{B0}\rangle$. *The extension relation of A and B is a FSM defined formally as follows:*

– $Q \subseteq (Q_A \times Q_B) \cup Q_A \cup Q_B$, $\Sigma = \Sigma_A \cup \Sigma_B$, $q_0 = (q_{A0}, q_{B0})$

[3] Recall that a negated exclamation mark $(\delta_i\ (q_i,\ \sigma)\neg!)$ means that there is no transition defined, while an exclamation mark $(\delta_i\ (q_i,\ \sigma)!)$ means that there is a transition defined.

– $\forall q=\langle q_A,q_B\rangle \in Q \cap (Q_A \times Q_B),\ \forall \sigma \in \Sigma$:
$(\delta(q,\sigma)!) \Longleftrightarrow (\delta_A(q_A,\sigma)! \vee \delta_B(q_B,\sigma)!)$
$(\delta_A(q_A,\sigma)! \wedge \delta_B(q_B,\sigma)\neg!) \Rightarrow \delta(q,\sigma) = \delta_A(q_A,\sigma)$
$(\delta_A(q_A,\sigma)\neg! \wedge \delta_B(q_B,\sigma)!) \Rightarrow \delta(q,\sigma) = \delta_B(q_B,\sigma)$
$(\delta_A(q_A,\sigma)! \wedge \delta_B(q_B,\sigma)!) \Rightarrow (\delta(q,\sigma)) = (\delta_A(q_A,\sigma) \times \delta_B(q_B,\sigma))$

– $\forall q=q_A \in Q \cap Q_A,\ \forall \sigma \in \Sigma$:
$(\delta(q,\sigma)!) \Longleftrightarrow (\delta_A(q_A,\sigma)!)$
$(\delta(q,\sigma)!) \Rightarrow \delta(q,\sigma) = \delta_A(q_A,\sigma)$

– $\forall q=q_B \in Q \cap Q_B,\ \forall \sigma \in \Sigma$:
$(\delta(q,\sigma)!) \Longleftrightarrow (\delta_B(q_B,\sigma)!)$
$(\delta(q,\sigma)!) \Rightarrow \delta(q,\sigma) = \delta_B(q_B,\sigma)$

Intuitively, if *A* and *B* specify two processes, then $A \oplus B$ is the global specification of the two processes behaving exclusively.

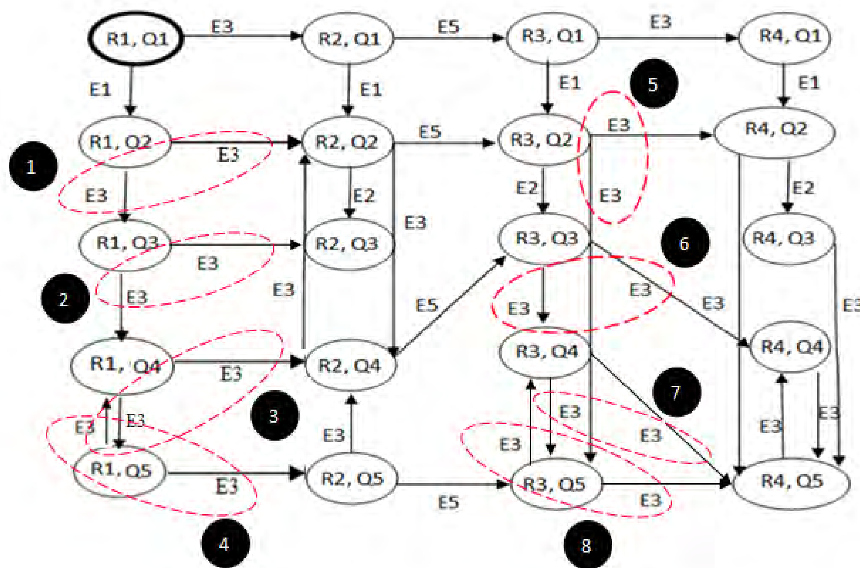


Fig. 6. Cross product of F_{CW} and F_{TWC}

As an example of a composition, Fig. 6 shows the result of the cross product of the FSMs corresponding to the features Call Waiting and Three Way Calling (shown in Fig. 4). We can observe the presence of non-determinism at index i for i=1,2,3,4,5,6,7,8, which is illustrated by having at least two transitions for one event going out from the same state and leading to two different states. There are 8 cases of non-determinism in Fig. 6, which are indexed

and high-lighted with the ellipses. For example, the state(R_3,Q_3) has two transitions to (R_3,Q_4) and to (R_4,Q_4) but using the same event E_3. This non determinism reflects the presence of an interaction between the composed features (CW and TWC in this case).

5.2. The Essential Behavioral Model (EBM)

Recall that the principle of our method for managing feature interactions consists in three phases: the *global behavior specification*, the *interaction detection* and the *interaction resolution*. Interactions can be presented by states called *conflicting states*. This can be a *deadlock* (blocking) situation, a *non-determinism* or a *constraints violation* that is presented as an incompatibility between two states of features in interaction.

There are two steps that are necessary in order to design a global behavior specification for a system with a set of features:

- **Step 1:** Specify formally each feature (involved in the interaction) with the basic system service (i.e. POTS in the case of a telecommunication system). This specification can possibly be partial.

- **Step 2:** Make a composition of the features, using a suitable composition mechanism (e.g., synchronized product, a cross-product, a sum, …), leading to the global behavior defined by the EBM, which then is subject to further analysis. For instance, if the synchronized product is used, it implies making a synchronized automaton product (as shown in definition 4) of the behaviors of the composed features. Note that the synchronization alphabet could be possibly empty.

5.3. Interaction Detection

Interaction Detection consists in the identification of the conflicting states by analyzing the EBM automaton produced in Step 2 (see Section 5.2 above). Such states could be either a state where a given transition can lead to two distinct states (this is the case of non-determinism which is defined in definition 1), to a deadlock state (where one can execute no transition) or to a state constraints violation (i.e. a state belonging to the product of two features specifications, and that results from two incompatible states). Formally, this violation means that two incompatible states allocate different “logical” values to the same variable.

For example, there is a feature interaction when we compose the two feature specifications Call Waiting (CW) and Three Way Calling (TWC) using a cross-product operation. For instance, when *A* is in communication with *B*

and *A* gets an incoming call from *C*, will the CW feature or the TWC feature be invoked?

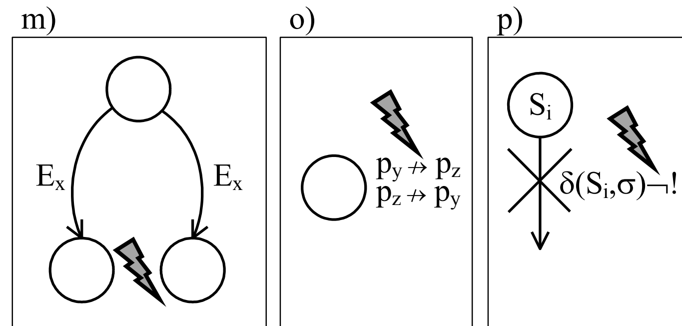


Fig. 7. Three detection aspects checking for composition singularities

When composing the aspects, a set of so-called *detection aspects* check the composition for possible conflicts. A detection aspect detects a singularity using a pointcut and its advice always declares a conflict, which makes the composition fail as long as the singularity is not corrected. Fig. 7 shows three detection aspects that detect the three aforementioned singularities: m) matches any state if there are more than one transition with the same event E_x, o) matches any state with contradicting prepositions p_y and p_z, and p) matches every blocking state S_i for which there is no outgoing transition. When necessary, developers can define their own detection aspects. Whenever one of the detection aspect' pointcuts matches in a composed system, its advice will report a conflict.

Detection aspects are in particular useful when composing many models and aspects that manipulate those models. Detecting composition singularities prevents any further incorrect processing of the system in a potentially undefined state. The above three detection aspects help automatically detecting the most important composition singularities. Therefore, the developer does no longer have to worry about them. Similar to related work on aspects interaction [5], [16], automatic feature interaction detection is enabled. However, automatic feature conflict resolution is not possible in general [5].

5.4. Interaction Resolution

In order to solve feature interactions, a resolution aspect can implement different resolution strategies. For instance, in our previous works we used the following ones:

– **Strategy 1**: Make a composition using an exclusive choice of the two features specifications involved in an interaction. The designer could use

existing merge algorithms [15] for LTS (Labeled Transition Systems) based specifications. Such algorithm produces a specification where its behavior extends the merged ones. The definition of the "extension" relation was given in Definition 5.

- **Strategy 2**: Solve the interaction by making a precedence order upon the occurrence of certain events of the features in interaction. This allows a feature to hide some events from the other feature.

- **Strategy 3**: Establish a protocol between features involved in an interaction. This protocol consists in exchanging the necessary information to avoid the interaction. This approach is more adapted in the case where the features are dedicated to be implemented on distant sites.

However, these strategies are largely based on pre-established and rigid conventions. Therefore, in this paper we propose a more flexible and customizable approach: to use aspects also as an interaction resolution mechanism. According to this proposal, for resolving the conflict, the developer needs to specify a set of resolution aspects. Each aspect intercepts the reception of events, and removes one or more singularities (e.g. cases of non-determinism) from the composed specification. Depending on corresponding context (e.g. the path to the current state and the received events), the aspect can make a choice concerning which of the conflicting features should be active and which not. Therefore, a resolution aspect defines a pointcut and advice for the corresponding conflict resolution, which may have been detected using a detection aspect. Its pointcut matches the conflict situation. Further, its advice declares what states and transitions to remove from the composition such that it becomes deterministic. In the following we explain the suggested method in the case where an interaction occurs between the call waiting (CW) feature and the Three Way Calling (TWC) Feature specified in Section 2, when they are composed using the cross-product operation (see Fig. 6).

First, the detection aspect, in Fig. 7 at index m, identifies this non-determinism singularity. Second, the developer specifies the resolution aspect in Fig. 8. The figure illustrates the pointcut as the thin solid lines that are used as pattern to be recognized on some automaton. It illustrates two pieces of advice as the bold solid lines that indicate what will be added to the automaton. In this case, there are pointcuts that matches the paths E1;CW.E3 and E3;TWC.E3. In both cases, the advice inserts a precedence constraint over the non-deterministic transition labeled with the E3 event, depending from which FSM this transition originates from CW.E3 or TWC.E3. In other words, the resolution aspect resolves the interaction of the CW and TWC features by defining precedence between those features that depends on the sequence of previous events. Intuitively, if a call of C arrives on agent A (event E1) before A presses the flash_back button (event E3), the CW feature will be active. In this case, the left pointcut in Fig. 8 will match and temporarily remove the transition TWC.E3. Conversely, if E3 takes place before

E1, then the TWC feature will be active. In this case, the right pointcut in Fig. 8 will match and temporarily remove the transition CW.E3.

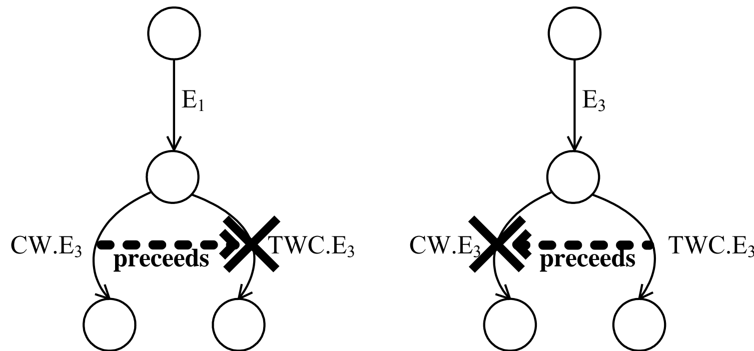


Fig. 8. A resolution aspect that resolves the CW/TWC interaction

In this way, these aspects can be applied to the cross-product in Fig. 6 in order to eliminate the non-determinism from index 1 and 2. In particular, in these indexes the left pointcut in Fig. 8 applies, while the corresponding a vice gives precedence to CW E3 transition with respect to TCW one. Thus, the interaction is resolved in favor of CW.

6. Implementation

This section describes the proof-of-concept implementation of the AO-FSM approach proposed in the previous sections. The proof-of-concept is provided as a domain-specific aspect language AO4FSM. On the one hand, we have implemented composition operators for state machines. On the other hand, we are using the implementation of AO4FSM to detect and to resolve feature interactions.

With this implementation, concrete solutions for feature interactions can be implemented by using aspects whose pointcuts detect conflict situations and advices to handle those situations.

We have implemented a prototype of AO4FSM in the Groovy language [18] using the POPART framework [17] that allows embedding DSLs and developing aspect-oriented extensions for those DSLs in form of plug-ins. Further, we have implemented the examples presented in [7] and which were used as a running example in this paper as a case study.

The implementation of AO4FSM is structured into four parts: the Embedded DSL, the Domain-Specific Join Point Model, the Domain-Specific Pointcut Language, and the Domain-Specific Advice Language. Each part will be elaborated in the following.

6.1. Embedded FSM DSL

The implementation of AO4FSM is based on an embedding of a small FSM DSL for describing finite state machines. The idea of embedding a language is to describe its syntax and semantics using an existing language – the host language – which is in our case Groovy. Basically, a language is implemented as a library, instead of implementing it with a parser and a compiler. Then the DSL programs are evaluated by invoking this library. Note that it is out of the scope of this paper to completely present the embedding of the FSM DSL. We therefore refer the reader who is interested in the general approach to [17].

The following excerpt in Lst. 1 gives a rough idea of how we embed FSMDSL into a Groovy class called `FSMDSL`. For the keyword in FSMDSL's syntax, the class defines methods, such as `fsm`, `state`, `transition`, and `when`.

```
public class FSMDSL {
  private State currentState;
  public StateMachine fsm(Map params, Closure body) {
    StateMachine stateMachine = new StateMachine(…);
    body.delegate = this; body.call();
    return stateMachine;}
  public State state(Map params, Closure stateDefinition){

  public void transitions(Closure transitionDefinitions) {…}
  public void when(Map params) {State from = currentState;
    def t = new Transition(from, params.to, params.event);
    from.addTransition(t);
…
```

Lst. 1. Groovy Code for embedding FSMDSL

Our host language is *Groovy.* Indeed we chose Groovy because it allows embedding DSLs (such as FSM DSL). Furthermore, Groovy is lightweight, dynamic, and provides a higher level of abstraction, but at the same time, you can mix Groovy code with Java. If one needs to extend a DSL with Aspects, like in our FSM DSL, one can do so by exploiting the dynamicity of Groovy provided by its Meta-Object Protocol (MOP) [17]. Despite the additional flexibility provided by using Groovy's MOP, Groovy only enhances Java instead of replacing it. Hence, our implementation runs on every standard JVM.

6.2. Domain-Specific Join Point Model

From a design point of view, Fig. 9 shows the join point types defined for AO4FSM. There are five join point types that represent points in the execu-

tion of a FSM program. The joinpoint *stargingStateMachine* is triggered when a state machine is started. The *EntringState* joinpoint could be reified once a FSM enters into a given state. And when it exits the given state, *ExitingState* joinpoint is then reified. The *ResetStateMachine* joinpoint is reified when moving back to the starting state. As far as concerned, the *EventReceiving* joinpoint, it is reified when the FSM receives an event.

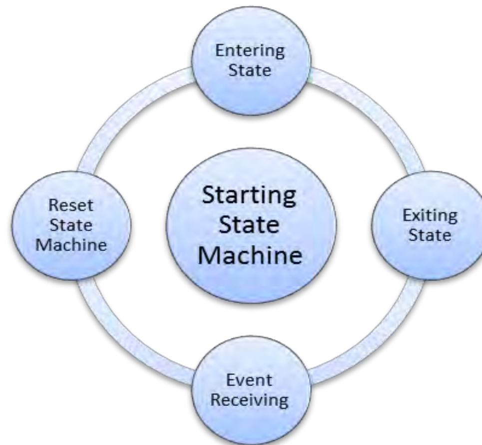


Fig. 9. The Join point types of AO-FSM

```
joinPointContext = new HashMap();
State thisTargetState =
(State)instrumentationContext.receiver;
joinPointContext.thisFSM =  thisTargetState.getOwningFsm();
joinPointContext.thisTargetState = thisTargetState;
def joinPoint = new EntringStateJoinPoint(joinPointContext);
joinPointContext.thisJoinPoint = joinPoint;
```

Lst. 2. Groovy Code for reifying an EntringStateJoinPoint instance

As shown in the Lst. 2, each join point holds a set of parameters that defines its *context*. For instance, the context of the *EntringState* join point refers to the name of the FSM and to the targeted State as parameters. To reify a join point at runtime, the POPART framework will execute this code, which creates an instance of the class `EntringStateJoinPoint`.

6.3. Domain-Specific Pointcut Language

The other important components of our implementation design are the pointcuts. In POPART, each pointcuts is implemented as a class which inherits the `Pointcut` class. In FSMDSL, pointcut sub-classes match the current state parameters with the context of a corresponding join point. It returns “true” if the pointcut matches – and “false” if not. All pointcuts implement a

similar pattern, as shown in Lst. 3 for the `EventReceivingPointcut`. This pointcut only matches `EventReceivingJoinPoint` with an event name given by `expectedEvent`.

In addition to join point types mentioned in the Fig. 9, we added a more history-based pointcut to AO4FSM: the *Stateful* pointcut. This pointcut uses a state machine as a pattern that should match the execution trace of an observed state machine. The pattern state machine is only internally visible for the pointcut, to keep track of the events received by an observed state machine.

```
public class EventReceivingPointcut extends Pointcut {
  String expectedEvent; //given by constructor
    public boolean match(JoinPoint jp) {
    if (jp instanceof EventReceivingJoinPoint) {
      EventReceivingJoinPoint esjp =
        (EventReceivingJoinPoint)jp;
      if (esjp.event.euqals(expectedEvent))
        return true;
      else
        return false;
    }
  }
}
```

Lst. 3. Excerpt of **`EventReceivingPointcut`** which matches reifications of **`EventReceivingJoinPoint`**

One can use the `StatefulPointcut` shown in Lst. 4 to detect the occurrence of patterns in FSMs, such as non-determinism. The first thing that the pointcut will do is that it makes sure that the *EventReceiving* joinpoints is triggered. Then it verifies that the current event matches the event and the name of the current State are the same once in the context of the join points (respectively: `esjp.getEvent()` and `esjp.getCurrentState()`). If the first state matches, we have to check the event that leads us to get out that the state does match as well and so on until we arrive to the final state in our pattern. Then we have to check if the current state in the pattern is final or not, if it is the case then the pointcut matches, and then its corresponding advice is applied.

```
public class StatefulPointcut extends Pointcut {
  StateMachine stateMachinePattern;
  public boolean match(JoinPoint jp) {
    State pCurrentState = stateMachinePattern.
                            getCurrentState();
    String pCurrentStateName = pCurrentState.getName();
    if (jp instanceof EventReceivingJoinPoint) {
      EventReceivingJoinPoint esjp = (…)jp;
      String jpCurrentStateName =
        esjp.getCurrentState().getName();
      if (((jpCurrentStateName == pCurrentStateName)) &&
          (pCurrentState.getEvents().
                          contains(esjp.getEvent()))) {
        stateMachinePattern.receiveEvent(esjp.getEvent());
      } else if((jpCurrentStateName == "*") &&
        pCurrentState.getEvents().contains(esjp.getEvent())){
          stateMachinePattern.receiveEvent(esjp.getEvent());
      }
      if(jpCurrentStateName.equals(
            stateMachinePattern.getFinalState().getName())) {
         return true;
      }
    }
    else
      return false;
  }
}
```

Lst. 4. Excerpt of `StatefulPointcut` to track an execution history

6.4. Domain-Specific Advice Language

The last part of the implementation is the advice language. This language deals with making changes to FSMs to which pointcuts have been matched. When implementing an advice using this language, developers can avoid the problem of non-determinism as mentioned earlier in this paper. For example, to resolve a non-determinism, the advice can remove one of the non-deterministic outgoing edges as suggested by the resolution aspect in Fig. 8.

In the implementation of the advice language, we heavily exploit the Groovy language features. We use closures and aspects that advise the FSMDSL language implementation in order to embed the semantics of the advice languages. One can think of the advice language of AO4FSM as a DSL that produces aspects for Groovy that change the implementation of the embedded FSMDSL. As the FSMDSL is changed by aspects, the evaluation of FSMDSL programs is adapted as well.

To implement the keywords in the advice language, we use *aspect templates*, which are closures whose evaluation returns a Groovy aspect that changes the behavior of some methods in FSMDSL. Such changes will remain dynamic, they do not change the static structure of the state machine,

because there will be no changes conducted in the real FSM instance, but they will be only applied by the aspects. Hence, we can consider those changes as if they were applied to a "copy" of the real FSM. This part of the implementation refers to the patterns described in Fig. 8. Indeed each advice tries to change the behavior of an existing method in the `State` class.

For instance, the `removeTransitionAdvice` is the keyword that represents the advice type at index v) from Fig. 5, whose implementation is shown in Lst. 5. The `removeTransitionAdvice` changes the behavior of a method called `State.handleEvent`, which is responsible for looking up the transition from the `current` to the `next` state. The advice eventually changes the behavior of the `handleEvent` method. If for the `current` state an event is received, that matches the `event` name passed in the advice template's arguments, and if it finds a corresponding transition in the `current` state, then it will do nothing but it will proceed as if the transition does not exist. Otherwise, the advice calls `proceed`, which will execute the `handleEvent` as normal, i.e. without the change.

```
removeTransitionAdvice = { current, next, event ->
  aspect(name:"generatedName_" +
         "removeTransitionAdvice\$instante\$" +
         current+"\$"+next+"\$"+event, perInstance:current) {
    around(method_execution("handleEvent")){
      if(matchesRemovedTransition(current,next,event)){
        //omitting proceed ignores the transition
      else {
        return proceed() //proceeds the transition }
…
```

Lst. 5. Implementation of the `AddTransitionAdvice`

7. Discussion

To validate the approach, we use the AO4FSM prototype to automatically detect the interactions, and we have developed a resolution aspect to revolve these interactions. We could achieve the objectives stated in our introduction, namely the support of the separation of concerns (in particular crosscutting features), the formalization of the behavior, and how to deal with interactions. With the current prototype, conflicts can be successfully detected and resolved. However, correct results depend on whether the developer completely specifies the model and correctly implements aspects with the AO-FSM tool.

In the remainder of this section, we discuss the details about generality and limitations of our approach with respect to our model (Section 7.1) and the current prototype implementation (7.2).

7.1. Discussion about the Model

When using aspects for feature interaction detection and resolution, the developer has to decide case by case what are all encountered interaction singularities in a composed global behavior, which may have combinations of the basic singularities shown in Fig. 7. For example, there can be a non-deterministic transition with the same label (instance of index m) to a state which has two contradicting prepositions (instance of index o) within a composition. Another example is the case where there are more than one blocking states (instance of index p). The compositionality can lead to complex scenarios of interactions. Still, the detections aspects will seek all instances of the three kinds of interactions and combinations thereof.

Regarding the totality of feature interaction detection, also detecting other kinds of singularities is conceivable. Possible singularities can be derived from properties observed by the theory of finite state machine and graph theory. There are properties like connectivity and cycles. Some of these properties could indicate a singularity of a possible interaction. Other properties are not relevant for interactions in our model. For example, with respect to the connectivity property from graph theory, all FSMs in an EBM are by definition a *connected graph*, i.e., there is a path from any node to any other node in the graph. It is trivial to conclude that all synchronized cross-product are *connected*. Therefore, this property does not need to be observed in our model.

An example of a property which could be relevant is the presence of cycles. A cycle in a directed graph is a path from one node back to the same node, which in our EBMs could indicate whether the execution of a feature's behavior will not terminate. For instance, if there is no cycle, the behavior will *terminate* (i.e. there are only finite sequences of input that are recognized by the FSMs), or it stays alive (*liveness* property, i.e., something good can happen). It can be interesting in certain domains, such as *security* or *safety*, whether an EBM has such a property or not. If there are two EBMs that are free of cycles, still there may be composition that has cycles, i.e., the composed behavior may not terminate or a liveness property is violated, which indicates a feature interaction.

The current three detection aspects do not consider these more complicated scenarios. They are out of scope of the paper because for the time being we found no interesting situation in the software product-line scenarios we focus on. Still, at the current stage, we cannot draw universally valid conclusions from the case study. A larger case would be more convincing. At the end, only a formalization proof of the formalism in a proof assistant (like Isabelle or Coq) would give absolute guarantees.

With respect generality, the detection capabilities of our three detection aspects are limited. But, if it is needed, researchers and developers can define new kind of detection aspects. Since our pointcut language is based on finite state machines and from automata theory, we can derive that interaction detection for all properties of the composed automaton in an EBM (the global behavior) that can be recognized by another finite state machine (the

pointcut). The class of properties that can be recognized by finite state machines is well known: it can recognize properties expressed by regular expressions over the input alphabet, Linear-Time Temporal Logic (LTL), and similar classes. With respect to generality, it can be considered as an advantage that the approach is based on formal methods of Automata Theory, which allows us to draw such conclusions from a large body of theoretic research.

The resolution capabilities in our model are complete with respect to finite state machine, because developers can add and remove all elements in an FSM, which are accessible as first-class objects in our aspect language. Developer can insert and delete both states and transitions. Developer can manipulate transitions by changing the incoming and outgoing state. We did not impose any restrictions with respect to the model into the aspect language about what could be manipulated in AO4FSM advice.

7.2. Discussion about the Implementation

Our prototype implementation only covers feature detection and resolution at design time. For save feature implementation, our approach could easily be integrated with a code generator from state machines to C or Java code.

Various practicable limitations need to be addressed by future work; the expressiveness of the model is confined by state machines and therefore systems whose behavior can be formalized as a regular language. The approach could be extended for models with richer semantics, which consequently would make it more complicated. Because we build the synchronized product of FSMs, the approach suffers from the well-known state explosion problem when using FSMs for modeling. Therefore, the prototype can only be used to analyze small models. In future work, we want to reduce synchronized products by finding equivalent states. Another limitation is that it currently does not nicely integrate with standard modeling notations, such as UML. In future work, we would like to support for importing UML state charts and let the developer enhance them to EBMs.

8. Related Work

Our work is related to the works in the field of FOP, AO modeling, and model driven development.

FOP [11] provides language support for implementing modular features that encapsulate basic functionality. Similar to FOP, our EBM and AO-FSM allow modular specification of features. While FOP uses so called *lifters* for inheriting features into a composition, we build on the sum for inheriting FSMs and the synchronized product for composing them. While FOP is an approach at the implementation level, we focus on the specification of features. FOP allows defining known interactions. In contrast, EBM and AO-

FSM allow automatic detecting of interactions that the developer is not aware of.

Aspect-oriented modeling has come up with various modeling notations into which aspects are woven. There are AO state machines [13] and other AO models available. However, they have been little explored in the context of detecting feature interactions in behavioral models. AO models can only detect conflicts involving aspects, but they cannot detect interactions between base features as we do.

An ongoing trend in language research is to extend more and more base languages with aspects, like we did for our state machine base language (FsmDSL). Recently, aspects for Petri nets have been proposed [20].

There are also a number of other extensible compilers and language workbenches that can be used for extending existing base languages with aspects, namely the Aspect SandBox [25], Reflex [22], JAsCo [21], the AspectBench Compiler [23], JAMI [24], and AspectASF [26]. These extensible language infrastructures mostly support only extensions to general-purpose language, but not to DSLs.

Achieving better modularization of language implementations in language engineering is a central subject of research in recent years. There are parser generators, such as ANTLR [28] and compiler-compilers, such as SableCC [29] that enable modular and extensible language implementations. In their specification languages, they use language constructs, such as inheritance, that enable better modularity in the language's specifications and their implementations. Aspect-oriented modularization itself has been proposed to be used in language engineering to improve modularity in language specifications [26], [30] and implementations [23]. Such special specification language constructs can be used to implement aspect-oriented language extensions in a modular way [27].

All above mentioned language implementation approaches use an external tool e.g. compiler that generates from the specification language (meta-language) the executable code in a particular target language. In contract to these approaches, we embed the DSL and aspects as internal DSLs. There is no external tool because DSL programs are processed with the same compiler as host programs. This fact allows us to use the extension constructs of the host language (inheritance and meta-object protocol) to extend the state machine DSL with aspect-oriented syntax and semantics using a modular AO language implementation.

Model-driven development proposes various kinds of models – not only FSMs. Life-Sequence Charts [19] are similar to AO-FSM. Such models are often used for code generation. While standard model notations do not adequately consider interactions, there are a few special models that allow expressing such constraints for a restricted set of domains, such as telecommunications for which special DSLs are available. Currently, developers are left alone to encode constraints on the modeled feature using constraint languages for which often there is no complete support for code generation. In contrast to this, possible domains for EBM and AO-FSM are not limited.

9. Conclusion

In this paper, we suggested a formal approach to detect and resolve feature interactions within a distributed software system. The approach is based on a new formalism for aspect-oriented state machines (AO-FSM) and language implementation AO4FSM based on finite-state machines and Essential Behavioral Models (EBM). An EBM defines states and transitions as a FSM, but states and transitions do not need to be completely specified and therefore it allows defining partial FSM models.

A specific mechanism for interactions detection and a strategy for feature interaction resolution were presented. The implementation of this mechanism and its associated strategy were made using the AO-FSM formalism. Therefore, the pointcut defines a state and transition pattern that selects all FSMs that the advice adapts, while the advice defines a state and transition pattern that it applies at the selected points. In fact, the approach uses aspect-oriented state machines to intercept, prevent, and manipulate events that cause conflicts.

Acknowledgements. This work was partially supported by the EMERGENT project (01IC10S01N), Federal Ministry of Education and Research (BMBF), Germany. A support was also provided by the DAAD (German Academic Exchange Service) and the Moroccan CSPT Research Program.

The authors would like to thank Professor Mira Mezini and the Software Technology Groups at the Technische Universität Darmstadt. Two of the co-authors have worked as part of her group while conducting this research.

The authors would like to thank the anonymous reviewers for their valuable comments and recommendations.

The authors would also like to thank Yassine Essadraoui who has contributed to the implementation of the prototype of AO4FSM and the telephone case study as part of his Master's thesis.

References

1. Clements, P. and Northrop, L., "Software product lines", Addison-Wesley, 2001.
2. Pohl, K. and Böckle, G. and Van Der Linden, F., "Software product line engineering: foundations, principles, and techniques", Springer-Verlag New York Inc, 2005.
3. K. Czarnecki and A. Wasowski. "Feature diagrams and logics: There and back again" in Proc. 11th Int. Software Product Line Conference (SPLC 2007), Washington, DC, USA, 2007, pp. 23–34.
4. Kiczales, G. and Lamping, J. and Mendhekar, A. and Maeda, C. and Lopes, C. and Loingtier, J.M. and Irwin, J.: "Aspect-oriented programming" in Proc. Europ. Conf. on Object-Oriented Programming, Springer, 1997, pp. 220–242.
5. G. Kniesel, "Detection and Resolution of Weaving Interactions. TAOSD: Dependencies and Interactions with Aspects", In Transactions on Aspect-Oriented

Software Development V, pp. 135–186, LNCS, vol. 5490, Springer Berlin / Heidelberg, 2009.
6. Tanter, E., “Aspects of composition in the Reflex AOP kernel”, Software Composition, Springer, 2006, pp. 98–113.
7. M. Erradi and A. Khoumsi, “Une approche pour le traitement des interactions de fonctionalités des systèmes téléphoniques”, in Proc. Colloque Francophone International sur l'Ingénierie des Protocoles (CFIP'95), Rennes, France, 1995.
8. M. Mernik, J. Heering, and A.M. Sloane, “When and how to develop Domain-Specific Languages” ACM Computing Surveys (CSUR), vol. 37, no. 4, 2005, pp. 316–344.
9. Pamela Zave, “Feature Interaction”, http://www2.research.att.com/~pamela/fi.html
10. E.J. Cameron, N.D. Griffeth, Y.-J. Lin, M. Nilson, W.K. Schnure, et H. Vlethuijsen. “A feature Interaction Benchmark for IN and beyond”, Feature Interactions in Telecommunications Systems, Eds. L.G. Bouma and H. Velthuijsen, IOS Press, Amsterdam, 1994.
11. Prehofer, C.: “Feature-oriented programming: A fresh look at objects” in Proc. ECOOP, Springer, 1997, pp.419–443.
12. Parnas, D.L., “On the criteria to be used in decomposing systems into modules”, *Communications of the ACM*, vol. 15, no. 12, 1972, pp. 1053–1058.
13. M. Mahoney, T. Elrad, “A Pattern Story for Aspect-Oriented State Machines”, LNCS, Vol. 5770, 2009.
14. G. v. Bochmann, “Finite State Description of Communication Protocols”, Computer Networks, Vol. 2 (1978), pp. 361-372.
15. F. Khendek and G. v. Bochmann, “Merging Behavior specifications”, Proc. FORTE'1993, Boston, USA.
16. W. Havinga, I. Nagy, L. Bergmans, M. Aksit, "A graph-based approach to modeling and detecting composition conflicts related to introductions". In Proc. International Conference on Aspect-Oriented Software Development, ACM, 2007.
17. T. Dinkelaker, M. Eichberg, and M. Mezini, „An Architecture for Composing Embedded Domain-Specific Languages”. In Proc. Aspect-Oriented Software Development ACM New York, 2010.
18. D. König, A. Glover, “Groovy in Action”. Manning, 2007.
19. W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design, vol. 19, no. 1, pp. 45–80, 2001.
20. T. Molderez, B. Meyers, D. Janssens and H. Vangheluwe, “Towards an Aspect-oriented Language Module: Aspects for Petri Nets”, In Proc. Workshop on Domain-specific Aspect Languages, ACM New York, 2012.
21. D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: “An Aspect-Oriented Approach tailored for Component-based Software Development.” In AOSD, pages 21-29, 2003.
22. E. Tanter. From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming. PhD thesis, Université de Nantes, France, 2004.
23. P. Avgustinov, J. Tibble, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, and G. Sittampalam, “abc: An extensible AspectJ Compiler.” In AOSD, pages 87-98, 2005.
24. W. Havinga, L. Bergmans, and M. Aksit, “Prototyping and Composing Aspect Languages using an Aspect Interpreter Framework.” In ECOOP, pages 180-206, 2008.

25. H. Masuhara, G. Kiczales, and C. Dutchyn, “A Compilation and Optimization Model for Aspect-Oriented Programs.” In CC 2003, volume 2622 of LNCS, pages 46-60, 2003.
26. P. Klint, T. van der Storm, and J. Vinju, “Term Rewriting Meets Aspect Oriented Programming.” In Proc. of Processes, Terms and Cycles: Steps on the Road to Infinity, Springer, LNCS 3838, 2005.
27. E. Van Wyk, “Aspects as modular language extensions.” In Electronic Notes in Theoretical Computer Science, volume 82(3), pages 555-574, Elsevier, 2003.
28. T. Parr, “The definitive ANTLR reference: building domain-specific languages.” The Pragmatic Bookshelf, 2007.
29. E.M. Gagnon and L.J. Hendren, “SableCC, an object-oriented compiler framework.” In Proc. Of Technology of Object-Oriented Languages, pages 140-154, IEEE, 1998.
30. M. Mernik, X. Wu, and B. Bryant, “Object-oriented language specifications: Current status and future trends.” In ECOOP Workshop: Evolution and Reuse of Language Specifications for DSLs (ERLS), 2004.

Tom Dinkelaker holds a PhD and a German Diploma in computer science from the Technische Universitaet Darmstadt, Germany. His research focuses on the implementation of embedded domain-specific languages and aspect-oriented programming languages. To provide support for customizing language semantics and implementation strategies, in his thesis, he explored the potentials of using meta-object protocols to enable open language semantics. Tom has embedded a set of languages that are language product-lines, i.e., their syntax and semantics can be extended by language developers or end users in order to customize them for special domains. Tom is now working at Ericsson R&D in the Customer Care team. At Ericsson, he develops the next generation of business support systems that delivers features on top of a flexible architecture that customers adapt for individual needs.

Mohammed Erradi has been a professor in Computer Science since 1986. He has been leading the distributed computing and networking research group since 1994 at ENSIAS (Ecole Nationale d’Informatique et d’Analyse des Systèmes) of Mohammed V-Souissi University (Rabat Morocco), and was head and founding member of the Alkhawarizmi Computing Research laboratory. Before joining ENSIAS, Professor Erradi has been affiliated with the University of Sherbrooke and the University of Quebec in Canada. His recent main research interests include Communication Software Engineering, Distributed Collaborative Applications, and Reflection and Meta-level Architectures. He obtained his Ph.D. in 1993 at University of Montreal in the area of Communicating Software Engineering under the supervision Professor Gregor Von Bochmann. He is currently the Principal Investigator of a number of research projects grants. Among the topics of these projects we find: Collaborative environment for Telediagnosis in NeuroScience, Cloud Computing Security, Security Policies composition, Adaptive Wireless Sensor Networks, Vertical Handover in Mobile Networks. Professor Erradi

has published more than 60 papers in international conferences and journals. He has organized and chaired five international scientific events and has been a member of the program committee in multiple international conferences.

Meryeme Ayache is a young security researcher, she graduated in 2012 from ENSIAS (Ecole Nationale Supérieur d'Informatique et d'Analyse des Systèmes, Rabat, Morocco) specializing in Security of Information Systems. She participated in the implementation of a project on "Behavioral Modeling with Aspect-Oriented State Machines" as an internship within the Software Technology Group of the Technical University of Darmstadt. Her interest is on mobile computing and security.

Received: December 16, 2011; Accepted: July 18, 2012.

DOI:10.2298/CSIS120203034C

A MOF based Meta-Model and a Concrete DSL Syntax of IIS*Case PIM Concepts

Milan Čeliković, Ivan Luković, Slavica Aleksić, and Vladimir Ivančević

University of Novi Sad, Faculty of Technical Sciences,
Trg D. Obradovića 6, 21000 Novi Sad, Serbia
{milancel, ivan, slavica, dragoman}@uns.ac.rs

Abstract. In this paper, we present a platform independent model (PIM) of IIS*Case tool for information system (IS) design. IIS*Case is a model driven software tool that provides generation of executable application prototypes. The concepts are described by Meta Object Facility (MOF) specification, one of the commonly used approaches for describing meta-models. One of the main reasons for having IIS*Case PIM concepts specified through the meta-model, is to provide software documentation in a formal way, as well as a domain analysis purposed at creation a domain specific language to support IS design. Using the PIM meta-model, we can generate test cases that may assist in software tool verification**.** The meta-model may be also a good base for the process of the concrete syntax generation for some domain specific language.

Keywords: information system modeling, domain specific languages, domain specific modelling, platform independent models.

1. Introduction

IIS*Case is a software tool that provides a model driven approach to information system (IS) design. It supports conceptual modeling of database schemas and business applications. IIS*Case, as a software tool assisting in IS design and generating executable application prototypes, currently provides:

- Conceptual modeling of database schemas, transaction programs, and business applications of an IS;
- Automated design of relational database subschemas in the 3rd normal form (3NF);
- Automated integration of subschemas into a unified database schema in the 3NF;
- Automated generation of SQL/DDL code for various database management systems (DBMSs);
- Conceptual design of common user-interface (UI) models; and
- Automated generation of executable prototypes of business applications.

In order to provide design of various platform independent models (PIM) by IIS*Case, we created a number of modeling, meta-level concepts and formal rules that are used in the design process. Besides, we have also developed and embedded into IIS*Case visual and repository based tools that apply such concepts and rules. They assist designers in creating formally valid models and their storing as repository definitions in a guided way. Main features of IIS*Case and the specification of its usage may be found in [1].

There is a strong need to have PIM concepts specified formally in a platform independent way, i.e. to be fully independent of repository based specifications that typically may include some implementation details. Our current research is based on two related approaches to formally describe IIS*Case PIM Concepts. One of them is based on Meta Object Facility (MOF) and the other one on a textual Domain Specific Language (DSL). In [2], we give a specification of the IIS*Case textual modeling language, named IIS*CDesLang that formalizes IIS*Case PIM concepts and provides modeling in a formal way. IIS*CDesLang meta-model is developed under a visual programming environment for attribute grammar specifications named VisualLISA [3].

In [4] we propose a meta-model of IIS*Case PIM concepts, which is based on the Meta Object Facility (MOF) 2.0. MOF 2.0 is a common meta-meta-model proposed by Object Management Group (OMG) where meta-models are created by the use of UML class diagrams and Object Constraint Language (OCL) [5]. As we could not find standardized implementation of MOF, we decided to use Ecore meta-meta-model. Ecore is the Eclipse implementation of MOF 2.0 in Java programming language which is provided by Eclipse Modeling Framework (EMF) [6]. Ecore concepts are not always identical to MOF 2.0 concepts, but they are expressive enough to create our IIS*Case meta-model. A benefit of such a meta-model is providing software documentation in a formal way. Besides, created meta-model can be used for the software tool verification in EMF environment. It also represents a domain analysis specification necessary to create IIS*CDesLang, as a textual DSL that supports IS design. In this paper we give an example that illustrates the process of modeling a part of an IS using IIS*Case PIM concepts. We also present a small part of a concrete syntax grammar that is based on the definition of IIS*Case PIM concepts.

In Fig. 1 we illustrate the four layered architecture of our solution, which is tailored from OMG four–layered architecture standard. Level M3 comprises meta-meta-model (MOF 2.0) [7] that is used for implementation of the IIS*Case meta–model (M2). M2 level represents the IIS*Case PIM meta-model specified by MOF specification and implemented in EMF. Using the IIS*Case PIM meta-model, a designer specify and implement a conceptual model of an IS that is placed at the M1 level of the four-layered data architecture from Fig. 1. By using IS applications generated by IIS*Case, end-users manipulate real data, i.e. they create and use models of entities from real world (M0), using the conceptual model (M1).

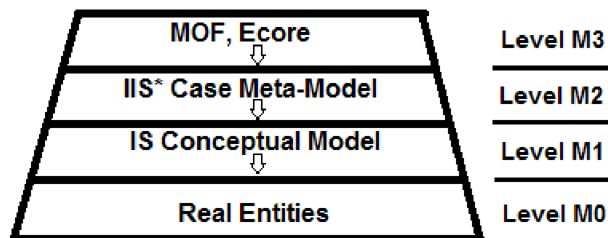


Fig. 1. Four layered meta-data architecture

Apart from Introduction and Conclusion, the paper is organized in four sections. In Section 2, we present a related work. In Section 3 we give a presentation of IIS*Case PIM concepts specified through the meta-model that is implemented in EMF environment. In Section 4 we illustrate an example of IIS*Case PIM concepts usage, while in Section 5 we give a concrete syntax definition of main PIM concepts.

2. Related Work

Nowadays, meta-modeling is widely spread area of research and there is a huge number of references covering MOF based meta-models. However, we could not find papers presenting formal approaches to the specification of meta-model implementation and design of CASE tools, based on MOF or Ecore meta-meta-models.

We found a vast number of meta-model specifications and implementations based on MOF or Ecore specifications. Meta-models based on MOF are also presented in [8] and [9]. The authors in both papers propose the meta-models of the Web Modeling Language. The meta-model specification and design is implemented under EMF environment. Defining W2000 [8] as a MOF meta-model, the authors specify it as an UML profile. In [9], the authors provide a solution for the generation of MOF meta-models from document type definition (DTD) specifications [10]. A formal specification of OCL is given in [11]. In their meta-model, the authors precisely define the syntax of OCL, as it is given in [5]. They propose a solution for the presented meta-model integration with the UML meta-model. In [12], the authors propose the Kernel MetaMetaModel (KM3) representing a DSL for meta-model definition. In [13], the authors propose the UML Profile, EUIS, used for the specification of business applications' user interfaces. Their solution provides automatic interface code generation that is based on their own HCI standard. They developed a DSL specified as UML Profile that offers user interface modeling and generation. In [14] the authors propose a solution for the kiosk applications development. They present KAG, a DSL that provides kiosk applications development in a more rapid way than standard high-level programming languages. While the presented DSL provides rapid application

prototyping of new applications, it also simplifies the maintenance process of existing applications. The DSL has also reduced the number of errors that were common in the process of programming using standard high level programming languages. The authors of the paper [15] present the DOMMLite, a DSL that provides the definition of database applications' static structures. The language structure has been defined at the level of the meta-model. The textual syntax has been defined in order to provide creation, update and persistence of DOMMLite models. They have also developed a textual Eclipse editor that provides generation of source code for graphical-user interface forms supporting CRUDS (Create-Read-Update-Delete-Search) operations. In [16], the authors present a selection of 75 key publications, covering the area of DSLs. They give an overview of the terminology, DSL examples, design methodologies, and implementation techniques. In [17], the authors give an overview of the problems in the decision, analysis, design, implementation and deployment phases of DSL development. They have identified patterns for the first four phases that can aid DSL developers. They have also presented language development systems and frameworks aimed at facilitating the development process. The authors of the paper [18] present Sequencer, a domain specific modeling language for programming or modeling measurement procedures without interacting with programming engineers. Sequencer provides development of measurement procedures inside the measurement system DEWESoft using DCOM objects. It is a DSL that provides textual or visual mode, customized for the application development in the measurement domain. Similar to the papers discussed in this section, we base our research on the development of the DSL in the domain of IS development. In this paper we focus on the meta-model specification of IIS*Case PIM concepts and the generation of concrete syntax.

There are various meta-modeling tools that are generally based on their own meta-meta-model specifications. One of them is Generic Modeling Environment (GME) [19], a configurable toolkit for domain specific modeling and program synthesis based on UML meta-models. MetaEdit+ [20], [21] and [22] allows creation and design of meta-models by the use of a graphical editor providing the Graph-Object-Property-Port-Role-Relationship data model. All of these tools may also be used for the IIS*Case PIM meta-model description in a formal way.

3. IIS*Case Meta-Model

IIS*Case provides a definition of several concepts embedded into IIS*Case repository, that typically may include some implementation details. In this paper, we present only IIS*Case PIM meta-model concepts specified by Ecore meta-meta-model. Hereby we overview here the following main IIS*Case PIM concepts: *Project, Domain* and *Attribute* as *Fundamental concepts*, *Program unit, Application system*, *Application type, Form type* and *Component type*. A model of the IIS*Case main concepts with their properties

and relationships is presented latter on, in Fig. 2. More information about these concepts may be found in [1] and [23], as well as in many other authors' references.

3.1. Project

In IIS*Case, modeling process is organized through one or more projects. Therefore, the central concept in our meta-model from Fig. 2 is *Project*. For each project, a designer defines the project name as its mandatory property. All existing elements in the repository of IIS*Case are always created in the context of a project. *Fundamental concepts* and *Application systems* are subunits of a *Project*. *Fundamental concepts* are formally independent of any application system. *Fundamental concept* instances can be used in more than one application system, because they are defined at the level of a project. *Fundamental concepts* comprise zero or more:

- *Attributes*,
- *Domains*,
- *Program units* and
- *Inclusion dependencies*.

Each project is organized through application systems and fundamental concepts. For each project, we can define zero, or more instances of the *Application system* concept. An IS designer may create application systems of various types. By the *Application type* concept, a designer may introduce various application system types and then associate each instance of an application system to exactly one application type.

At the level of a project, IIS*Case provides generation of various types of repository reports. As the Report is not a real modeling concept, it does not belong the IIS*Case PIM concepts. However, the IIS*Case repository contains Report concept. It is used by the IIS*Case reporting tools.

3.2. Domain

Domains specify allowed values of database attributes. They are classified as:

- Primitive and
- User defined.

Therefore, in our meta-model, there are two classes: *PrimitiveDomain* and *UserDefinedDomain* that are subclasses of a *Domain* class.

Primitive domains represent primitive data types that exist in formal languages, such as string, integer, char, etc. The reason behind the existence of user defined domain concept is to allow designers to create their own data types in order to raise the expressivity of their models. Each domain has its name, description and default value. At the level of a primitive domain, a designer may specify *length required* item value. It denotes if a numeric length: must be, may be, or is not to be given. For user defined domains, a

designer needs to define a domain type and a check condition. IIS*Case supports two classes of user defined domains:

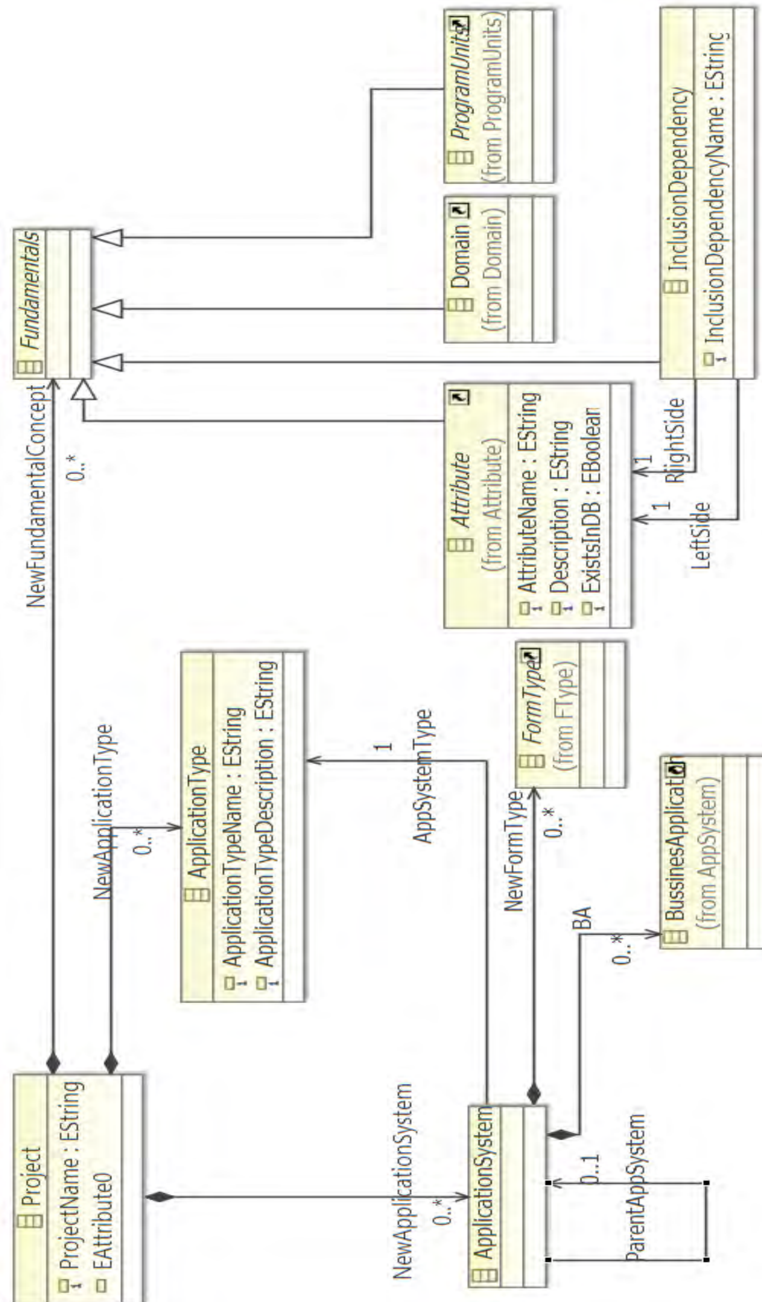


Fig. 2. A meta-model of IIS*Case main PIM concepts

- Domains created by the inheritance rule and
- Complex domains.

A domain created by the inheritance rule references a specification of some primitive or user defined domain. We call it a child domain, while the referenced domain is also called a superordinated or parent domain. By using the inheritance, all the rules defined at the level of a parent domain also hold for the child domain.

Complex domains may be created by the tuple rule, set rule, or choice rule. A domain created by the tuple rule we simply call the tuple domain, because it represents a tuple of values. The items of such a tuple structure are some of already created attributes. A domain created by the choice rule we call a choice domain. It is specified in almost the same way as a tuple domain. The choice domain concept is the same as the choice type of XML Schema Language. Each value of a choice domain corresponds to exactly one attribute. A set domain represents sets of allowed values over a specified domain.

Check condition is a regular expression that additionally constrains possible values of a domain created by a designer.

Domain concept allows definition of display properties of screen items that correspond to attributes and their domains. Each domain corresponds to exactly one element of the *Display* type. The *Display* concept specifies rules, later used by the application generator to generate screen or report items. Generated screen or report items correspond to some of the attributes, and attributes correspond to some of domains. Technical aspects of the display properties implementation may be found in [24] and [25].

3.3. Attribute

In Fig. 3, we present a meta-model of the *Attribute* concept. Each attribute in a project is identified by its name. It also has a description and a Boolean property specifying if it belongs to the database schema. In practice, the most of created attributes belong to the database schema. For attributes representing derived (calculated) values in reports or screen forms a designer may decide if they are to be included in the database schema. By this, we classify attributes as: a) included or b) non-included in a database schema.

According to the way how an attribute gains a value, we classify attributes as: a) non-derived or b) derived. A value of a non-derived attribute is created by an end-user. A value of derived attribute is always calculated from the values of other attributes, by applying some function, i.e. a calculation formula. There is a rule that any non-included attribute must be specified as derived one.

A function that is used to calculate a derived attribute value is formally specified in the IIS*Case repository. Additionally, a designer may specify parameters that are passed to the function. The *Function* concept will be presented in the next subsection, Program Units. If an attribute is non-included in a database schema, the function is referenced as a query function.

Only derived attributes that are included in a database schema may additionally reference three IIS*Case repository functions specifying how to calculate the attribute values on the following database operations: insert, update and delete.

An attribute may be specified as a) elementary or b) renamed. A renamed attribute references a previously defined attribute. The source of such an attribute is the referenced attribute, but with the different semantics. The renamed attribute needs to be included in database schema. Renaming is a concept that also exists in the Entity-Relationship and relational data models. By means of renaming, a designer may differentiate between semantics of "similar" attributes. If a designer introduces a new attribute A1 and specifies it as a renamed from the existing attribute A, he or she actually specifies an inclusion dependency of the form $[A1] \subseteq [A]$ at the level of a universal relation scheme. More information about the use of renaming concept in the context of IIS*Case tool may be found in [1]. Inclusion dependency is modeled in Fig. 2 in our meta-model as the class *InclusionDependency* inheriting *Fundamentals.* It is also related with class Attribute over two relationships, that actually represent left and right side of the inclusion dependency.

To each attribute a domain must be associated. This association allows defining a default value and a check condition. If the attribute value is not specified, the default value is assigned to it. Check condition is the attribute check expression that represents the regular expression that additionally constrains the value of the attribute.

At the level of an attribute, we can specify the display properties. The concept of the *Display* properties is same as the one at the level of the *Domain* concept. Values of display properties, specified at the level of the associated domain, may be inherited or overridden according to the requirements of an IS project.

3.4. Program Units

The *Program unit* concept is used to express complex application functionalities. We classify program units as: a) *Functions*, b) *Packages* and c) *Events*.

The *Function* concept is used to specify any complex functionality that later may be used in other specifications. Each function has its name and return type that are mandatory properties, as well as a formal specification of a function body and a description that are optional. The return type is a reference to a domain. A function specification may include a list of formal parameters. Each formal parameter of a function is specified by its name and a sequence number, as mandatory properties. Exactly one domain is associated with each formal parameter. Any parameter may also have a default value specified. With respect to the ways of exchanging values between the function and its calling environment, we classify formal parameters as: a) In, b) Out and c) In-Out, with a usual meaning as it is in many general purpose programming languages.

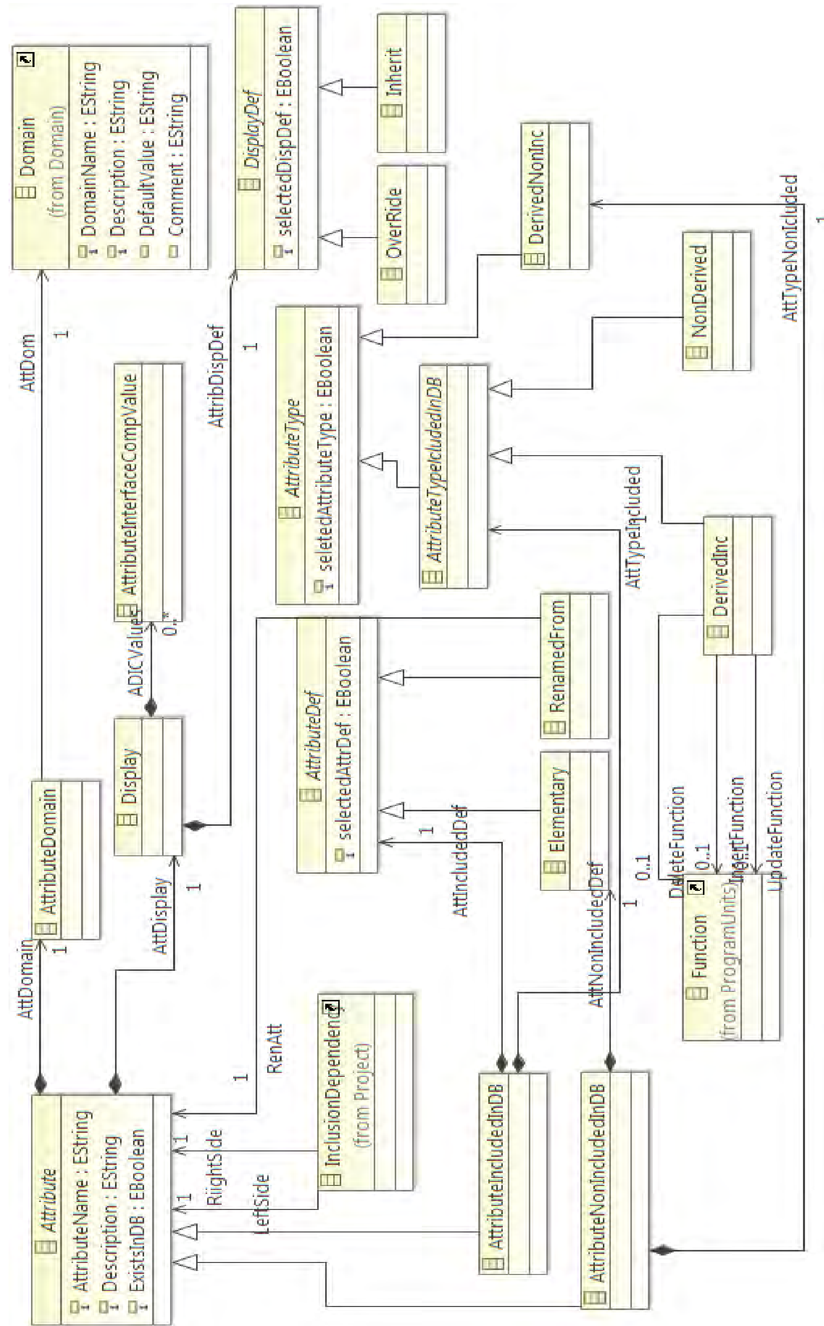


Fig. 3. A meta-model of the IIS*Case Attribute concept

IIS*Case provides grouping of created functions into packages. Each function may be included into one or more packages, or may stay as a stand-alone object. By the location of the deployment in a multi-layer architecture, the packages are classified as: a) Database server packages, b) Application server packages and c) Client packages. A package is identified by its name, and may have an optional description.

The *Package* concept is modeled by the inheritance rule. We have the abstract class named *Package*. It is superordinated to the classes: *DBServerPackage*, *ApplicationServerPackage* and *ClientPackage*. For each instance of the *Package* class, there may be zero or more references to the instances of the *Function* class.

The *Event* concept is used to represent any software event that may trigger some action under a specified condition. Each event is identified by its name, and may have an optional description. Similar to the packages, by the location of the deployment in a multi-layer architecture, we also classify events as: a) Database server events, b) Application server events and c) Client events. The *Event* concept is modeled in the similar way like *Package*, by applying the inheritance rule.

3.5. Application System

The *Application System* concept is used to model organizational parts of each Project. Each application system has its name and a description as mandatory properties. Besides, it may reference other, subordinated application systems that we call child application systems. By this, a designer may create a hierarchy of application systems in a project. Application system hierarchy is modeled by a recursive reference.

Various kinds of IIS*Case repository objects may be created at the level of an application system, but in this paper we focus on two of them only, as PIM concepts: a) *Form type* and b) *Business Application*.

3.6. Form type

Form type is the main concept in IIS*Case. The meta-model of this concept is presented in Fig. 4. It abstracts document types, screen forms, or reports that end-users of an information system may use in a daily job. By means of the *Form type* concept, designers indirectly specify at the level of PIMs a model of a database schema with attributes and constraints included. At the same time, they also specify a model of transaction programs and applications of an information system.

Apart from creating form types in application systems, designers may include into their application systems form types created in other application systems being modeled. Therefore, we classify form types as: a) owned and b) referenced. A form type is owned if it is created in an application system. It

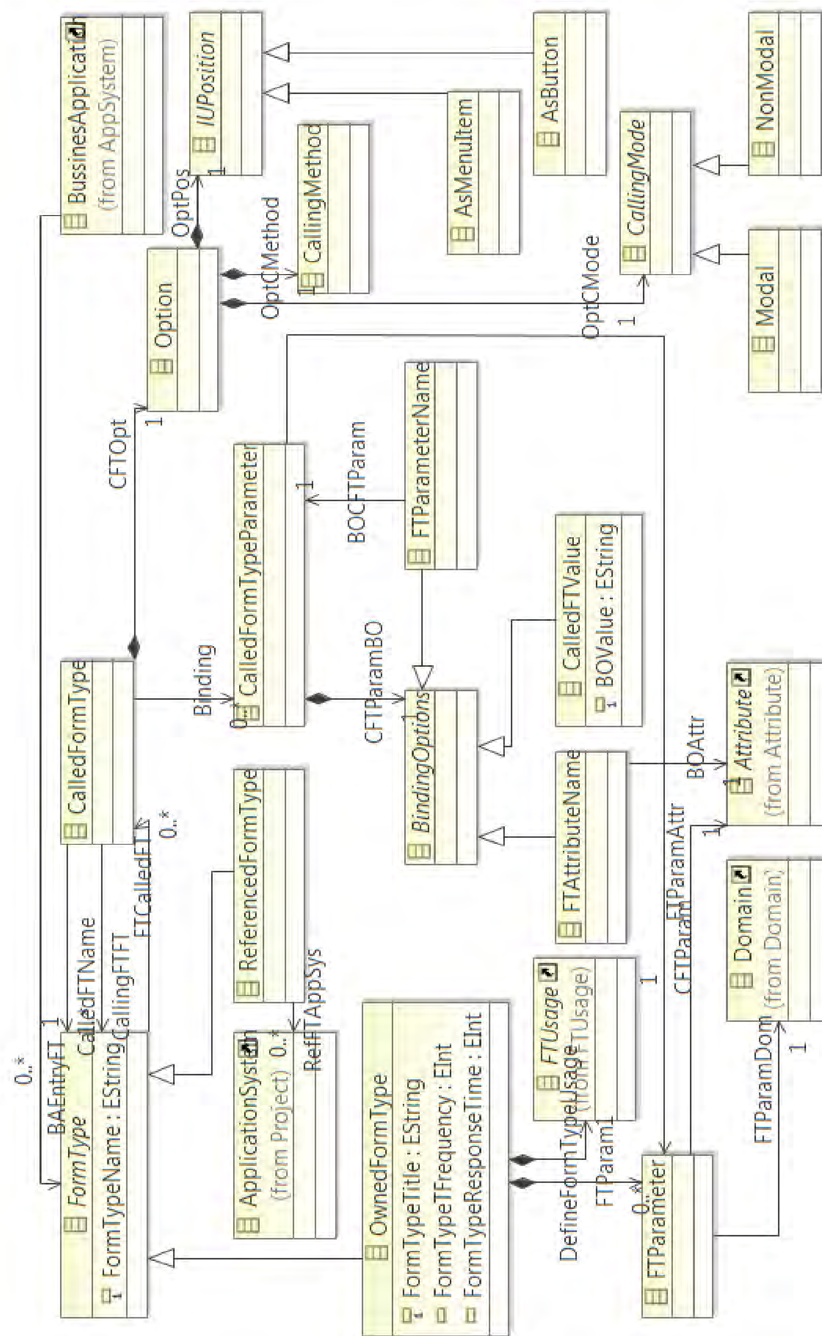


Fig. 4. A meta-model of the IIS*Case Form Type concept

may be modified later on through the same application system without any restrictions. A referenced form type is created in another application system and then included into the application system being considered. All the referenced form types in an application system are read-only.

Each form type has a name that identifies it in the scope of a project, a title, frequency of usage, response time and usage type. Frequency is an optional property that represents the number of executions of a corresponding transaction program per time unit. Response time is also an optional property specifying expected response time of a program execution. By the usage type property, we classify form types as: a) menus and b) programs.

Menu form types are used to model menus without data items. Program form types model transaction programs providing data operations over a database. They may represent either screen forms for data retrievals and updates, or just reports for data retrievals. As a rule, a user interface of such programs is rather complex. A program form type may be designated as *considered in database schema design* or *not considered in database schema design*. Form types considered in database schema design are used later as the input into the database schema generation process. Form types not considered in database schema design are not used in the database schema generation process. They may represent reports for data retrievals only.Each program form type is a tree of component types. A component type has a name, title, number of occurrences, allowed operations and a reference to the parent component type, if it is not a root component type. Name is the component type identifier. All the subordinated component types of the same parent must have different names.

Each instance of the superordinated component type in a tree may have more than one related instance of the corresponding subordinated component type. The number of occurrences constrains the allowed minimal number of instances of a subordinated component type related to the same instance of a superordinated component type in the tree. It may have one of two values: 0-N or 1-N. The 0-N value means that an instance of a superordinated component type may exist while not having any related instance of the corresponding subordinated component type. The 1-N value means that each instance of a superordinated component type must have at least one related instance of the subordinated component type.

The allowed operations of a component type denote database operations that can be performed on instances of the component type. They are selected from the set {*read*, *insert*, *update*, *delete*}.

A designer can also define component type display properties that are used by the program generator. The concept of component type display is defined by properties: window layout, data layout, relative order, layout relative position, window relative position, search functionality, massive delete functionality and retain last inserted record.

Window layout has two possible values: “New window” and “Same window” and specifies if the component type is to be placed in a new window or in the same window as the parent component type. Data layout specifies the way of component type representation in a screen form. Two values are possible:

"Field layout" or "Table layout". By the "Field layout", only one record at a time is displayed in a form. By the "Table layout", a set of records at a time is displayed in a screen form, in a form of a table. The relative order is a sequence number representing the order of a component type relative to the other sibling component types of the same parent in a form type tree. The layout relative position represents the component type relative position to the parent component type. We may select "Bottom to parent" value if we want to place the component type below the layout of the parent component type in a generated screen form, or "Right to parent" value if it is to be placed right to the parent one. Window relative position is to be specified only when "New window" layout is selected. A designer may specify one of the three possible values: "Center", "Left on top", or "Custom". The "Center" value denotes that the center of a new window is positioned to match the center of the parent window. "Left on top" specifies that the top left corner of the new window will match the top left corner of the parent window. By selecting the "Custom" value, a relative position of the new window top left corner to the top left corner of the parent window is explicitly specified by giving X and Y relative positions.

"Search functionality" represents the Boolean property that enables generation of the filter for data selection. If search functionality is enabled, end-users are allowed to refine the WHERE clause of a SQL SELECT statement. If checked, "massive delete functionality" provides a generation of a delete option next to each record in a table layout. The "retain last inserted record" property specifies if the last inserted record is to be retained on the screen for future use.

Each component type includes one or more attributes. A component type attribute is a reference to a project attribute from the Fundamentals category. It has a title that will appear in the generated screen form. Also, it may be declared as mandatory or optional on the screen form. The allowed operations of a component type attribute denote database operations that can be performed on the attribute, by means of the corresponding screen item. They are selected from the set {*query*, *insert*, *update*, *nullify*}. For a component type attribute a designer may also specify display properties and by this define its presentation details in the screen form. The display properties are specified in the same way as it is for attribute specifications. Values of the display properties may be inherited from the attribute specification or overridden.

So as to unify the layout formatting rules of selected component type attributes, a designer may group them into items groups. Each item group may include one or more component type attributes or other item groups from the same component type. Any item group has its name, title, context and overflow properties. The name and title are mandatory properties. Context and overflow are Boolean properties, specifying if an item group is to be used for presenting layout contextual information or as a layout overflow area.

Each component type attribute provides defining a "List of values" (LOV) functionality. To do that, a designer needs to reference a form type that will serve as a LOV form type. He or she should also define how an end-user can edit attributes: "Only via LOV" or "Directly & via LOV". "Only via LOV" property

means that attribute value may not be inserted or edited using a keyboard, but only using the LOV. “Directly & via LOV” means that inserting or editing attribute values is provided both via keyboard and LOV. “Filter value by LOV” property specifies if all values from LOV will be displayed, or only those filtered according to the pattern given by an end-user. Restrict expression represents the where clause that is concatenated to the rest of where clause in the SQL statement supporting the LOV.

Each component type has one or more keys. Each component type key comprises one or more component type attributes. It represents the unique identification of a component type instance but only in the scope of its superordinated component instance. Uniqueness constraints may be defined for each component type also. Each component type uniqueness constraint comprises at least one component type attribute, but may have more than one. If uniqueness constraint attributes have non-null values, it is possible to uniquely identify a component type instance but only in the scope of the superordinated component instance.

3.7. Business Application

Business Application concept represents the way to formally describe an IS functionality and is organized through a structure of form types. Each business application has a name and a description. One of the form types included into the structure must be declared as the entry form type of the application. It represents the first transaction program invoked upon the launching of the application. Each business application must have the entry form type. To create the form type structure of an application, a concept of the form type call is used. By the form type calls, designers model execution of calls between generated transaction programs. They are also used to model parameters and passing the values between two transaction programs during the call executions. The concept of a form type call comprises two form types: a calling form type and a called form type.

Any form type may have formal parameters defined. Each formal parameter has a mandatory name as the identifier. It must be related to exactly one domain. In the specification of a form type call, it is possible to associate each parameter to a called form type attribute. By this, a designer specifies to which attributes real parameter values will be passed during the call execution.

For a called form type in a call we need to specify Binding and Options properties. Binding property comprises formal parameters of a called form type. For each parameter a designer specifies how a real argument value is to be passed to the parameter. There are three possible options: “value”, “attribute reference”, or “parameter reference”. The value is a constant that will be passed during a call execution. The “attribute reference” provides a relation to a calling form type attribute that gives a value to be passed to the parameter during a call execution. The “parameter reference” provides a

relation to a calling form type parameter that gives a value to be passed to the parameter during a call execution.

The Options properties comprise: calling method, calling mode, and UI position. Calling method comprises two Boolean properties: a) “Select on open” and b) ”Restricted select”. “Select on open” means that the called form type is opened with an automatic data selection. “Restricted select” allows the data selection in the called form type restricted just to the values of passed parameters. Calling mode specifies a general behavior of the calling form type during the call execution. Three possibilities are allowed: “Modal”, “Non-modal” or “Close calling form”. “Modal” means that a user cannot activate the calling form type while the called form type is opened. “Non-modal” means that both the calling and the called form type are simultaneously active in the screen. “Close calling form” is used to cause the closing of the calling form type during the call execution. UI position specifies how a call will be provided at the level of UI: as a menu item or as a button item.

4. IIS*Case PIM Concepts Usage

For many years, IIS*Case provides visually oriented tools for the IS specification in a formal way. In this section we present a different approach where an IS is modeled using the IIS*Case PIM concepts specified at the level of meta-model in EMF. EMF is not only the framework that provides modeling at the level of meta-models, but also supports model implementations based on the created meta-models. In this section, by an example we illustrate the usage of some PIM concepts belonging to our meta-model trough EMF.

In Fig. 5 we present a part of the project Student Service IS. It represents a form type Student_Grades that refers to information about students’ grades. In the following text the project and its main parts are explained in more details.

Using the Eclipse Modeling Framework (EMF), end-users are able to specify the model of Student Service IS using the IIS*Case PIM concepts. In Fig. 6, we present a part of the formal specification of Student Service IS in a form of a tree structure, created by means of the PIM concepts modeled in EMF. It represents the form type from Fig. 5. In the following text we also explain the model from Fig. 6 in more details.

Modeled IS consists of two application systems: *Student Service* and *Faculty Organization*. *Student Service* application system, referenced in Fig. 5 in the upper left rectangle, is a child application system of the parent application system *Faculty Organization* that is referenced in the upper right corner of Fig. 5. In Fig. 6 we have defined the Project, where a value of the *Name* property is *FacultyIS*. We have also defined at the level of the Project two kinds of application types: a) *System* and b) *Subsystem*. Further, we classified application system *Faculty Organization* as the *System* and *Student Service* as *the Subsystem* application type. In Fig. 6, at the level of the Project

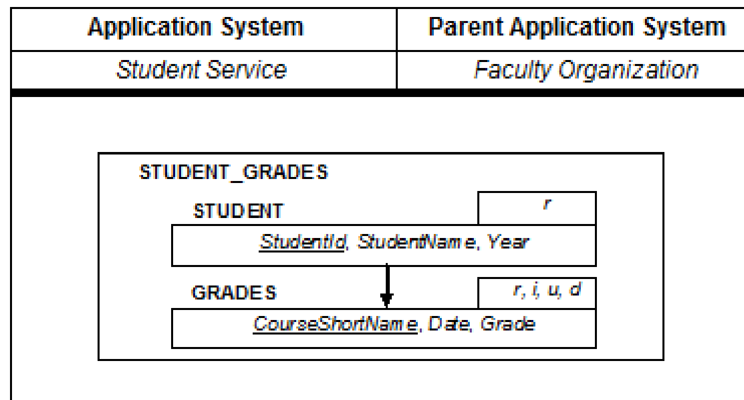


Fig. 5. Application system Student Service

FacultyIS, we have also created a set of attributes, including: *StudentID*, *StudentName*, *Year*, *CourseShortName*, *Date* and *Grade*. The set of these attributes is defined in the Fundamentals category. The attributes defined in the Fundamentals category are later used in the specification of other IS components.

Further, we illustrate the usage of the Form Type concept. We have the form type *Student_Grades,* placed inside the main area of Fig. 5. It has two component types: *Students* and *Grades*. *Student_Grades* form type is presented in Fig. 6 as the Owned Form Type *STG – Student Grades* at the level of the application System *Student Service.* It refers to the information about student grades.

The rectangles that represent *Student* and *Grades* component types are located inside the rectangle representing the form type *Student_grades*. While *Student* component type represents instances of students, *Grades* component type represents instances of grades for each student. *Student* component type is the parent to the *Grades* component type. *Student* and *Grades* component types are modeled in Fig. 6 at the level of the Owned Form Type *STG – Student Grades.*

Allowed database operations for the component type are: read, update, insert and delete. They are presented in Fig. 5 with the abbreviations: r, u, i, d, respectively. The only allowed database operation for *Student* is read, while the allowed operations for *Grades* are read, insert, update and delete. The allowed database operations for the component types are specified in our Project modeled in EMF, although they could not be seen in Fig. 6. End-users of the generated transaction program specified by the form type *Student_Grades* will be able to read data about student instances. They may read, update and delete existing grades for each student, as well as insert new instances of the grades.

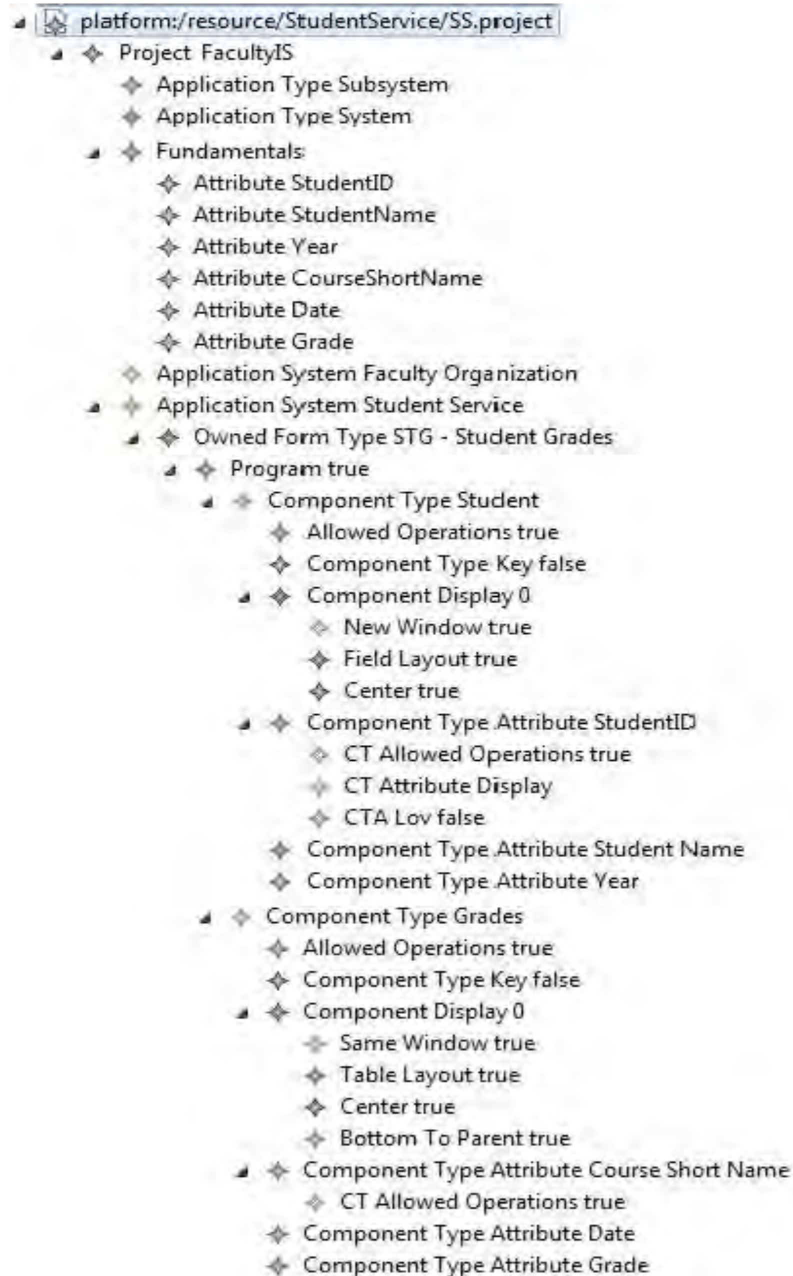


Fig. 6. Model of the Application System Student Service

For each of the *Student* component type attributes, a designer needs to specify its *Name*, *Title*, if it is mandatory or optional for entering values on the

screen form, *Behaviour*, and the list of the *Allowed operations* on the screen form. A set of display and *LOV* properties may also be given. In Fig. 6, at the level of the component type attribute *StudentID*, we presented the properties: *Allowed operations*, *Display* and *LOV*.

In a similar way a designer creates a specification of the *Grades* component type with attributes *CourseShortName*, *Date* and *Grade*. *CourseShortName* is a key of the *Grades* component type.

In this section we have presented an approach to IS conceptual modeling in EMF using IIS*Case PIM concepts. Such approach is valuable not only to create concrete IS models but also to check and validate if IIS*Case PIM concepts are specified correctly and completely. A designer may also use it for fast specification of some IS characteristics. On the other hand, IIS*Case provides specialized, visually oriented and repository based tools supporting the same modeling approach. In general, it is expected to be more convenient for the practical usage, since EMF does not have specialized functionalities and tools to make the IS development process easier for designers.

5. A Concrete Syntax Generation

Generation of the concrete syntax is one of the important steps in the process of the implementation of some DSL. One of our research goals is an implementation of the DSL that will assist in the IS design process. We need to specify the grammar that defines the structure and semantics of the concepts at the meta-level. Such specification actually represents a DSL that could be used in the process of conceptual IS modeling.

A concrete syntax definition is based on the abstract syntax. While concrete syntax expresses a user's perception of a language, the abstract syntax expresses a viewpoint close to the compiler. A DSL implemented for the IIS*Case tool may be used by IS designers. Our plan is to develop a model checker using the abstract syntax specified by EMF. By this, we create a possibility of checking the formal correctness of models, during the whole process of the IS modeling. It is an important feature of each modeling environment aimed at providing IS development in a formal way.

There are different tools for the DSL development. They provide different approaches and techniques to the DSL implementation process. A meta-model specified by Ecore meta-meta-model in EMF may be used as the abstract syntax specification in Eclipse plug-in named EMF text. As we have already developed the meta-model under the EMF using Ecore meta-meta-model, we have decided to use EMF text plug-in and test if the IIS*Case meta-model as the abstract syntax specification may be transformed to the equivalent concrete syntax.

In this section we present only a small part of the concrete syntax grammar, successfully generated by the EMF text plug-in. The IIS*Case meta-model specified by the Ecore meta-meta-model was the input specification for the generation process. The concrete syntax is the output

specification. It is expressed in Human Usable Textual Notation (HUTN) [26] that provides concrete textual language representations for any MOF model. In the following text we present the concrete syntax rules only for the main IIS*Case PIM concepts.

Production rule for defining a *Project* is:

```
Project ::= "Project" "{" "ProjectName" ":"
ProjectName['"','"'] ("NewApplicationType" ":"
NewApplicationType | "NewFundamentalConcept" ":"
NewFundamentalConcept | "NewApplicationSystem" ":"
NewApplicationSystem)* "}";
```

It specifies a name of a project (ProjectName), possible types of application systems (NewApplicationType), different fundamental concepts (NewFundamentalConcept) and application systems (NewApplicationSystem) created in the context of the project.

The rule for specification of an *Application System* is:

```
ApplicationSystem ::= "ApplicationSystem" "{"
"AppSystemName" ":" AppSystemName['"','"']
"AppSystemDescription" ":" AppSystemDescription['"','"']
"AppSystemType" ":" AppSystemType[]
("ParentAppSystem" ":" ParentAppSystem[])?
( "JoinDependency" ":" JoinDependency[] |
"ClosureGraph" ":" ClosureGraph['"','"'] | "BA" ":" BA |
"NewFormType" ":" NewFormType  |
"RelationScheme" ":" RelationScheme[])* "}";
```

It requires specifying the application system name (AppSystemName), description (AppSystemDescription), a type of the application system (AppSystemType), parent application system (ParentAppSystem) created join dependencies (JoinDependecy), a closure graph (ClosureGraph), business applications (BA), form type categories (NewFormType), and generated relation schemes (RelationScheme).

The generated rule for defining *Primitive domain* is:

```
PrimitiveDomain ::= "PrimitiveDomain" "{"
"DomainName" ":" DomainName['"','"']
"Description" ":" Description['"','"']
("DefaultValue" ":" DefaultValue['"','"'])?
("Comment" ":" Comment['"','"'])?
("DecimalPlaces" ":" DecimalPlaces[INTEGER])?
"LenReq" ":" LenReq[] "}";
```

It describes a domain name (DomainName), a description (Description) and a comment (Comment) for the domain, a default value (DefaultValue), decimal places value (DecimalPlaces) and a required length (LenReq).

Production rule for specification a *User defined domain* is:

```
UserDefinedDomain ::= "UserDefinedDomain" "{"
"DomainName" ":" DomainName['"','"']
"Description" ":" Description['"','"']
("DefaultValue" ":" DefaultValue['"','"'])?
("Comment" ":" Comment['"','"'])?
("CheckCondition" ":" CheckCondition['"','"'])?
"USDDT" ":" USDDT[]
"DomainDisplay" ":" DomainDisplay "}";
```

Similar to the previous definition PrimitiveDomain we have DomainName, Description, Comment and DefaultValue. The UserDefinedDomain is also specified by the check condition (CheckCondition), a type of the domain (USDDT) and the specification of how the attributes corresponding to the domain will be displayed (DomainDisplay).

Production rule for defining the *Attribute that is incuded in DB* is:

```
AttributeIncludedInDB ::= "AttributeIncludedInDB" "{"
"AttDomain" ":" AttDomain
"AttributeName" ":" AttributeName['"','"']
"Description" ":" Description['"','"']
"AttDisplay" ":" AttDisplay
"AttIncludedDef" ":" AttIncludedDef
"AttTypeIncluded" ":" AttTypeIncluded "}";
```

The attribute is specified by its name (AttributeName), the attribute domain (AttDomain), a description (Description), the specification of how the attribute is displayed (AttDisplay), a definition of the attribute (AttIncludedDef) and a type of the attribute.

Production rule for the specification of the *Attribute that is not incuded in DB* is similar to the previous one:

```
AttributeNonIncludedInDB ::= "AttributeNonIncludedInDB"
"{" "AttDomain" ":" AttDomain
"AttributeName" ":" AttributeName['"','"']
"Description" ":" Description['"','"']
"AttDisplay" ":" AttDisplay
"AttTypeNonIcluded" ":" AttTypeNonIcluded
"AttNonIncludedDef" ":" AttNonIncludedDef "}";
```

Production rule for definintion of *Function* is:

```
Function ::= "Function" "{"
"FunctionName" ":" FunctionName['"','"']
("Description" ":" Description['"','"'])?
("FunctionBody" ":" FunctionBody['"','"'])?
("FuncParamList" ":" FuncParamList)*
("FunctionReturnType" ":" FunctionReturnType[])? "}";
```

Each function is described by its name (FunctionName), a description (Description), body (FunctionBody), a set of the parameters (FuncParamList) and the function return type (FunctionReturnType).

The specification rule of the *Parameter* is:

```
Parameter ::= "Parameter" "{"
"ParameterSeqNo" ":" ParameterSeqNo[INTEGER]
"ParameterName" ":" ParameterName['"','"']
("ParameterDefValue" ":" ParameterDefValue['"','"'])?
"ParamInOut" ":" ParamInOut
"ParamDomain" ":" ParamDomain[] "}";
```

It requires the definition of a sequence number in the list (ParameterSeqNo), a parameter name (ParameterName), a default value (ParameterDefValue), a type (ParamInOut), and a domain the parameter is corresponding to (ParamDomain)

Production rule for specification of a *Business application* is:

```
BussinesApplication ::= "BussinesApplication" "{"
"BussinesAppName" ":" BussinesAppName['"','"']
"BussinesAppDescription" ":"
BussinesAppDescription['"','"']
("BAEntryFT" ":" BAEntryFT[])* "}";
```

It describes a business application by its name (BussinesAppName), description (BussinesAppDescription), and the entry form type (BAEntryFT).

ReferencedFormType production rule is:

```
ReferencedFormType ::= "ReferencedFormType" "{"
"FormTypeName" ":" FormTypeName['"','"']
("FTCalledFT" ":" FTCalledFT[]
"RefFTAppSys" ":" RefFTAppSys[])* "}";
```

Each referenced form type has its name (FormTypeName), the reference to the called form type (FTCalledFT), and the application system (RefFTAppSys).

Production rule for the definition of an *OwnedFormType* is:

```
OwnedFormType ::= "OwnedFormType" "{"
"FormTypeName" ":" FormTypeName['"','"']
("FTCalledFT" ":" FTCalledFT[])*
"FormTypeTitle" ":" FormTypeTitle['"','"']
("FormTypeFrequency" ":" FormTypeFrequency[INTEGER])?
("FormTypeResponseTime" ":"
FormTypeResponseTime[INTEGER])?
("FTParam" ":" FTParam)*
"DefineFormTypeUsage" ":" DefineFormTypeUsage "}";
```

It requires the definition of a form type specifying its name (FormTypeName), title (FormTypeTitle), the form type that is called (FTCalledFT), frequency (FormTypeFrequency), usage (DefineFormTypeUsage), and the response time (FormTypeResponseTime) of the form type, and the list of the form type parameters (FTParam).

Production rule that represents *Program* definition is:

```
Program ::= selectedFormTypeUsage
["selectedFormTypeUsage" : ""] "Program" "{"
"ConsideredINDBSchDesign" ":" ConsideredINDBSchDesign[]
("NewComponentType" ":" NewComponentType)* "}";
```

The production rule specifies the program by the component type tree structure that consists of a set of component types.

Production rule for the definition of a *Component type* is:

```
ComponentType ::= "ComponentType" "{"
"CompTypeName" ":" CompTypeName ['"','"']
"NoOfOccurrences" ":" NoOfOccurrences['"','"']
"CompTypeTitle" ":" CompTypeTitle['"','"']
"AO" ":" AO  ("IG" ":" IG[])*
("CTU" ":" CTU)*
("CompTypeKey" ":" CompTypeKey)*
("CompTypeCheckConstraint" ":"
CompTypeCheckConstraint['"','"'])?
"CompTypeCompDisplay" ":" CompTypeCompDisplay
("CompTypeParent" ":" CompTypeParent [])?
("CTAttrib" ":" CTAttrib)* "}";
```

ComponentType rule describes a component type specifying its name (CompTypeName), number of occurrences (NoOfOccurrences), a title (CompTypeTitle), allowed operations for the component type (AO), the item group (IG), the unique constraint (CTU) and the key (CompTypeKey).

Production rule for the definition for the *Component type attribute* is:

```
ComponentTypeAttribute ::= CompTypeAttribMandatory
["CompTypeAttribMandatory " : ""]
"ComponentTypeAttribute" "{"
"CompTypeAttribTitle" ":" CompTypeAttribTitle ['"','"']
("CompTypeAttribBehavior" ":" CompTypeAttribBehavior
['"','"'])?
("CTADeafultValue" ":" CTADeafultValue['"','"'])?
"CompTypeAttribName" ":" CompTypeAttribName []
("CTAttribFunction" ":" CTAttribFunction[])?
"CTAttribAO" ":" CTAttribAO "CTAttribLov" ":" CTAttribLov
"CTAttribDisplay" ":" CTAttribDisplay "}";
```

Each component type attribute has its title (CompTypeAttribTitle), behavior specification (CompTypeAttribBehavior), a default value (CTADeafultValue), a reference to the attribute (CompTypeAttribName), a reference to the function (CTAttribFunction), allowed operations for the component type attribute (CTAttribAO), list of values (CTAttribLov), and the set of display properties (CTAttribDisplay).

In Fig. 7 we present a fragment of the program that corresponds to the example specified in Fig. 5.

Firstly, we have created an instance of the project concept, named *Faculty IS.* After that we have specified attributes (AttributeIncludedInDB) with their *AttributeName* values. New attributes are presented in a form of a new fundamental concept instances. Before the specification of an application system, we need to specify one or more application types at the level of the project. In the example shown in Fig. 7, Project *Faculty IS* comprises two application systems (*ApplcationSystem*). While the first one is a specification of the *Faculty Organization* application system, the other one represents *Student Service* application system. For each instance of the *ApplicationSystem* concept it is necessary to define its name (*AppSystemName*), description (*AppSystemDescription*) and type (*AppSystemType*). In Fig. 7, *FacultyOrganization* is a parent (*ParentAppSystem*) application system for the *StudentService.*

At this stage, in the example in Fig. 7, we define a set of the form types (*NewFormType*) for each application system. For each form type, we specify the name (*FormTypeName*), title (*FormTypeTitle)* and the form type usage. Each form type in Fig 7., has the property values for frequency (*FormTypeFrequency*) and response time (*FormTypeResponseTime*). It also includes a list of component type specifications (*NewComponentType*)*.* Form type *STG – Student Grades* comprises two component types, a parent component type (*CompTypeParent*) *STUDENT* and its child component type *GRADES*.

For each component type, in the example presented in Fig. 7, we define the name (*CompTypeName*), title (*CompTypeTitle*) and the set of display properties (*CompTypeCompDisplay*). For *STUDENT* component type search functionality (*SearchFuncionality*) is enabled. The component type *STUDENT* is to be positioned in a new window (*CompDisplayPosition*) and the data need

to be displayed in a field data layout (*CompDispplayDataLayout*). For each component type, we also define component type attributes (*CompTypeAttribute*). The definition of component type attributes requires the name (*CompTypeAttribName)* and the title (*CompTypeAttribTitle*) to be specified.

After the list of component type attributes, the list of component type constraints is given. We may give specifications of key, uniqueness and check constraints. In the example shown in Fig. 7, only component type keys are specified for *STUDENT* by the property *CompTypeKey*.

```
Project {
  ProjectName : "Faculty IS"
  //definition of the fundamental concepts
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "StudentID"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "StudentName"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "Year"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "CourseShortName"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "Date"
    }
  NewFundamentalConcept :
    AttributeIncludedInDB {
      AttributeName : "Grade"
    }
  //definition of the applicaiton types
  NewApplicationType :
    ApplicationType {
      ApplicationTypeName : "ProjectSubsystem"
    }
  //definition of the applicaiton systems
  NewApplicationSystem :
    ApplicationSystem {
      AppSystemName : "FacultyOrganization"
      AppSystemDescription : "A unit of a Faculty IS"
      AppSystemType : ProjectSubsystem
```

```
    }
  NewApplicationSystem :
    ApplicationSystem {
      AppSystemName : "StudentService"
      AppSystemDescription : "A unit of a FacultyOrgan."
      AppSystemType : ProjectSubsystem
      ParentAppSystem : FacultyOrganization
      //definition of the new form type
      NewFormType :
        OwnedFormType {
        FormTypeName : "STG-StudentGrades"
        FormTypeTitle : "Catalogue of student grades"
        DefineFormTypeUsage :
          Program {
            ConsideredINDBSchDesign : true
            //definition of the new component type
            NewComponentType :
              ComponentType {
               CompTypeName : "STUDENT"
               CompTypeTitle : "Student Records"
                CompTypeCompDisplay :
                 SearchFuncionality
                 ComponentDisplay {
                  CompDisplayPosition : NewWindow { }
                  CompDispplayDataLayout :FieldLayout { }
                 }
                 CompTypeAttribute :
                  ComponentTypeAttribute {
                    CompTypeAttribName : StudentID
                    CompTypeAttribTitle : "StudentId"
                    CompTypeAttribBehavior : "queryOnly"
                  }
                 CompTypeAttribute :
                  ComponentTypeAttribute {
                    CompTypeAttribName : StudentName
                  }
                 CompTypeAttribute :
                  ComponentTypeAttribute {
                    CompTypeAttribName : Year
                  }
                 CompTypeKey :
                  ComponentTypeKey {
                    CompTypeKeyAttribute : StudentId
                  }
              }
            //definition of the new component type
            NewComponentType :
              ComponentType {
```

```
                CompTypeName : "GRADES"
                CompTypeParent : STUDENT
                  CompTypeCompDisplay :
                   ComponentDisplay {
                    CompDisplayLayoutRelativePosition :
                                        BottomToParent { }
                    CompDispplayDataLayout:TableLayout { }
                   }
      //definition of the new component type continues
              }
          }
          FormTypeFrequency : 1
          FormTypeResponseTime : 1
        }
    }
}
```

Fig. 7. A fragment of program that corresponds to the example in Fig. 5

In this section we presented only a small part of the concrete syntax of a DSL that assists in the process of an IS development. As the process of concrete syntax generation is automatic, we can easily produce a new language based on the whole IIS*Case meta-model. Generated language provides the syntax and semantics for creating the PIM specifications of an IS, which is one of the most important activities in our approach to IS development process.

6. Conclusion

In this paper we presented a part of the IIS*Case PIM meta-model, created by the use of the MOF 2.0 meta-meta model specification. Our intention was not to present all the elements of our meta-model in detail. Instead, we tried to focus just on those meta-model details that are necessary to give a general picture of the model. We believe that the formal specification of our meta-model is not for documentation purposes only. It is also a necessary step in creating a textual DSL to support IS design and give another view of the IS description. In this paper we have presented only one part of the concrete syntax generated from the IIS*Case PIM meta-model. The syntax of such a DSL is not simple. It is a consequence of the complexity of our IIS*Case PIM meta-model. One of the further steps is to generate the whole concrete syntax of the DSL. The concrete syntax should be developed for the textual DSL, although we plan to support the visual approach, too.

The abstract syntax specified by the MOF model is the input specification for the development of the model checker. We may use the IIS*Case PIM meta-model in the verification of generated relational database schemas. Currently, IIS*Case supports an assistance to designers in detecting formal

conflicts at the level of relational database model. By this, the algorithms for detection and resolving constraint collisions at the level of relational data model has already been implemented in IIS*Case. In our future research, we may extend this support so as to assist designers at the level of created PIM models in searching for the appropriate solutions of detected problems. In this way, the process of collision resolving will be raised to the PIM level of abstraction.

Our further research will include experiments with other technologies that rely on MOF. The presented meta-model is a good base for a research in the area of Query View Transform (QVT) set of languages. Our intention is to embed into IIS*Case transformations between different data models. Providing data model transformations may play an important role in the IS design process. In the course of data reengineering process, our plan is to provide the data integration from various sources based on different data models. Data transformation rules specified by QVT could be applied at the level of meta-models specified by various data-models, all expressed in a unified manner in MOF. Our intention is to provide transformations of the models specified in IIS*Case to the UML models. Providing such transformations we allow designers to have models specified in UML standard with OCL constraints.

Acknowledgment. The research presented in this paper was supported by Ministry of Education and Science of Republic of Serbia, Grant III-44010: Intelligent Systems for Software Product Development and Business Support based on Models.

References

1. I. Luković, P. Mogin, J. Pavićević, S. Ristić, “An Approach to Developing Complex Database Schemas Using Form Types”, Software: Practice and Experience, 2007, DOI: 10.1002/spe.820, Vol. 37, No. 15, pp. 1621-1656.
2. I. Luković, M. J. Varanda Pereira, N. Oliveira, D. Cruz, P. R. Henriques, “A DSL for PIM Specifications: Design and Attribute Grammar based Implementation”, Computer Science and Information Systems (ComSIS), ISSN: 1820-0214, DOI: 10.2298/CSIS101229018L, Vol. 8, No. 2, 2011, pp. 379-403.
3. N. Oliveira, M. J. Varanda Pereira, P. R. Henriques, D. Cruz, B. Cramer, “VisualLISA: A Visual Environment to Develop Attribute Grammars”, Computer Science an Information Systems, (ComSIS), ISSN:1820-0214, Vol. 7, No. 2, 2010, pp. 265-289.
4. M. Čeliković, I. Luković, S. Aleksić, V. Ivančević, "A MOF based Meta-Model of IIS*Case PIM Concepts", Federated Conference on Computer Science and Information Systems (FedCSIS), 3rd Workshop on Advances in Programming Languages (WAPL 2011), September 18-21, 2011, Szczecin, Poland, Proceedings, IEEE Computer Society Press and Polish Information Processing Society, ISBN 978-83-60810-39-2, pp. 833-840.
5. Object Management Group (OMG), OCL SpecificationVersion 2.0, [Online] Available: http://www.omg.org/docs/ptc/05-06-06.pdf, June 2005.
6. Eclipse Modeling Framework, [Online] Available: http://www.eclipse.org/ modeling/emf/.

7. Meta-Object Facilty, [Online] Available: http://www.omg.org/mof/.
8. L. Baresi, F. Garzotto, M. Maritati, "W2000 as a MOF Metamodel." In Proc. of the 6th World Multiconference on Systemics, Cybernetics and Informatics - Web Engineering track. Orlando, USA, 2002.
9. A. Schauerhuber, M. Wimmer, E. Kapsammer, "Bridging existing web modeling languages to model-driven engineering: A metamodel for webML", International Workshop on Model Driven Web Engineering (2nd), Palo Alto, CA, 2006.
10. Document Type definition (DTD), [Online] Available: http://www.w3.org/TR/html4/sgml/dtd.html.
11. M. Richters, M. Gogolla, "A meta-model for OCL" In Proc. of the 2nd international conference on The unified modeling language beyond the standard, ISBN:3-540-66712-1, 1999.
12. F. Jouault, J. Bézivin, "KM3: a DSL for Metamodel Specification", In Proc. of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy, 2006, Springer LNCS 4037, pp. 171-185.
13. B. Perišić, G. Milosavljević, I. Dejanović, B. Milosavljević, "UML Profile for Specifying User Interfaces of Business Applications", Computer Science and Information Systems (ComSIS), ISSN: 1820-0214, DOI: 10.2298/CSIS110112010P, Vol. 8, No. 2, 2011, pp. 405-426.
14. Živanov Ž., Rakić P., Hajduković M.: "Using Code Generation Approach in Developing Kiosk Applications", Computer Science and Information Systems, (ComSIS), ISSN:1820-0214, Vol. 5, No. 1, 2008, pp. 41-59.
15. Dejanović I., Milosavljević G., Perišić B., Tumbas M.: A Domain-Specific Language for Defining Static Structure of Database Applications, Computer Science and Information Systems, (ComSIS), ISSN:1820-0214, Vol. 7, No. 3, 2010, pp. 409-440.
16. Van Deursen A, Klint P, Visser J: Domain-specific languages: an annotated bibliography, ACM SIGPLAN Not 35(6), 2000, pp. 26–36.
17. Mernik M., Heering J., Sloane A.M.: When and how to develop domain-specific languages, ACM Computing Surveys, 2005, Vol. 37, No. 4, pp. 316–344.
18. Kos, T., Kosar, T., Knez, J., Mernik, M.: From DCOM interfaces to domain-specific modeling language: A case study on the Sequencer. , Computer Science and Information Systems, (ComSIS), ISSN:1820-0214, Vol. 8, No. 2, 2011, pp. 361-378.
19. GME: Generic Modeling Environment, [Online] Available: http://www.isis.vanderbilt.edu/Projects/gme/.
20. MetaCase Metaedit+, [Online] Available: http://www.metacase.com/.
21. Kelly, S. Lyytinen, K. Rossi,M.: MetaEdit+: a fully configurable multi-user and multi-tool CASE and CAME environment, Advanced Information Systems Engineering 1080, 1996, pp. 1–2.
22. Kelly, S. Tolvanen,J.-P. Domain-Specific Modeling: Enabling Full Code Generation, Wiley–IEEE Computer Society Press, 2008.
23. I. Luković, S. Ristić, P. Mogin, J. Pavićević, "Database Schema Integration Process – A Methodology and Aspects of Its Applying", Novi Sad Journal of Mathematics, Serbia, ISSN: 1450-5444, Vol. 36, No. 1, 2006, pp. 115-150.
24. J. Banović, "An Approach to Generating Executable Software Specifications of an Information System", Ph.D. Thesis, University of Novi Sad, Faculty of Technical Sciences, Novi Sad, 2010.
25. A. Popović, "A Specification of Visual Attributes and Business Application Structures in the IIS*Case Tool", Mr (M.Sc.) Thesis, University of Novi Sad, Faculty of Technical Sciences, 2008.

26. Human Usable Textual Notation (HUTN) [Online] Available http://www.omg.org/spec/HUTN/.

Milan Čeliković graduated in 2009 at the Faculty of Technical Sciences, Novi Sad, at the Department of Computing and Control. Since 2009 he has worked as a teaching assistant at the Faculty of Technical Sciences, Novi Sad, at the Chair for Applied Computer Science. In 2010, he started his Ph.D. studies at the Faculty of Technical Sciences, Novi Sad. His main research interests are focused on: Domain specific modeling, Domain specific languages, Databases and Database management systems. At the moment, he is involved in the projects concerning application of DSLs in the field of software engineering.

Ivan Luković received his M.Sc. (5 year, former Diploma) degree in Informatics from the Faculty of Military and Technical Sciences in Zagreb in 1990. He completed his Mr (2 year) degree at the University of Belgrade, Faculty of Electrical Engineering in 1993, and his Ph.D. at the University of Novi Sad, Faculty of Technical Sciences in 1996. Currently, he works as a Full Professor at the Faculty of Technical Sciences at the University of Novi Sad, where he lectures in several Computer Science and Informatics courses. His research interests are related to Database Systems and Software Engineering. He is the author or coauthor of over 90 papers, 4 books, and 30 industry projects and software solutions in the area.

Slavica Aleksić received her M.Sc. (5 year, former Diploma) degree from Faculty of Technical Sciences in Novi Sad. She completed her Mr (2 year) degree at the University of Novi Sad, Faculty of Technical Sciences. Currently, she works as a teaching assistant at the Faculty of Technical Sciences at the University of Novi Sad, where she assists in teaching several Computer Science and Informatics courses. Her research interests are related to Database Systems, Theory of Data Models, System Design, Logical and Physical Database Design, Development and Usage of MDSE / CASE tools in Software Engineering and System Design, Reengineering of Information Systems and Model Transformations in MDA.

Vladimir Ivančević is a PhD student in Applied Computer Science and Informatics and a teaching assistant at the Faculty of Technical Sciences, University of Novi Sad (Serbia), where he also gained his BSc and MSc in Electrical Engineering and Computing. His research interests include domain specific languages (DSLs), data mining (DM), and databases. At the moment, he is involved in several projects concerning application of DSLs and DM in the fields of software engineering, education, and public health.

Received: February 03, 2012; Accepted: August 17, 2012.

DOI: 10.2298/CSIS111216023S

LL conflict resolution using the embedded left LR parser

Boštjan Slivnik

University of Ljubljana
Faculty of Computer and Information Science
Tržaška cesta 25, 1000 Ljubljana, Slovenia
bostjan.slivnik@fri.uni-lj.si

Abstract. A method for resolving $\mathrm{LL}(k)$ conflicts using small $\mathrm{LR}(k)$ parsers (called embedded left $\mathrm{LR}(k)$ parsers) is described. An embedded left $\mathrm{LR}(k)$ parser is capable of (a) producing the prefix of the left parse of the input string and (b) stopping not on the end-of-file marker but on any string from the set of lookahead strings fixed at the parser generation time. The conditions regarding the termination of the embedded left $\mathrm{LR}(k)$ parser if used within $\mathrm{LL}(k)$ (and similar) parsers are defined and examined in-depth. It is proved that an $\mathrm{LL}(k)$ parser augmented with a set of embedded left $\mathrm{LR}(k)$ parsers can parse any deterministic context-free grammar in the same asymptotic time as $\mathrm{LR}(k)$ parser. As the embedded left $\mathrm{LR}(k)$ parser produces the prefix of the left parse, the $\mathrm{LL}(k)$ parser augmented with embedded left $\mathrm{LR}(k)$ parsers still produces the left parse and the compiler writer does not need to bother with different parsing strategies during the compiler implementation.

Keywords: embedded parsing, left LR parsing, LL conflicts.

1. Introduction

Choosing the right parsing method is an important issue in a design of a modern compiler for at least two reasons. First, the parser represents the backbone of the compiler's front-end as the syntax-directed translation of the source program to the (intermediate) code is based upon it. And second, syntax errors cannot be scrupulously reported without the appropriate support of the parser.

As the study of available open-source compilers reveal [18], nearly all of the most popular parsing methods nowadays belong to one of the two large classes, namely LL and LR [16, 17]. LR parsing, the most popular bottom-up parsing method, is generally praised for its power while LL parsing, the principal top-down method, is credited for being simpler to implement and debug, and better for error recovery and the incorporation of semantic actions [14].

Many variations of the original LL and LR parsing methods [7, 8] have been devised since their discovery decades ago. Some methods, e.g., SLL, SLR and LALR [16, 17], focus on reducing the space complexity by producing smaller parsers (either less code or smaller parsing tables), and some tend to produce

faster parsers [1]. Other methods extend the class of languages that can be parsed by the canonical LL or LR parsers. Methods like GLR and GLL are able to parse all context-free languages in qubic time (compared with the linear time achieved by the classical LL and LR methods) [21, 22, 15, 14] while $\mathrm{LL}(*)$ parsers (produced by the popular ANTLR parser generator) are able to parse even some context-sensitive languages by resorting to backtracking in some cases [11]. Finally, some methods modify the behavior of the LR parsing so that by producing the left parse of the program being compiled instead of the right parse, they behave as if the top-down, e.g., LL, was used [13, 20].

The discourse on whether LL or LR parsing is more suitable either in general or in some particular case still goes on. It has been reignited lately by the online paper entitled “Yacc is dead” [10] and two issues have been made clear (again): first, parser generators are appreciated, and second, both methods, LR and LL, remain attractive [18].

To combine the advantages of both bottom-up and top-down parsing, left corner parsing was introduced [12, 3]. Basically it uses the top-down parsing and switches to bottom-up parsing to parse the left corner of each derivation subtree. However, modern variations switch to bottom-up parsing only when bottom-up parsing is needed indeed [6, 2]. Left corner parsing never gained much popularity, most likely because it produces a mixed order parse which makes incorporating semantic actions tricky.

As described, left corner parsing uses bottom-up parsing to resolve the problems arising during the top-down parsing while $\mathrm{LL}(*)$ parser uses DFAs for LL conflict resolution. The former produces a tricky parse and the latter must always rescan the symbols already scanned by a DFA. In this paper an embedded left $\mathrm{LR}(k)$ parser which can be used within an $\mathrm{LL}(k)$ parser instead of a DFA, is proposed. As it produces the left parse it does not require rescanning of tokens already scanned or backtracking, and thus guarantees the linear parsing time for all $\mathrm{LR}(k)$ grammars.

Another method, namely packrat parsing [4], could perhaps have been used to resolve $\mathrm{LL}(k)$ conflicts, but there are two obstacles. First, packrat parsers are made for parsing expression grammars where the productions are ordered — the conversion of a context-free grammar to a parsing expression grammar is tricky even for the human and cannot be made by the parser generator. Second, packrat parsers do not handle left recursion well — something in particular that the embedded left $\mathrm{LR}(k)$ parser must handle instead of $\mathrm{LL}(k)$ parser.

The problem, i.e., the requirements for embedding an $\mathrm{LR}(k)$ parser into the $\mathrm{LL}(k)$ parser, is formulated in Section 2. The solution is described in Sections 3 and 4: the former contains the solution of correct termination of the embedded left $\mathrm{LR}(k)$ parser while the latter contains how the parser can produce the shortest prefix of the left parse as soon as possible. The evaluation of the embedded left $\mathrm{LR}(k)$ parser is given in Section 5 together with a brief evaluation of the new parser.

An intermediate knowledge of LL and LR parsing is presumed. The notation used in [16] and [17] is adopted except in two cases. First, a single parser

step is not described by relation $\Longrightarrow$ (as if a pushdown automaton is defined as one particular kind of a rewriting system [16]) but by relation $\vdash$ among the instantaneous descriptions of a pushdown automaton [5]. Second, the notation $[A \to \alpha \bullet \beta, x]$ where $S \Longrightarrow^{*}_{\mathrm{rm}} \gamma' A v \Longrightarrow_{\mathrm{rm}} \gamma' \alpha\beta v = \gamma\beta v$ and $x \in \mathrm{FIRST}_k(z)$, denotes the $\mathrm{LR}(k)$ item valid for γ.

Finally, it is assumed that the result the parser produces is the *left (right) parse* of the input string, i.e., the (reversed) list of productions needed to derive the input string from the initial grammar symbol using the leftmost (rightmost) derivation.

2. On resolving $\mathrm{LL}(k)$ conflicts

Consider an $\mathrm{LR}(k)$ but non-$\mathrm{LL}(k)$ grammar $G = \langle N, T, P, S\rangle$, i.e., $G \in \mathrm{LR}(k) \setminus \mathrm{LL}(k)$. If the input string $w \in L(G)$ is derived by a derivation

$$S \Longrightarrow^{\pi_u}_{G,\mathrm{lm}} uA\delta \Longrightarrow^{\pi_{v'}}_{G,\mathrm{lm}} uv'\delta \Longrightarrow^{\pi_{v''}}_{G,\mathrm{lm}} uv'v'' = uv = w \quad , \tag{1}$$

the expected result of parsing it with an $\mathrm{LL}(k)$ parser is the left parse

$$\pi_w = \pi_u \pi_{v'}\ \pi_{v''} \in P^* \quad . \tag{2}$$

Since $G \notin \mathrm{LL}(k)$, an $\mathrm{LL}(k)$ conflict is likely to occur and must therefore be resolved. $\mathrm{LL}(*)$ parsing [11], for instance, tries to determine the next production using a set of DFAs: if A causes an $\mathrm{LL}(1)$ conflict in the derivation (1), a DFA for A determines the next production by scanning the first few (but sometimes more) tokens of the string $v = v'v''$; afterwards the $\mathrm{LL}(*)$ parser continues parsing by reading the entire string v again (not just the unscanned suffix of it). While $\mathrm{LL}(*)$ parser produces the left parse (2), it reads some tokens more than once and in some cases it must even resort to backtracking (if the DFA cannot determine the next production). Furthermore, $\mathrm{LL}(*)$ parsing prohibits left-recursive productions.

To produce the left parse but to avoid rescanning, backtracking and prohibiting left-recursive productions, small $\mathrm{LR}(k)$ parsers can be used instead of DFAs. However, these small $\mathrm{LR}(k)$ parsers must differ from the classical $\mathrm{LR}(k)$ parsers in two regards:

1. **$\mathrm{LR}(k)$ parsers used within an $\mathrm{LL}(k)$ parser cannot rely on the end-of-input symbol \$ to terminate** (unlike the standard $\mathrm{LR}(k)$ parsers can).
More precisely, if an $\mathrm{LR}(k)$ parser is to be used for parsing the substring v' of the string w derived by the derivation (1), it must be capable of terminating with any string $x \in \mathrm{FIRST}^G_k(\delta\$)$ in its lookahead buffer (instead of $\$$).
2. **$\mathrm{LR}(k)$ parsers used within an $\mathrm{LL}(k)$ parser must produce the left parse of its input** (instead of the right parse as the standard $\mathrm{LR}(k)$ parsers do).
More precisely, a standard $\mathrm{LR}(k)$ parser for A produces the right parse of v', but if used within an $\mathrm{LL}(k)$ parser, it should produce the left parse $\pi_{v'}$.

If an $\mathrm{LR}(k)$ parser fulfills both conditions, it is called the *embedded left* $\mathrm{LR}(k)$ *parser*: embedded as it can be used within the *backbone* $\mathrm{LL}(k)$ parser, and left as it produces the left parse and thus guarantees that the overall result of parsing is also the left parse.

3. Termination of the embedded $\mathrm{LR}(k)$ parser

The main problem regarding the termination of the embedded $\mathrm{LR}(k)$ parser can be explained most conveniently by the following example.

Example 1. Consider the grammar G_{ex1} with the start symbol S and productions

$$S \longrightarrow aAb \,|\, bAab \quad \text{and} \quad A \longrightarrow Aa \,|\, a \quad .$$

As A is a left-recursive nonterminal, it causes the $\mathrm{LL}(1)$ conflict whenever a is in the lookahead buffer of the $\mathrm{LL}(1)$ parser.

If the input string starts with aa, then after the first two steps, namely

$$\$S\,\mathbf{I}\,aa\ldots\$ \;\vdash_{\mathrm{LL}}\; \$bAa\,\mathbf{I}\,aa\ldots\$ \;\vdash_{\mathrm{LL}}\; \$bA\,\mathbf{I}\,a\ldots\$ \quad ,$$

the backbone $\mathrm{LL}(1)$ parser reaches the configuration $\$bA\,\mathbf{I}\,a\ldots\$$ (the strings on the left side and on the right side of $\mathbf{I}$ represent the stack contents and the remaining (yet unscanned) part of the input, respectively; the topmost stack symbol and the contents of the lookahead buffer are close to $\mathbf{I}$). The configuration $\$bA\,\mathbf{I}\,a\ldots\$$ exhibits an $\mathrm{LL}(1)$ conflict on $A\,\mathbf{I}\,a$. At this point, an embedded $\mathrm{LR}(1)$ parser for A should be used: as b is never derived from A, it can function as the end-of-input marker.

If the input string starts with baa, then $\mathrm{LL}(1)$ parsing starts as

$$\$S\,\mathbf{I}\,baa\ldots\$ \;\vdash_{\mathrm{LL}}\; \$baAb\,\mathbf{I}\,baa\ldots\$ \;\vdash_{\mathrm{LL}}\; \$baA\,\mathbf{I}\,aa\ldots\$ \quad .$$

The backbone $\mathrm{LL}(1)$ parser reaches the configuration $\$baA\,\mathbf{I}\,aa\ldots\$$ where the embedded $\mathrm{LR}(1)$ parser must be used. This time the embedded $\mathrm{LR}(1)$ parser for A cannot be used as it cannot stop on a that follows A in the production $S \longrightarrow bAab$. More precisely, after shifting the first a on the stack and reducing it to A, i.e.,

$$\$[\varepsilon]\,\mathbf{I}\,aa\ldots\$ \;\vdash_{\mathrm{LR}}\; \$[\varepsilon][a]\,\mathbf{I}\,a\ldots\$ \;\vdash_{\mathrm{LR}}\; \$[\varepsilon][A]\,\mathbf{I}\,a\ldots\$ \quad ,$$

the embedded $\mathrm{LR}(1)$ parser faces the second a in its lookahead buffer, but it cannot determine whether it should be shifted or not. If the entire input is $baab$, the embedded $\mathrm{LR}(1)$ parser should terminate and handle the control back to the backbone $\mathrm{LL}(1)$ parser, otherwise it should continue by shifting and reducing using $A \longrightarrow Aa$. Therefore, the embedded $\mathrm{LR}(1)$ parser for Aa, i.e., one that can terminate on b for the same reason as above, must be used instead of the one for A.

(Modifying the problem to any k is left as an exercise.) ■

Two conclusions follow from Example 1:

1. **The embedded $\mathbf{LR}(k)$ parser must sometimes parse substrings derived from a sentential form starting with the $\mathbf{LL}(k)$-conflicting nonterminal instead of from that nonterminal only.** More precisely, if the first part of the derivation (1) is rewritten as

$$S \Longrightarrow^{\pi_{u'}}_{G'\mathrm{lm}} u'B\delta' \Longrightarrow_{G'\mathrm{lm}} u'\beta_1 A\beta_2\delta' \Longrightarrow^{\pi_{u''}}_{G'\mathrm{lm}} u'u''A\beta_2\delta' = uA\delta \quad , (3)$$

   the parser for $A\beta_2'$, where $\beta_2 = \beta_2'\beta_2''$ in $B \longrightarrow \beta_1 A\beta_2$, might be needed instead of the parser for A. In Example 1 a parser for Aa is needed in production $S \longrightarrow bAab$ instead of a parser for A.
2. **The right context of the left sentential form the embedded $\mathbf{LR}(k)$ parser is made for, is important.** More precisely, the right context is the prefix of the string that comes after the string derived from the sentential form the embedded parser is made for, i.e., in the derivation (3) the termination of the embedded $\mathrm{LR}(k)$ parser for $A\beta_2'$ depends on the contents of the set $\mathrm{FIRST}_k^G(\beta_2''\delta')$.

Hence, in general an embedded $\mathrm{LR}(k)$ parser for $A\beta_2'$ capable of termination on any string from $\mathrm{FIRST}_k^G(\beta_2''\delta')$ is needed.

The easiest way to resolve the right context of the embedded $\mathrm{LR}(k)$ parser is to transform grammar $G = \langle N, T, P, S\rangle$ into grammar $\bar{G} = \langle \bar{N}, T, \bar{P}, \bar{S}\rangle$ by applying the transformation of an $\mathrm{LL}(k)$ grammar to an $\mathrm{SLL}(k)$ grammar [17]: in the transformed grammar $\bar{G}$ each nonterminal occurs in exactly one right context. More precisely, the start symbol becomes $\bar{S} = \langle S, \{\varepsilon\}\rangle$ and the set $\bar{N}$ of nonterminals is defined as

$$\bar{N} = \{\langle A, \mathcal{F}_A\rangle;\ S \Longrightarrow^*_{\mathrm{lm}} uA\delta \wedge \mathcal{F}_A = \mathrm{FIRST}_k^G(\delta)\} \quad .$$

For any nonterminal $\langle A, \mathcal{F}_A\rangle$ the new set $\bar{P}$ of productions includes productions

$$\langle A, \mathcal{F}_A\rangle \longrightarrow \bar{X}_1\bar{X}_2\dots\bar{X}_n$$

where, for any $i = 1, 2, \dots, n$,

$$\bar{X}_i = \begin{cases} X_i & X_i \in T \\ \langle X_i, \mathrm{FIRST}_k^G(X_{i+1}X_{i+2}\dots X_n\mathcal{F}_A)\rangle & X_i \in N \end{cases}$$

provided that $A \longrightarrow X_1X_2\dots X_n \in P$. (This transformation does not introduce any new $\mathrm{LL}(k)$ conflicts; in fact, if $k > 1$, it even reduces the number of $\mathrm{LL}(k)$ conflicts for some non-$\mathrm{SLL}(k)$ grammars [17].)

Example 2. If the grammar G_{ex1} is transformed, a grammar $\bar{G}_{\mathrm{ex1}}$

$$\begin{aligned} \langle S, \{\varepsilon\}\rangle &\longrightarrow a\langle A, \{b\}\rangle b \,|\, b\langle A, \{a\}\rangle ab \\ \langle A, \{a\}\rangle &\longrightarrow \langle A, \{a\}\rangle a \,|\, a \\ \langle A, \{b\}\rangle &\longrightarrow \langle A, \{a\}\rangle a \,|\, a \end{aligned}$$

is obtained. Two embedded $\mathrm{LR}(1)$ parsers are needed: $\langle A, \{b\}\rangle$ and $\langle Aa, \{b\}\rangle$:

1. The parser for $\langle A,\{b\}\rangle$ results from production $\langle S,\{\varepsilon\}\rangle \longrightarrow a\langle A,\{b\}\rangle b$: the parser's right context is $\{b\} = \mathrm{FIRST}_k^G(b\{\varepsilon\})$: b follows $\langle A,\{b\}\rangle$ in production $\langle S,\{\varepsilon\}\rangle \longrightarrow a\langle A,\{b\}\rangle b$ and $\{\varepsilon\}$ (from $\langle S,\{\varepsilon\}\rangle$) determines the right context of the entire production $\langle S,\{\varepsilon\}\rangle \longrightarrow a\langle A,\{b\}\rangle b$.
2. The parser for $\langle Aa,\{b\}\rangle$ results from production $\langle S,\{\varepsilon\}\rangle \longrightarrow b\langle A,\{a\}\rangle ab$, again with the right context $\{b\} = \mathrm{FIRST}_k^G(b\{\varepsilon\})$: b follows the sentential form $\langle A,\{a\}\rangle a$ in production $\langle S,\{\varepsilon\}\rangle \longrightarrow b\langle A,\{a\}\rangle ab$ and $\{\varepsilon\}$ (from $\langle S,\{\varepsilon\}\rangle$) determines the right context of the entire production.

After the $\mathrm{LR}(1)$ parsers are embedded, productions for $\langle A,\{a\}\rangle$ and $\langle A,\{a\}\rangle$ are eliminated as they are no longer needed — the embedded $\mathrm{LR}(k)$ parsers are based on the original grammar G_{ex1}. ■

To resolve conflicts during $\mathrm{LL}(k)$ parsing based on the grammar $\bar{G}$, every production

$$\langle B,\mathcal{F}_B\rangle \longrightarrow \beta_1\langle A,\mathcal{F}_A\rangle\beta_2 \in \bar{P} \quad (4)$$

with an $\mathrm{LL}(k)$-conflicting nonterminal $\langle A,\mathcal{F}_A\rangle$ is supposed to be replaced with a production

$$\langle B,\mathcal{F}_B\rangle \longrightarrow \beta_1\langle\!\langle A\beta_2',\mathcal{F}_{A\beta_2'}\rangle\!\rangle\beta_2''$$

where $\beta_2 = \beta_2'\beta_2''$ and $\mathcal{F}_{A\beta_2'} = \mathrm{FIRST}_k^{\bar{G}}(\beta_2''\mathcal{F}_B)$. The new symbol $\langle\!\langle A\beta_2',\mathcal{F}_{A\beta_2'}\rangle\!\rangle \notin \bar{N}$ acts as a trigger for the embedded $\mathrm{LR}(k)$ parser for $A\beta_2'$ capable of termination on any string from $\mathcal{F}_{A\beta_2'}$.

As the amount of LR parsing is to be minimal, β_2' should be as short as possible, i.e., ε in the best case. If, on the other hand, not even $\beta_2' = \beta_2$ and $\beta_2'' = \varepsilon$ suffices for the safe termination of the embedded $\mathrm{LR}(k)$ parser, $\langle B,\mathcal{F}_B\rangle$ must be declared a conflicting nonterminal.

Finally, if marker $\langle\!\langle\beta,\mathcal{F}\rangle\!\rangle$ is introduced into the grammar $\bar{G} = \langle\bar{N},T,\bar{P},\bar{S}\rangle$ (based on $G = \langle N,T,P,S\rangle$), an embedded $\mathrm{LR}(k)$ parser for β that terminates on any lookahead string $x \in \mathcal{F}$, is needed. The easiest way to achieve this is to build the $\mathrm{LR}(k)$ parser for the *embedded grammar*

$$\hat{G}_{\beta,\mathcal{F}} = \langle\hat{N},T,\hat{P},S_1\rangle$$

where $\hat{N} = N \cup \{S_1,S_2\}$ for $S_1,S_2 \notin N$ and

$$\hat{P} = P \cup \{S_1 \longrightarrow S_2x, S_2 \longrightarrow \beta\ ;\ x \in \mathcal{F}\} \quad .$$

The trick is obvious: the *embedded* $\mathrm{LR}(k)$ *parser for* $\hat{G}_{\beta,\mathcal{F}}$ must accept its input no later than when the reduction on $S_2 \longrightarrow \beta$ is due. In other words, if the reduce on $S_2 \longrightarrow \beta$ is replaced with the accept action, the parser never pushes any symbol of any string $x \in \mathcal{F}$ onto the stack. If the reduce on $S_2 \longrightarrow \beta$ cannot be determined (because of the $\mathrm{LR}(k)$ conflict), the embedded $\mathrm{LR}(k)$ parser for $\langle\!\langle\beta,\mathcal{F}\rangle\!\rangle$ cannot be used.

Determining whether the embedded $\mathrm{LR}(k)$ parser does not contain any $\mathrm{LR}(k)$ conflicts is time consuming if a brute-force approach of using testing whether $\hat{G}_{\beta,\mathcal{F}} \in \mathrm{LR}(k)$ is used. However, the method based on the following theorem significantly reduces the time complexity of testing the embedded $\mathrm{LR}(k)$ parser for $\mathrm{LR}(k)$ conflicts.

Theorem 1. *Let $G = \langle N, T, P, S\rangle$ be an $\mathrm{LR}(k)$ grammar with the derivation*

$$S \Longrightarrow^*_{G,\mathrm{lm}} uB\delta \Longrightarrow_{G,\mathrm{lm}} u\beta_1\beta'_2\beta''_2\delta \quad .$$

Grammar $\hat{G} = \langle \hat{N}, T, \hat{P}, S_1\rangle$ where

$$\hat{N} = N \cup \{S_1, S_2\} \text{ for } S_1, S_2 \notin N \text{ and}$$
$$\hat{P} = P \cup \{S_1 \longrightarrow S_2x, S_2 \longrightarrow \beta'_2\ ;\ x \in \mathrm{FIRST}^G_k(\beta''_2\delta)\} \quad ,$$

is not an $\mathrm{LR}(k)$ grammar if and only if

- *either $\beta''_2 = \varepsilon$ and $[S \to \beta'_2\bullet, x'], [B \to \beta'_2\bullet, x'] \in [\$\beta'_2]_{\hat{G}}$*
- *or $\beta''_2 \neq \varepsilon$ and $[S_2 \to \beta'_2\bullet, x'], [A \to \alpha\bullet\alpha', y'] \in [\$\beta'_2]_{\hat{G}}$ where $\alpha' \neq \varepsilon$ and $x' \in \mathrm{FIRST}^G_k(\alpha'y')$.*

Proof. The idea the proof is based on is rather simple. Because of the leftmost derivation specified by this theorem, there is a state of the $\mathrm{LR}(k)$ machine for G that includes all $[B \to \beta_1\bullet\beta'_2\beta''_2, y]$ where $y \in \mathrm{FIRST}^G_k(\delta)$. This state corresponds to the initial state of the $\mathrm{LR}(k)$ machine for $\hat{G}$. By careful examination of all possibilities only those possibilities permitting $\mathrm{LR}(k)$ conflicts in $\hat{G}$ are singled out. The formal proof follows.

First, the structure of the grammar $\hat{G}$ implies that items

$$[S_1 \to \bullet S_2x, \$] \text{ and } [S_2 \to \bullet\beta'_2, x'] \quad ,$$

where $x \in \mathrm{FIRST}^G_k(\beta''_2\delta)$ and $x' \in \mathrm{FIRST}^G_k(\beta''_2\delta\$)$, appear only in the initial state $[\$]_{\hat{G}}$ of the canonical $\mathrm{LR}(k)$ parser for the (\$-augmented version of) grammar $\hat{G}$. Likewise, items

$$[S_1 \to \psi_1\bullet\psi_2, \$] \text{ and } [S_2 \to \psi_1\bullet\psi_2, x'] \quad ,$$

where $x' \in \mathrm{FIRST}^G_k(\beta''_2\delta\$)$, appear only in $[\$\psi_1]_{\hat{G}}$. Furthermore, states $[\$S_2\psi]_{\hat{G}}$, for various ψ, contain only items based on productions $S_1 \longrightarrow S_2x$.

Second, as $G \in \mathrm{LR}(k)$ and is thus unambiguous, the leftmost derivation

$$S \Longrightarrow^*_{G,\mathrm{lm}} uB\delta \Longrightarrow^*_{G,\mathrm{lm}} w$$

implies the existence of the rightmost derivation

$$S \Longrightarrow^*_{G,\mathrm{rm}} \gamma Bv'' \Longrightarrow_{G,\mathrm{rm}} \gamma\beta_1\beta'_2\beta''_2v'' \Longrightarrow^*_{G,\mathrm{rm}} w \quad .$$

Moreover, if $\delta \Longrightarrow^*_G v''$, then the viable prefix γ depends only on the left sentential form $uB\delta$, i.e., it is unique for all w. Therefore,

$$\{[B \to \beta_1\bullet\beta'_2\beta''_2, y'];\ y' \in \mathrm{FIRST}^G_k(\delta\$)\} \subseteq [\$\gamma\beta_1]_G$$

where $[\$\gamma\beta_1]_G$ is the state $[\$\gamma\beta_1]_G$ of the canonical $\mathrm{LR}(k)$ machine for the (\$-augmented version of) grammar G.

Consider any two items i_1 and i_2 (except items based on the production $S' \longrightarrow \$S_1\$$ as these items are never involved in an $\mathrm{LR}(k)$ conflict) in any state $[\$\hat{\gamma}]_{\hat{G}}$ of the canonical $\mathrm{LR}(k)$ machine for $\hat{G}$, i.e., $i_1, i_2 \in [\$\hat{\gamma}]_{\hat{G}}$:

1. If i_1 and i_2 are based on productions in P, then $i_1, i_2 \in [\$\gamma\beta_1\hat{\gamma}]_G$ and there is no $\mathrm{LR}(k)$ conflict between i_1 and i_2 since $G \in LR(k)$.
2. If i_1 and i_2 are based on productions in $\hat{P} \setminus P$, the following three cases must be considered:
   (a) $i_1 = [S_1 \to \hat{\gamma}\bullet\alpha, \$]$ and $i_2 = [S_1 \to \hat{\gamma}\bullet\alpha', \$]$:
   If $\hat{\gamma} = \varepsilon$, then i_1 and i_2 imply no actions because α and α' start with S_2. Otherwise they imply no reduce action (if $\alpha \neq \varepsilon$ and $\alpha' \neq \varepsilon$), imply the same action (as $i_1 = i_2$ if $\alpha = \varepsilon$ and $\alpha' = \varepsilon$), or imply the reduce on $\$$ and shift on non-$\$$ (if $\alpha = \varepsilon$ and $\alpha' \neq \varepsilon$; or vise versa).
   (b) $i_1 = [S_1 \to \hat{\gamma}\bullet\alpha, \$]$ and $i_2 = [S_2 \to \hat{\gamma}\bullet\alpha', y']$ (or vice-versa):
   i_1 implies no action if $\hat{\gamma} = \varepsilon$ as α starts with S_2. The other case, if $\hat{\gamma} \neq \varepsilon$, is impossible: $\hat{\gamma}$ starts with S_2 in i_1 and does not start with S_2 in i_2.
   (c) $i_1 = [S_2 \to \hat{\gamma}\bullet\alpha, y]$ and $i_2 = [S_2 \to \hat{\gamma}\bullet\alpha', y']$:
   $\alpha = \alpha'$ and both items imply either the same action or imply no action.
3. If i_1 is based on a production in $\hat{P} \setminus P$ and i_2 is based on a production in P (or vice versa), the following two cases must be considered:
   (a) $i_1 = [S_1 \to \hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, \$]$ and $i_2 = [A \to \hat{\gamma}_2\bullet\alpha', y']$:
   If $\hat{\gamma}_1\hat{\gamma}_2 = \varepsilon$, then i_1 implies no action as α starts with S_2. The other case, if $\hat{\gamma}_1\hat{\gamma}_2 \neq \varepsilon$, is impossible: $i_1 \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$ while $i_2 \notin [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$.
   (b) $i_1 = [S_2 \to \hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, y]$ and $i_2 = [A \to \hat{\gamma}_2\bullet\alpha', y']$:
   As $i_1, i_2 \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$, so does

   $$[B \to \beta_1\gamma_1\hat{\gamma}_2\bullet\alpha\beta_2'', y''], [A \to \hat{\gamma}_2\bullet\alpha', y'] \in [\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$$

   where $y \in \mathrm{FIRST}_k^G(\beta_2'' y_2'')$.
   - If $\alpha \neq \varepsilon$ and $\alpha' \neq \varepsilon$, then neither i_1 nor i_2 implies a reduce action.
   - If $\alpha \neq \varepsilon$ and $\alpha' = \varepsilon$, then

     $$[S_2 \to \hat{\gamma}_1\hat{\gamma}_2\bullet\alpha, y], [A \to \hat{\gamma}_2\bullet, y'] \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$$

     exhibit a shift-reduce conflict if and only if $y' \in \mathrm{FIRST}_k^G(\alpha y)$. But then items

     $$[B \to \beta_1\hat{\gamma}_1\hat{\gamma}_2\bullet\alpha\beta_2'', y''], [A \to \hat{\gamma}_2\bullet, y'] \in [\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$$

     exhibit a conflict. This is not possible as $G \in \mathrm{LR}(k)$ and therefore items i_1 and i_2 do not exhibit a conflict in $\hat{G}$.
   - If $\alpha = \varepsilon$ and $\alpha' \neq \varepsilon$, then

     $$[S_2 \to \hat{\gamma}_1\hat{\gamma}_2\bullet, y], [A \to \hat{\gamma}_2\bullet\alpha', y'] \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$$

     exhibit a shift-reduce conflict if $y \in \mathrm{FIRST}_k^G(\alpha' y')$. But

     $$[B \to \beta_1\hat{\gamma}_1\hat{\gamma}_2\bullet\beta_2'', y''], [A \to \hat{\gamma}_2\bullet\alpha', y'] \in [\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$$

     and the only possibility of a shift-reduce conflict in $[\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$ without the conflict in $[\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$ is that $\beta_2' = \hat{\gamma}_1\hat{\gamma}_2$ and $\beta_2'' \neq \varepsilon$.

– If $\alpha = \varepsilon$ and $\alpha' = \varepsilon$, then

$$[S_2 \to \hat{\gamma}_1\hat{\gamma}_2\bullet, y], [A \to \hat{\gamma}_2\bullet, y'] \in [\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$$

exhibit a reduce-reduce conflict if $y = y'$. But

$$[B \to \beta_1\hat{\gamma}_1\hat{\gamma}_2\bullet\beta_2'', y''], [A \to \hat{\gamma}_2\bullet, y] \in [\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$$

and the only possibility of a reduce-reduce conflict in $[\$\hat{\gamma}_1\hat{\gamma}_2]_{\hat{G}}$ without the conflict in $[\$\gamma\beta_1\hat{\gamma}_1\hat{\gamma}_2]_G$ is that $\beta_1\gamma_1 = \varepsilon$, $\beta_2' = \hat{\gamma}_2$ and $\beta_2'' = \varepsilon$.

Finally, proving the theorem in the opposite direction is trivial — if the canonical $\mathrm{LR}(k)$ machine for the grammar $\hat{G}$ contains an $\mathrm{LR}(k)$ conflict, then clearly $\hat{G} \notin \mathrm{LR}(k)$. ■

Corollary 1. *Let $G = \langle N, T, P, S\rangle$ be an $\mathrm{LR}(k)$ grammar with the derivation*

$$S \Longrightarrow^*_{G,\mathrm{lm}} uB\delta \Longrightarrow_{G,\mathrm{lm}} u\beta_1\beta_2'\delta \quad .$$

Grammar $\hat{G} = \langle \hat{N}, T, \hat{P}, S_1\rangle$ where $\hat{N} = N \cup \{S_1, S_2\}$ for $S_1, S_2 \notin N$ and $\hat{P} = P \cup \{S_1 \longrightarrow S_2x, S_2 \longrightarrow \beta_2' \,;\, x \in \mathrm{FIRST}^G_k(\delta)\}$ is not an $\mathrm{LR}(k)$ grammar if and only if

$$[S_2 \to \bullet\beta_2', x'] \;\mathbf{desc}^*\; [B \to \bullet\beta_2', x']$$

where $B \neq S_2$ and $x' \in \mathrm{FIRST}^G_k(\delta\$)$ [18]. ■

To conclude this section, Algorithm 1 is given. It is based on Theorem 1 and is (to be) used for computing the shortest prefix of $\langle A, \mathcal{F}_A\rangle\beta_2$ in production

$$\langle B, \mathcal{F}_B\rangle \longrightarrow \beta_1\langle A, \mathcal{F}_A\rangle\beta_2$$

where the embedded $\mathrm{LR}(k)$ parser must be employed to resolve the $\mathrm{LL}(k)$ conflict caused by $\langle A, \mathcal{F}_A\rangle$. Once Theorem 1 is digested, the algorithm comes out relatively simple: it just checks both conditions exposed by Theorem 1, one for $\beta_2'' = \varepsilon$ and the other for $\beta_2'' \neq \varepsilon$.

4. Terminating while producing the left parse

As mentioned in Section 2, the embedded $\mathrm{LR}(k)$ parser must produce the left parse instead of the right parse. To achieve this, the left $\mathrm{LR}(k)$ parser [20] (based on the Schmeiser-Barnard $\mathrm{LR}(k)$ parser [13]) is taken as the starting point.

Consider an $\mathrm{LR}(k)$ grammar $G = \langle N, T, P, S\rangle$ and the input string $w = uv$ derived by the rightmost derivation

$$S \Longrightarrow^*_{G,\mathrm{rm}} \gamma v \Longrightarrow^*_{G,\mathrm{rm}} uv \quad . \tag{5}$$

After reading the prefix u, the canonical $\mathrm{LR}(k)$ parser for grammar G reaches the configuration

$$\$[\$][\$X_1][\$X_1X_2]\ldots[\$X_1X_2\ldots X_n] \,|\, v\$ \tag{6}$$

Algorithm 1 Computing the shortest prefix β' of the sentential form $\beta = \beta'\beta''$ so that the embedded $\mathrm{LR}(k)$ grammar $\hat{G}_{\beta',\mathcal{F}'} \in \mathrm{LR}(k)$ where $\mathcal{F}' = \mathrm{FIRST}_k^G(\beta''\mathcal{F})$.
INPUT: The sentential form $\beta = X_1X_2\ldots X_n$ and the right context $\mathcal{F}$.
OUTPUT: The prefix β' (or $\perp$ if the prefix does not exist).

1: **for** $i \leftarrow 1\ldots(n-1)$ **do**
2: $\quad\beta' = X_1X_2\ldots X_i$ and $\beta'' = X_{i+1}X_{i+2}\ldots X_n$
3: $\quad$**if** $\neg(\exists[A\to\alpha\bullet\alpha', y'] \in [\$\beta']_{\hat{G}} : \mathrm{FIRST}_k^{\hat{G}}(\alpha'y') \cap \mathrm{FIRST}_k^{\hat{G}}(\beta''\mathcal{F}\$) \neq \emptyset)$ **then**
4: $\quad\quad$**return** β'
5: $\quad$**end if**
6: **end for**
7: $\beta' = X_1X_2\ldots X_n$ and $\beta'' = \varepsilon$
8: **if** $\neg(\exists[A\to\alpha\bullet, x'] \in [\$\beta']_{\hat{G}} : x' \cap \mathrm{FIRST}_k^{\hat{G}}(\mathcal{F}\$) \neq \emptyset)$ **then**
9: $\quad$**return** β'
10: **end if**
11: **return** $\perp$

where $X_1X_2\ldots X_n = \gamma$, $[\$X_1X_2\ldots X_n]$ is the current parser state and $x = k{:}\,v\$$ is the contents of the lookahead buffer. ($[\$X_1X_2\ldots X_j]$, for $j = 0,1,\ldots,n$, denotes the state of the canonical $\mathrm{LR}(k)$ machine M_G reachable from the state $[\$]$ by string $X_1X_2\ldots X_j$ where M_G is based on the \$-augmented grammar G' obtained by adding the new start symbol S' with production $S' \longrightarrow \$S\$$ to G).

The Schmeiser-Barnard $\mathrm{LR}(k)$ parser augments each nonterminal pushed on the stack with the left parse of the substring derived from that nonterminal and thus reaches the configuration

$$\$\langle[\$];\varepsilon\rangle\langle[\$X_1];\pi(X_1)\rangle\langle[\$X_1X_2];\pi(X_2)\rangle\ldots\ldots\langle[\$X_1X_2\ldots X_n];\pi(X_n)\rangle \mathbin{|} v\$ \quad (7)$$

instead. $\pi(X_j)$ denotes the left parse of the substring derived from X_j and thus

$$X_1X_2\ldots X_n \Longrightarrow_{G,\mathrm{lm}}^{\pi(X_1)\pi(X_2)\ldots\pi(X_n)} u \quad .$$

To accumulate left parses on the stack, the actions are modified as follows:

– If the parser performs the shift action, no production is pushed on the stack, i.e., the terminal pushed is augmented with the empty left parse ε.
– If the parser performs the reduce action, the left parses accumulated in states removed from the stack are concatenated, and prefixed by the production the reduction is made on. The resulting left parse is pushed on the stack together with the new nonterminal.

Note that if this method is used, the first production of the left parse is produced only at the very end of parsing.

Example 3. Consider the embedded grammar G_{ex3} with productions

$$S_1 \longrightarrow S_2c\ , \quad S_2 \longrightarrow A\ , \quad A \longrightarrow aa\,|\,aB\,|\,bBa\,|\,bBaa\ , \quad B \longrightarrow Bb\,|\,\varepsilon \quad .$$

Table 1. Parsing the string $bbbaac \in L(G_{\mathrm{ex3}})$ using the Schmeiser-Barnard $\mathrm{LR}(1)$ parser.

	STACK	INPUT
1	$\$ \langle [\$]; \varepsilon \rangle$	$bbbaac\$$
2	$\$ \langle [\$]; \varepsilon \rangle \langle [\$b]; \varepsilon \rangle$	$bbaac\$$
3	$\$ \langle [\$]; \varepsilon \rangle \langle [\$b]; \varepsilon \rangle \langle [\$bB]; \pi_1 = B \to \varepsilon \rangle$	$bbaac\$$
4	$\$ \langle [\$]; \varepsilon \rangle \langle [\$b]; \varepsilon \rangle \langle [\$bB]; \pi_1 = B \to \varepsilon \rangle \langle [\$bBb]; \varepsilon \rangle$	$baac\$$
5	$\$ \langle [\$]; \varepsilon \rangle \langle [\$b]; \varepsilon \rangle \langle [\$bB]; \pi_2 = B \to Bb \cdot \pi_1 \rangle \langle [\$bBb]; \varepsilon \rangle$	$aac\$$
6	$\$ \langle [\$]; \varepsilon \rangle \langle [\$b]; \varepsilon \rangle \langle [\$bB]; \pi_3 = B \to Bb \cdot \pi_2 \rangle$	$aac\$$
7	$\$ \langle [\$]; \varepsilon \rangle \langle [\$b]; \varepsilon \rangle \langle [\$bB]; \pi_3 = B \to Bb \cdot \pi_2 \rangle \langle [\$bBa]; \varepsilon \rangle$	$ac\$$
8	$\$ \ \ldots \ \langle [\$bB]; \pi_3 = B \to Bb \cdot \pi_2 \rangle \langle [\$bBa]; \varepsilon \rangle \langle [\$bBaa]; \varepsilon \rangle$	$c\$$
9	$\$ \langle [\$]; \varepsilon \rangle \langle [\$A]; \pi_4 = A \to bBaa \cdot \pi_3 \rangle$	$c\$$
10	$\$ \langle [\$]; \varepsilon \rangle \langle [\$S_2]; \pi_5 = S_2 \to A \cdot \pi_4 \rangle$	$c\$$
11	$\$ \langle [\$]; \varepsilon \rangle \langle [\$S_2]; \pi_6 = S_2 \to A \cdot \pi_5 \rangle \langle [\$S_2c], \varepsilon \rangle$	$\$$
12	$\$ \langle [\$]; \varepsilon \rangle \langle [\$S_1]; \pi_7 = S_1 \to S_2c \cdot \pi_6 \rangle$	$\$$

where $\pi_7 = S_1 \to S_2c \cdot S_2 \to A \cdot A \to bBaa \cdot B \to Bb \cdot B \to Bb \cdot B \to \varepsilon$

Parsing of the input string $bbbaac$ using the Schmeiser-Barnard $\mathrm{LR}(1)$ parser is shown in Table 1. Note that the first production of the resulting left parse, namely $S_1 \longrightarrow S_2c$, is not known until the end of parsing. ■

The left $\mathrm{LR}(k)$ parser [20] is able to compute the prefix of the left parse of the substring corresponding to the prefix of the input string read so far during parsing (although this is not possible in every parser configuration). In other words, if corresponding to the derivation (5) the input string $w = uv$ is derived by the leftmost derivation

$$S \Longrightarrow^{\pi(u)}_{G,\mathrm{lm}} u\delta \Longrightarrow^{*}_{G,\mathrm{lm}} uv \quad , \tag{8}$$

then the left $\mathrm{LR}(k)$ parser can compute the left parse $\pi(u)$ in configuration (7) provided that certain conditions specified later on are met. As this part of the left $\mathrm{LR}(k)$ parser is modified, it deserves more attention.

By theory [17], configurations (6) and (7) imply that machine M_G contains at least one sequence of valid k-items

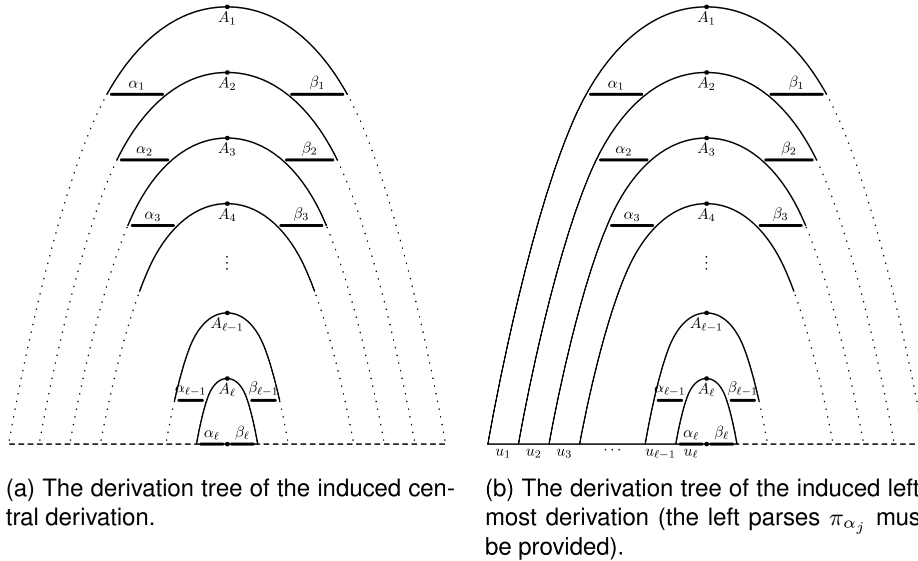
$$\begin{array}{c} [A_0 \to \bullet\alpha_0 A_1 \beta_0, x_0] \cdot \ldots \cdot [A_0 \to \alpha_0 \bullet A_1 \beta_0, x_0] \cdot \\ \cdot [A_1 \to \bullet\alpha_1 A_2 \beta_1, x_1] \cdot \ldots \cdot [A_1 \to \alpha_1 \bullet A_2 \beta_1, x_1] \cdot \\ \vdots \\ \cdot [A_\ell \to \bullet\alpha_\ell A_{\ell+1} \beta_\ell, x_\ell] \cdot \ldots \cdot [A_\ell \to \alpha_\ell \bullet A_{\ell+1} \beta_\ell, x_\ell] \end{array} \tag{9}$$

where $[A_0 \to \bullet\alpha_0 A_1 \beta_0, x_0] = [S' \to \bullet\$S\$, \varepsilon]$, $\gamma = \alpha_0\alpha_1 \ldots \alpha_\ell$ and $k{:}\, v\$ \in \mathrm{FIRST}_k^{G'}(A_{\ell+1}\beta_\ell x_\ell)$ (and $A_{\ell+1} = \varepsilon$); the horizontal dots denote repetitive application of operation $\mathbf{passes}$ (or GOTO) while the vertical dots denote the application of $\mathbf{desc}$ (or $\mathrm{CLOSURE}$).

Sequence (9) induces the *(induced) central derivation*

$$S' = A_0 \Longrightarrow_G \alpha_0 A_1 \beta_0 \Longrightarrow_G \alpha_0 \alpha_1 A_2 \beta_1 \beta_0 \Longrightarrow_G \dots \Longrightarrow_G$$
$$\Longrightarrow_G \alpha_0 \alpha_1 \dots \alpha_\ell A_{\ell+1} \beta_\ell \beta_{\ell-1} \dots \beta_0 \quad ;$$

the name "central" becomes obvious if the corresponding derivation tree presented in Figure 1(a) is observed.



(a) The derivation tree of the induced central derivation.

(b) The derivation tree of the induced leftmost derivation (the left parses π_{α_j} must be provided).

Fig. 1. The derivation trees corresponding to various kinds of induced derivations; remember that $A_{\ell+1} = \varepsilon$ in all three cases.

However, if the left parses $\pi(\alpha_0), \pi(\alpha_1), \dots, \pi(\alpha_\ell)$, where $\alpha_j \Longrightarrow^{\pi(\alpha_j)}_{G',\mathrm{lm}} u_j$ for $j = 0, 1, \dots, \ell$, are provided, sequence (9) induces the *(induced) leftmost derivation*

$$\begin{aligned} S' = A_0 &\Longrightarrow_{G,\mathrm{lm}} \alpha_0 A_1 \beta_0 \Longrightarrow^{\pi(\alpha_0)}_{G,\mathrm{lm}} u_0 A_1 \beta_0 \\ &\Longrightarrow_{G,\mathrm{lm}} u_0 \alpha_1 A_2 \beta_1 \beta_0 \Longrightarrow^{\pi(\alpha_1)}_{G,\mathrm{lm}} u_0 u_1 A_2 \beta_1 \beta_0 \\ &\;\vdots \\ &\Longrightarrow_{G,\mathrm{lm}} u_0 u_1 \dots u_{\ell-1} \alpha_\ell A_{\ell+1} \beta_\ell \beta_{\ell-1} \dots \beta_0 \\ &\qquad \Longrightarrow^{\pi(\alpha_\ell)}_{G,\mathrm{lm}} u_0 u_1 \dots u_\ell A_{\ell+1} \beta_\ell \beta_{\ell-1} \dots \beta_0 \end{aligned}$$

where $u = u_0 u_1 \dots u_\ell$ and $k{:}\, v\$ \in \mathrm{FIRST}^{G'}_k(\beta_\ell \beta_{\ell-1} \dots \beta_0 \$)$. The corresponding derivation tree is shown in Figure 1(b) and the left parse of the induced leftmost

derivation is therefore

$$\pi(u) = A_0 \longrightarrow \alpha_0 A_1 \beta_0 \cdot \pi(\alpha_0) \cdot A_1 \longrightarrow \alpha_1 A_2 \beta_1 \cdot \pi(\alpha_1) \cdot \quad \ldots \quad \cdot \\ \cdot A_\ell \longrightarrow \alpha_\ell A_{\ell+1} \beta_\ell] \cdot \pi(\alpha_\ell) \quad . \tag{10}$$

(Likewise, if the right parses $\pi(\beta_1), \pi(\beta_2), \ldots, \pi(\beta_\ell)$ are known, then sequence (9) induces the *(induced) rightmost derivation*.)

Subparses $\pi(\alpha_j)$ of the left parse (10) are available on the parser stack because $\alpha_0\alpha_1 \ldots \alpha_\ell = \gamma = X_1 X_2 \ldots X_n$, but productions $A_j \longrightarrow \alpha_j A_{j+1} \beta_j$ are not. However, if sequence (9) is known, the missing productions and in fact the entire prefix of the left parse can be computed [20]. Starting with $\pi = \varepsilon$ and $i = [A_\ell \to \alpha_\ell \bullet A_{\ell+1} \beta_\ell, x_\ell]$, the stack is traversed downwards:

– If $i = [A \to \bullet\beta, x]$, then (a) i expands the nonterminal A by production $A \longrightarrow \beta$ and (b) i', the item that precedes i in sequence (9), is in the same state. Hence, let $\pi := A \longrightarrow \beta \cdot \pi$ and $i := i'$.
– If $i = [A \to \alpha X \bullet \beta, x] \in [\$\gamma X]$ for some γ, then (a) the left parse $\pi(X)$ is available on the stack and (b) i' is in the state $[\$\gamma]$ (which is found beneath $[\$\gamma X]$). Hence, let $\pi := \pi(X) \cdot \pi$ and $i := i'$; furthermore, proceed one step downwards along the stack, i.e., to the state $[\$\gamma]$.

The downward traversal stops when the item $[S_2 \to \bullet\beta, x] \in [\$]$, for some $\beta \in (N \cup T)^*$ and $x \in (T \cup \{\$\})^{*k}$, is reached (the production $S_2 \longrightarrow \beta$ is not added to the resulting left parse).

This method can be upgraded to compute the prefix of the left parse and the viable suffix δ^R in derivation (8) as well since $\delta = A_{\ell+1}\beta_\ell\beta_{\ell-1} \ldots \beta_0$ — see Figure 1(b). Hence, start with $\delta = A_{\ell+1}\beta_\ell$ and whenever $i = [A \to \bullet\beta, x]$, let $\delta := \delta \cdot \beta'$ where $i' = [A' \to \alpha' \bullet A\beta', x']$ is the item preceding i in sequence (9).

Example 4. Consider again the grammar G_{ex3} and the input string $bbbaac \in L(G_{\text{ex3}})$ from Example 3. After the prefix $bbba$ of the input string has been read, the parser reaches the configuration shown in the 7th line of Table 1. But as illustrated in Figure 2, there is only one item active for the current lookahead string a in state $[\$bBa]$, namely $[A \to bBa\bullet a, \$]$. Furthermore, there exist exactly one sequence of $\mathrm{LR}(1)$ items starting with $[S' \to \bullet\$S_1\$, \varepsilon] \in [\varepsilon]$ and ending with $[A \to bBa\bullet a, \$] \in [\$bS_2a]$:

$$[S' \to \bullet\$S_1\$, \varepsilon] \cdot [S' \to \$\bullet S_1\$, \varepsilon] \cdot [S_1 \to \bullet S_2 c, \$] \cdot [S_2 \to \bullet A, c] \cdot \\ \cdot [A \to \bullet bBaa, \$] \cdot [A \to b\bullet Baa, \$] \cdot \ \ldots \cdot [A \to bB\bullet aa, \$] \cdot [A \to bBa\bullet a, \$]$$

Hence, the prefix of the left parse and the corresponding viable suffix can be computed as shown in Figure 3 using the method outlined above. ■

In general, cases where exactly one sequence (9) exists (as in Example 4) are extremely rare, but all sequences (9) that differ only in lookahead strings x_j, where $j = 1, 2, \ldots, \ell$, induce the same (leftmost) derivation. In other words, the lookahead strings x_j are not needed for computing the prefix of the left parse and the viable suffix.

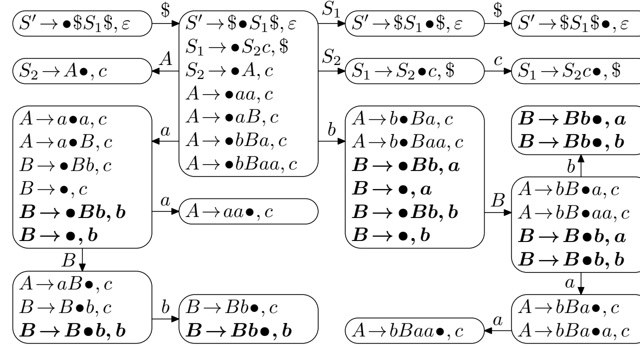


Fig. 2. The canonical $\mathrm{LR}(1)$ machine for G_{ex3} — items that end multiple sequences starting with $[S' \to \bullet\$S\$, \varepsilon] \in [\varepsilon]$ are shown in bold face.

The left $\mathrm{LR}(k)$ parser uses an additional parsing table called LEFT to establish whether the prefix of the left parse can be computed in some state $[\$\gamma]$ for some lookahead string x, and the *left-parse-prefix* automaton (LPP) to actually compute sequence (9) with the lookahead strings omitted.

The LEFT table implements mapping

$$\text{LEFT: } Q_k^G \times (T \cup \{\$\})^{*k} \longrightarrow (I_0^G \cup \{\bot\})$$

where Q_k^G and I_0^G denote the set of $\mathrm{LR}(k)$ states and the set of $\mathrm{LR}(0)$ items for grammar G', respectively. It maps $\mathrm{LR}(k)$ state $[\$\gamma]$ and the contents x of the lookahead buffer to either

– $[A_\ell \to \alpha_\ell \bullet A_{\ell+1}\beta_\ell]$, where $\alpha_\ell \neq \varepsilon$, if all sequences (9) that are active for x, i.e., they end with some some $\mathrm{LR}(k)$ item $[A_\ell \to \alpha_\ell \bullet A_{\ell+1}\beta_\ell, x_\ell]$ (for different x_ℓ) where $x \in \mathrm{FIRST}_k^{G'}(A_{\ell+1}\beta_\ell x_\ell)$, differ in lookahead strings only, or
– $\bot$ otherwise.

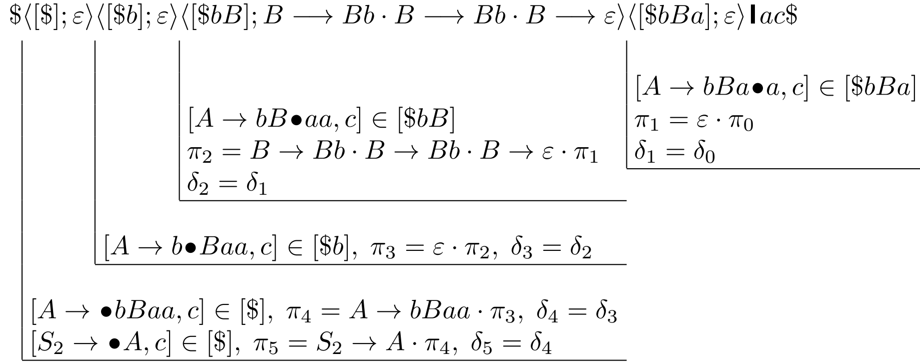
Hence, the parser can produce the prefix of the left parse and compute the viable suffix if and only if $\text{LEFT}([\$\gamma], x) \neq \bot$.

The above definition of LEFT works well for the left $\mathrm{LR}(k)$ parser [20]. But as

$$[\$] = \mathbf{desc}^*(\{[S' \to \$\bullet S_1\$, \varepsilon]\})$$

(note that the embedded grammar is being used) and there is only one path to $\{[S' \to \$\bullet S_1\$, \varepsilon]\} \in [\$]$, the value of $\text{LEFT}([\$], x)$ is set to $[S' \to \$\bullet S_1\$]$ for all $x \in \mathrm{FIRST}_k^{G'}(S_1\$)$ if the definition suitable for the let $\mathrm{LR}(k)$ parser is used. It is valid but useless because if the method outlined in Example 4 is used, the embedded left $\mathrm{LR}(k)$ parser would print ε and stop before ever producing any production of the left parse.

Thus, an exception must be made in state $[\$]$. Provided that the grammar includes the productions $S_1 \longrightarrow S_2 y$ and $S_2 \longrightarrow A\beta$, the value of $\text{LEFT}([\$], x)$ must be set to either



The result: $\pi=S_2\to A\cdot A\to bBaa\cdot B\to Bb\cdot B\to Bb\cdot B\to\varepsilon$ and $\delta=a$

Fig. 3. Computing the prefix of the left parse of the string $bbbaac\in L(G_{\mathrm{ex3}})$ and the corresponding viable suffix after $bbba$ has been read: the computation starts at the top of the stack (right side of the figure) with $\pi_0=\varepsilon$ and $\delta_0=a$, and traverses the stack downwards (towards the left side of the figure, and then downwards).

– $[A_\ell\to\ \bullet A_{\ell+1}\beta_\ell]$ if all sequences (9) that are active for x, i.e., they end with some some $\mathrm{LR}(k)$ item $[A_\ell\to\ \bullet A_{\ell+1}\beta_\ell,x_\ell]$ (for different x_ℓ) where $x\in\mathrm{FIRST}_k^{G'}(A_{\ell+1}\beta_\ell x_\ell)$, differ in lookahead strings only and

$$[S_2\to\bullet A_\ell\beta,y]\ \mathbf{desc}\ [A_\ell\to\bullet A_{\ell+1}\beta_\ell,x_\ell]\quad,$$

or

– $\perp$ otherwise.

The left-parse-prefix automaton represents mapping

$$\text{LPP: } I_0^G\times Q_k^G\longrightarrow I_0^G$$

which is a compact representation of all possible sequences (9) with lookahead strings stripped off. Hence, $\text{LPP}(i_0,[\$\gamma])=i_0'$ if and only if there exists some sequence (9) with two consecutive $\mathrm{LR}(k)$ items i_k',i_k, where $i_k\in[\$\gamma]$, so that i_0 (i_0') is equal to i_k (i_k') without the lookahead string.

Example 5. The left-parse-prefix automaton for the grammar G_{ex3} is shown in Figure 4. (In this example, the left-parse-prefix automaton is trivial, i.e., without any loop, but if the grammar is bigger and describes a more complex language, the corresponding LPP gets more complicated — see [20].)

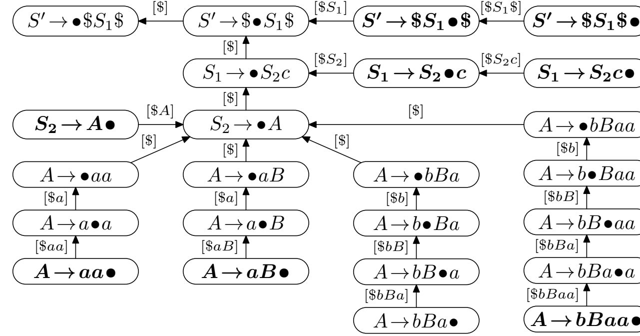


Fig. 4. The left-parse-prefix automaton for G_{ex3} — items that are not needed during embedded left $\text{LR}(1)$ parsing are shown in bold face.

Mapping LEFT for G_{ex3} is defined as

$$\begin{aligned}
\text{LEFT}([\$S_2], c) &= [S_2 \to A\bullet c] \\
\text{LEFT}([\$a], a) &= [A \to a\bullet a] \\
\text{LEFT}([\$a], b) &= [A \to a\bullet B] \\
\text{LEFT}([\$bBa], \$) &= [A \to bBa\bullet] \\
\text{LEFT}([\$bBa], b) &= [A \to bBa\bullet a]
\end{aligned}$$

(in all other cases, the value of LEFT equals $\bot$). Note that $\text{LEFT}([\$], a) = \bot$ and $\text{LEFT}([\$], b) = \bot$ because of $A \longrightarrow aa|aB$ and $A \longrightarrow bBa|bBaa$, respectively. ■

The algorithms for computing LEFT and LPP can be found in [20]. Once mappings LEFT and LPP are available, the method for computing the prefix of the left parse and the viable suffix as outlined above and illustrated by Example 4 can be formalized as Algorithm 2. It is basically an algorithm which performs a *long reduction*: a sequence of reductions on productions whose right sides have been only partially pushed on the stack.

Algorithm 2 Computing the prefix of the left parse and the viable suffix.
INPUT: Stack contents of the left $\text{LR}(k)$ parser and a state of LPP automaton.
OUTPUT:The prefix of the left parse and the corresponding viable suffix.

long-reduction $(\Gamma, [A \to \alpha\bullet\beta]) = \langle \pi, \beta \cdot \delta \rangle$ where
  $\langle \pi, \delta \rangle =$ *long-reduction′* $(\Gamma, [A \to \alpha\bullet\beta])$

  long-reduction′ $(\Gamma, [S' \to \$\bullet S\$]) = \langle \varepsilon, \varepsilon \rangle$

  long-reduction′ $(\Gamma \cdot \langle [\$\gamma X], \pi(X) \rangle, [A \to \bullet\beta]) = \langle A \longrightarrow \beta \cdot \pi, \delta \cdot \beta' \rangle$
    where $[A' \to \alpha' \bullet A\beta'] = \text{LPP}([A \to \bullet\beta], [\$\gamma X])$
      $\langle \pi, \delta \rangle =$ *long-reduction′* $(\Gamma \cdot \langle [\$\gamma X], \pi(X) \rangle, [A' \to \alpha' \bullet A\beta'])$

  long-reduction′ $(\Gamma \cdot \langle [\$\gamma X'], \pi(X') \rangle \cdot \langle [\$\gamma X' X], \pi(X) \rangle, [A \to \alpha\bullet\beta]) = \langle \pi(X) \cdot \pi, \delta \rangle$
    where $\langle \pi, \delta \rangle =$ *long-reduction′* $(\Gamma \cdot \langle [\$\gamma X'], \pi(X') \rangle, \text{LPP}([A \to \alpha\bullet\beta], [\$\gamma X]))$

Algorithm 3 Embedded left $\mathrm{LR}(k)$ parsing.

1: let $q \in Q_k^G$ denote the topmost state
2: let $x \in (T \cup \{\$\})^{*k}$ denote the LA buffer contents
3: **while** $(i \leftarrow \text{LEFT}(q, x)) = \bot$ **do**
4: perform a step of the Schmeiser-Barnard $\mathrm{LR}(k)$ parser
5: **end while**
6: $\langle \pi, \delta \rangle \leftarrow$ *long-reduction* (stack, i)
7: PRINT π
8: **return** δ

If compared with the similar method used by the left $\mathrm{LR}(k)$ parser [20], this one is not only augmented to compute the viable suffix but also simplified in that it does not leave any markers on the stack about which subparses accumulated on the stack have already been printed out. It does not need to do this as after the first long reduction the LR parsing stops, the LR stack is cleared, and the control is given back to the backbone $\mathrm{LL}(k)$ parser.

Finally, for the sake of completeness, the sketch of the embedded left $\mathrm{LR}(k)$ parser is given as Algorithm 3: in essence, it is a Schmeiser-Barnard $\mathrm{LR}(k)$ parser [13] with the option of (a) premature termination and (b) computing the viable suffix.

Algorithm 3 always terminates: if not sooner (including cases where it detects a syntax error), the parser eventually reaches the (final) state $[\$S_2] = \{[S_1 \to S_2 \bullet x, \$]\}$ where $\text{LEFT}([\$S_2], \$) = [S_1 \to S_2 \bullet x]$ causing it to exit the loop in lines 3–5.

5. The embedded left $\mathrm{LR}(k)$ parser

The *embedded left* $\mathrm{LR}(k)$ *parser* is the left $\mathrm{LR}(k)$ parser for the embedded grammar (with a modified mapping LEFT) which (a) produces the left parse of the substring parsed and the remaining viable suffix, and (b) terminates after the first (simplified) long reduction.

Below, the first theorem establishes that the combination of $\mathrm{LL}(k)$ parsing and $\mathrm{LR}(k)$ parsing is asymptotically as fast as $\mathrm{LR}(k)$ parsing, and the second states that it is just as powerful as $\mathrm{LR}(k)$ parsing.

Theorem 2. *A backbone* $\mathrm{LL}(k)$ *parser augmented with embedded left* $\mathrm{LR}(k)$ *parsers can parse the input string* w *derived by the derivation* $S \Longrightarrow^{\pi} w$ *in time* $\mathcal{O}(|w|) + \mathcal{O}(|\pi|)$.

Proof. Each symbol of w is shifted only once, either by the backbone $\mathrm{LL}(k)$ parser or one of the embedded left $\mathrm{LR}(k)$ parsers, hence the $\mathcal{O}(|w|)$ part.

Each production in π is either produced by the backbone $\mathrm{LL}(k)$ parser or reduced upon by one of the embedded left $\mathrm{LR}(k)$ parsers. There are two different kinds of reductions: reductions performed during the long reduction require

time $k_1|\alpha|$ and ordinary "left" reductions require time $k_2|\alpha|$ for a reduction on $A \longrightarrow \alpha$ (but $|\alpha|$ is bounded by a constant depending on the grammar only). Hence the $\mathcal{O}(|\pi|)$ part. ∎

Theorem 3. *A backbone* $\mathrm{LL}(k)$ *parser augmented with embedded left* $\mathrm{LR}(k)$ *parsers can parse any deterministic context-free language.*

Proof. If L is DCFL, then there exists an $\mathrm{LR}(k)$ grammar G so that $L(G) = L$. For each $\mathrm{LL}(k)$-conflicting nonterminal A of $\bar{G}$ (the "$\mathrm{SLL}(k)$" variant of G)

- either an embedded left $\mathrm{LR}(k)$ parser can be constructed
- or a nonterminal on the left side of the production where A appear on the right side can be declared $\mathrm{LL}(k)$-conflicting nonterminal.

By repeatedly applying this trick all $\mathrm{LL}(k)$ conflicts get resolved — if not otherwise, when the initial symbol of $\bar{G}$ is declared to be an $\mathrm{LL}(k)$-conflicting symbol (note that the embedded left $\mathrm{LR}(k)$ parser for G with the terminating set $\{\$\}$ can always be constructed). ∎

It must be admitted that Theorem 3 should be taken with a grain of salt. While its proof is technically correct, it exposes the true nature of resolving $\mathrm{LL}(k)$ conflicts with embedded left $\mathrm{LR}(k)$ parsers. Namely, if embedded left $\mathrm{LR}(k)$ parsers are triggered for $\mathrm{LL}(k)$ conflicting nonterminals deriving relatively short substrings, then employing embedded left $\mathrm{LR}(k)$ parsers makes sense as the amount of a hidden bottom-up parsing is kept within some reasonable limits. Otherwise, if the grammar requires that an embedded left $\mathrm{LR}(k)$ parser is triggered relatively close to the root of the derivation tree, then a large part of the input string is going to be parsed by the embedded $\mathrm{LR}(k)$ parser and the method loses much of its appeal (to the point that perhaps the left $\mathrm{LR}(k)$ parser is more suitable [20]).

6. Conclusion

The embedded left $\mathrm{LR}(k)$ parser has been obtained by modifying the left $\mathrm{LR}(k)$ parser in two ways. First, the left $\mathrm{LR}(k)$ parser was made capable of computing the viable suffix which the unread part of the input string is derived from. Second, if was simplified not to leave any markers on the stack about which subparses accumulated on the stack have been printed out already — as the parser stops after the first "long" reduction anyway. However, the algorithm for minimizing the embedded left $\mathrm{LR}(k)$ parser, i.e., for removing states that are not reachable before the first long reduction is performed, is still to be formalized.

At present, both, the backbone LL parser and the embedded left LR parsers, need to use the lookahead buffer of the same length. However, if the LL parser was built around $\mathrm{LA}(k)\mathrm{LL}(\ell)$ parser (where $k \geq \ell$) as defined in [17], then the combined parsing could most probably be formulated as the combination of $\mathrm{LL}(\ell)$ and $\mathrm{LR}(k)$ parsing (note that $\mathrm{LL}(\ell) \subseteq \mathrm{LA}(\ell')\mathrm{LL}(\ell)$ for any $\ell' \geq \ell$). This would make the combined parser even more memory efficient.

The left $\mathrm{LR}(k)$ parser could be based on the $\mathrm{LA}(k)\mathrm{LR}(\ell)$ parser (most likely for $\ell = 0$) instead of on the canonical $\mathrm{LR}(k)$ parser. This would further reduce the parsing tables while the strength of the resulting combined parser would be reduced from $\mathrm{LR}(k)$ to $\mathrm{LA}(k)LR(\ell)$: not a significant issue as today $\mathrm{LA}(1)\mathrm{LR}(0)$ is used instead of $\mathrm{LR}(1)$ whenever LR parsing is applied.

By using an $\mathrm{LL}(k)$ parser augmented by the embedded left $\mathrm{LR}(k)$ parsers instead of the left $\mathrm{LR}(k)$ parser the error recovery can be made much better — especially if the error recovery of the embedded left $\mathrm{LR}(k)$ parsers is made using the method described in [19].

Finally, apart from using the embedded left $\mathrm{LR}(k)$ parser for $\mathrm{LL}(k)$ conflict resolution, the embedded left $\mathrm{LR}(k)$ parser can be a convenient method for parsing the embedded domain-specific languages [9]. Furthermore, the termination condition formulated in Section 3 can be considered as a guideline for designing an embedded domain-specific language which fits gently into the enclosing (usually general-purpose) programming language, i.e., without explicit markers denoting the border between the embedded and the enclosing language; the termination condition also provides an efficient automatic method for detecting any syntactic problems arising from the embedding itself.

References

1. Aycock, J., Horspool, N., Janoušek, J., Melichar, B.: Even faster generalized LR parsing. Acta Informatica 37(9), 633–651 (2001)
2. Boyland, J., Spiewak, D.: TOOL PAPER: ScalaBison recursive ascent-descent parser generator. Electronic Notes in Theoretical Computer Science 253(7), 65–74 (2010)
3. Demers, A.J.: Generalized left corner parsing. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages POPL'77. pp. 170–182. ACM, ACM, Los Angeles, CA, USA (1977)
4. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of the 31st ACM SIGACT-SIGPLAN symposium on Principles of programming languages POPL'04. pp. 111–122. ACM, ACM, Venice, Italy (2004)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading, MA, USA (1979)
6. Horspool, R.N.: Recursive ascent-descent parsers. In: Hammer, D. (ed.) Compiler Compilers, Third International Workshop CC '90, Schwerin, FRG, Lecture Notes in Computer Science, vol. 477, pp. 1–10. Springer-Verlag (1990)
7. Knuth, D.E.: On the translation of languages from left to right. Information and Control 8(6), 607–639 (1965)
8. Lewis II, P.M., Stearns, R.E.: Syntax directed transduction. In: Proceedings of the 7th Annual Symposion on Switching and Automata Theory (SWAT'66). pp. 21–35. IEEE Computer Society Press, Berkeley, CA, USA (1966)
9. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys 37(4), 316–344 (2005)
10. Might, M., Darais, D.: Yacc is dead. Available online at Cornell University Library (arXiv.org:1010.5023) (2010)
11. Parr, T., Fischer, K.: LL(*): The foundation of the ANTLR parser generator. ACM SIGPLAN Notices - PLDI'10 46(6), 425–436 (2011)

12. Rosenkrantz, D.J., Lewis, P.M.: Deterministic left corner parsing. In: Proceedings of the 11th Annual Symposium on Switching and Automata Theory (SWAT 1970). pp. 139–152. IEEE Computer Society, Washington, DC, USA (1970)
13. Schmeiser, J.P., Barnard, D.T.: Producing a top-down parse order with bottom-up parsing. Information Processing Letters 54(6), 323–326 (1995)
14. Scott, E., Johnstone, A.: GLL parsing. Electronic Notes in Theoretical Computer Science 253(7), 177–189 (2010)
15. Scott, E., Johnstone, A., Economopoulos, R.: BRNGLR: a cubic Tomita-style GLR parsing algorithm. Acta Informatica 44(6), 427–461 (2007)
16. Sippu, S., Soisalon-Soininen, E.: Parsing Theory, Volume I: Languages and Parsing, EATCS Monographs on Theoretical Computer Science, vol. 15. Springer-Verlag, Berlin, Germany (1988)
17. Sippu, S., Soisalon-Soininen, E.: Parsing Theory, Volume II: LR(k) and LL(k) Parsing, EATCS Monographs on Theoretical Computer Science, vol. 20. Springer-Verlag, Berlin, Germany (1990)
18. Slivnik, B.: The embedded left LR parser. In: Proceedings of the Federated Conference on Computer Science and Information Systems. pp. 871–878. IEEE Computer Society Press, Szczecin, Poland (2011)
19. Slivnik, B., Vilfan, B.: Improved error recovery in generated LR parsers. Informatica 28(3), 257–263 (2004)
20. Slivnik, B., Vilfan, B.: Producing the left parse during bottom-up parsing. Information Processing Letters 96(6), 220–224 (2005)
21. Tomita, M.: Efficient Parsing for Natural Language. Kluwer Academic Publisher, Boston, MA, USA (1985)
22. Tomita, M. (ed.): Generalized LR Parsing. Springer-Verlag, Berlin, Germany (1991)

Boštjan Slivnik received the M.Sc. and Ph.D. degrees in computer science from the University of Ljubljana in 1996 and 2003 respectively. He is currently at the University of Ljubljana, Faculty of Computer and Information Science. His research interests include parsing algorithms, compilers, formal languages, and distributed algorithms. He has been a member of the ACM since 1996.

Received: December 16, 2011; Accepted: April 2, 2012.

DOI: 10.2298/CSIS111220024T

Indexing Ordered Trees for (Nonlinear) Tree Pattern Matching by Pushdown Automata

Jan Trávníček, Jan Janoušek, and Borivoj Melichar

Faculty of Information Technology
Czech Technical University in Prague
Thákurova 9, 160 00 Prague 6, Czech Republic
{Jan.Travnicek, Jan.Janousek, melichar}@fit.cvut.cz

Abstract. Trees are one of the fundamental data structures used in Computer Science. We present a new kind of acyclic pushdown automata, the *tree pattern pushdown automaton* and the *nonlinear tree pattern pushdown automaton*, constructed for an ordered tree. These automata accept all tree patterns and nonlinear tree patterns, respectively, which match the tree and represent a full index of the tree for such patterns. Given a tree with n nodes, the numbers of these distinct tree patterns and nonlinear tree patterns can be at most $2^{n-1} + n$ and at most $(2+v)^{n-1} + 2$, respectively, where v is the maximal number of nonlinear variables allowed in nonlinear tree patterns. The total sizes of nondeterministic versions of the two pushdown automata are $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$, respectively. We discuss the time complexities and show timings of our implementations using the bit-parallelism technique. The timings show that for a given tree the running time is linear to the size of the input pattern.

Keywords: Tree pattern matching, nonlinear tree pattern matching, indexing trees, pushdown automata

1. Introduction

Trees are one of the fundamental data structures used in Computer Science. Finding occurrences of tree patterns in trees is an important problem with many applications such as compiler code selection, interpretation of nonprocedural languages, implementation of rewriting systems, or various tree finding and tree replacement systems. Tree patterns are trees in which leaves can be labelled also by a special linear variable S, which serves as a placeholder for any subtree. Nonlinear tree patterns can further contain leaves labelled by specific nonlinear variables, where each of nonlinear variables represents a specific subtree. Nonlinear tree pattern matching is used especially in the implementation of term rewriting systems, in which the terms can be represented as tree structures with nonlinear variables.

Generally, there exist two basic approaches to pattern matching problems. The first approach is represented by the use of a pattern matcher which is constructed for patterns. In other words, the patterns are preprocessed. Given a

tree of size n, such tree pattern matcher typically perform the search phase in time linear in n [15, 6, 12, 24]. This approach is suitable for cases when one wants look for occurrences of a given pattern in input subject structures. The second basic approach is represented by the use of an indexing data structure constructed for the subject in which we search. In other words, the subject is preprocessed. Examples of such indexing structures are suffix or factor automata [9, 10, 22, 25] in the area of string processing or subtree pushdown automaton [18], which represents a complete index of an ordered tree for subtrees. This approach is suitable especially for cases when one wants look for occurrences of different input patterns in a given subject structure.

The theory of formal tree languages have been extensively studied and developed since the 1960s and its main models of computation are various kinds of tree automata [14, 6, 8]. However, trees can also be represented as strings, for example in their prefix (also called preorder) or postfix (also called postorder) notation. A linear notation of a tree can be obtained by the corresponding traversing of the tree. Moreover, every sequential algorithm on a tree traverses nodes of the tree in a sequential order and so follows a linear notation of the tree. [20] proves that the deterministic pushdown automaton (PDA) is an appropriate model of computation for labelled ordered trees in linear notation and that the trees in postfix notation acceptable by deterministic PDA form a proper superclass of the class of regular tree languages [14], which are accepted by finite tree automata. Recently, pushdown automata gain a popularity in solving practical problems of processing trees, for example in processing XML documents [13].

In this paper we present a new kind of acyclic pushdown automata for an ordered tree. The *tree pattern pushdown automaton* and the *nonlinear tree pattern pushdown automaton* represent a complete index of the tree for tree patterns and nonlinear tree patterns, respectively, and accept all tree patterns and nonlinear tree patterns, respectively, which match the tree. Given a tree with n nodes, the numbers of distinct tree patterns and nonlinear tree patterns which match the tree can be at most $2^{n-1} + n$ and at most $(2 + v)^{n-1} + 2$, respectively, where v is the maximal number of distinct nonlinear variables allowed in nonlinear tree patterns. We describe the construction of nondeterministic (nonlinear) tree pattern pushdown automata and discuss their time and space complexities. We are not aware of any other existing pushdown automaton which would represent such an index. The presented nondeterministic pushdown automata are input–driven and therefore can be determinised.

The presented (nonlinear) tree pattern pushdown automata have two kinds of transitions. First, transitions reading symbols of the alphabet of labels of tree nodes. Second, transitions reading the variables in (nonlinear) tree patterns. If our pushdown automata have only the former transitions, they would be analogous to string nondeterministic suffix or factor automata. Efficient methods of implementing nondeterministic string suffix automata by the bit–parallelism technique are well-known [23]. The bit–parallelism technique can be also used for the latter transitions efficiently. We describe our implementations using the

bit-parallel technique and show their timings. The timings show that the running time is for a given tree linear to the size of the input (nonlinear) tree patterns, which is a result similar to the result described in [23].

We note that the presented PDAs have only one pushdown symbol and therefore can be easily transformed to counter automata, which are a weaker and simpler model of computation than the PDA. We present the automata in this paper as PDAs because the PDA is a more fundamental and more widely-used model of computation than the counter automaton.

Since our pushdown automata accept finite languages, which correspond to finite sets of various connected subgraphs of the tree, a finite automaton could also be used instead of a pushdown automaton. However, such finite automaton would have significantly more states than the PDA, in which the underlying tree structure is efficiently processed by the pushdown store.

Early presentations of the tree pattern pushdown automaton and the nonlinear tree pattern pushdown automaton can be found in [21, 19] and [26], respectively. This paper can be considered as an extended version of these publications.

The paper is organised as follows. The second section discusses related works of existing (nonlinear) tree pattern matching algorithms. The third section contains basic definitions. The fourth section is devoted to indexing trees for tree pattern matching by pushdown automata. The fifth section describes indexing trees for nonlinear tree pattern matching by pushdown automata. Section 6 deals with nonlinear tree pattern matching for more than one nonlinear variable in nonlinear tree patterns. The seventh section describes our implementations and show experimental results. The last section is a conclusion.

2. Related Works

Some algorithms for (nonlinear) tree pattern matching are known. All of them use the approach which is represented by the preprocessing of the (nonlinear) tree pattern. For tree pattern matching algorithms see [15, 6, 12].

Nonlinear tree pattern matching algorithm described in [24] reads the Euler linear notation of both a subject tree and a nonlinear tree pattern. Euler notation is a tree linear notation, which contains a node each time it is visited during the preorder traversing of the tree. This means that every node appears exactly $1 + arity(node)$-times in Euler notation. Our method presented in this paper uses a standard tree prefix notation, which contains every node just once.

In [24] factors which represent some subtrees in a subject tree in Euler notation are constructed. The standard Aho-Corasick automaton [1] is then constructed for these factors. The subject tree in Euler notation is processed by the constructed Aho-Corasick automaton and a binary array is constructed for each factor of the nonlinear tree pattern. Locations of factors of the input subject tree in Euler notation are then transformed to arrays of ones and zeros, which describes locations of this factor in the subject tree in Euler notation. In this way the nonlinear variables are matched.

3. Basic notions

We define notions on trees similarly as they are defined in [2, 14, 15].

3.1. Alphabet

An *alphabet* is a finite nonempty set of *symbols*. A *ranked alphabet* is a finite nonempty set of symbols each of which has a unique nonnegative *arity* (or *rank*). Given a ranked alphabet $\mathcal{A}$, the arity of a symbol $a \in \mathcal{A}$ is denoted $Arity(a)$. The set of symbols of arity p is denoted by $\mathcal{A}_p$. Elements of arity $0, 1, 2, \ldots, p$ are respectively called nullary (constants), unary, binary, $\ldots$, p-ary symbols. We assume that $\mathcal{A}$ contains at least one constant. In the examples we use numbers at the end of identifiers for a short declaration of symbols with arity. For instance, $a2$ is a short declaration of a binary symbol a.

3.2. Tree, tree pattern, nonlinear tree pattern

Based on concepts from graph theory (see [2]), a tree over an alphabet $\mathcal{A}$ can be defined as follows:

An *graph* G is a pair (N, R), where N is a set of nodes and R is a set of edges such that each element of R is of the form (f, g), where $f, g \in N$. This element will indicate that, for node f, there is an edge between node f and node g.

A *directed graph* G is a graph, where each element of R of the form (f, g) indicates that, there is an edge leaving node f and entering node g. This edge is ordered from f to g. An *undirected graph* G is a graph in which no such ordering of edges is given.

A sequence of nodes $(f_0, f_1, \ldots, f_n)$, $n \geq 1$, is a *path* of length n from node f_0 to node f_n if there is an edge which leaves node f_{i-1} and enters node f_i for $1 \leq i \leq n$. A *labelling* of an ordered graph $G = (N, R)$ is a mapping of N into a set of labels. In the examples we use a_f for a short declaration of node f labelled by symbol a.

A directed graph is *connected* if there exists a path from f_u to f_v for each pair of nodes (f_u, f_v), $u \neq v$, of the graph.

A *cycle* is a path $(f_0, f_1, \ldots, f_n)$ in which $f_0 = f_n$.

Given a node f of a directed graph, its *out-degree* is the number of distinct pairs $(f, g) \in R$, where $g \in N$. By analogy, the *in-degree* of node f is the number of distinct pairs $(g, f) \in R$, where $g \in N$.

A *tree* is a connected graph without any cycle. In this paper we assume that the tree has at least one node. Any node of a tree can be selected as a *root* of the tree. A tree with a root is called *rooted tree.*

A tree can be *directed*. A *rooted and directed tree* t is an acyclic connected directed graph $t = (N, R)$ with a special node $r \in N$, called the *root*, such that (1) r has in-degree 0, (2) all other nodes of t have in-degree 1, (3) there is just one path from the root r to every $f \in N$, where $f \neq r$.

Nodes of a directed tree with out-degree 0 are called *leaves*.

A *labelled, (rooted, directed) tree* is a tree having the following property: (4) every node $f \in N$ is labelled by a symbol $a \in \mathcal{A}$, where $\mathcal{A}$ is an alphabet.

A *ranked, (labelled, rooted, directed) tree* is a tree labelled by symbols from a ranked alphabet and out-degree of a node f labelled by symbol $a \in \mathcal{A}$ equals to $Arity(a)$. Nodes labelled by nullary symbols (constants) are leaves.

An *ordered, (ranked, labelled, rooted, directed) tree* is a tree where direct descendants $a_{f1}, a_{f2}, \ldots, a_{fn}$ of a node a_f having an $Arity(a_f) = n$ are ordered.

Example 1. Consider a ranked alphabet $\mathcal{A} = \{a2, a1, a0\}$. Consider an ordered, ranked, labelled, rooted, and directed tree $t_1 = (\{a2_1, a2_2, a0_3, a1_4, a0_5, a1_6, a0_7\}, R_1)$ over $\mathcal{A}$, where R_1 is a set of the following ordered pairs:
$R_1 = \{(a2_1, a2_2), (a2_1, a1_6), (a2_2, a0_3), (a2_2, a1_4), (a1_4, a0_5), (a1_6, a0_7)\}$.

Tree t_1 in prefix notation is $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$.

Trees can be represented graphically, and tree t_1 is illustrated in Figure 1.

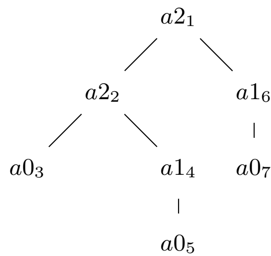


Fig. 1. Tree t_1 from Example 1

A *subtree* of a tree $t = (N, R)$ is any tree $t' = (N', R')$ such that: (1) N' is nonempty and contained in N, (2) $R' = A' \times A' \cap R$, and (3) No node of $A \setminus A'$ is a descendant of a node in A'.

The height of a tree t, denoted by *Height(t)*, is defined as the length of the longest path leading from the root of t to a leaf of t.

To define a *tree pattern*, we use a special nullary symbol S, not in alphabet $\mathcal{A}$, $Arity(S) = 0$, which is a variable and serves as a placeholder for any subtree. A tree pattern is defined as a labelled ordered tree over an alphabet $\mathcal{A} \cup \{S\}$. We will assume that the tree pattern contains at least one node labelled by a symbol from $\mathcal{A}$. A tree pattern containing at least one symbol S will be called a *tree template*.

A tree pattern p with $k \geq 0$ occurrences of the symbol S *matches* a subject tree t at node n if there exist subtrees $t_1, t_2, \ldots, t_k$ (not necessarily the same) of the tree t such that the tree p', obtained from p by substituting the subtree t_i for the i-th occurrence of S in p, $i = 1, 2, \ldots, k$, is equal to the subtree of t rooted at n.

The *nonlinear tree pattern* can contain also other special nullary symbols – nonlinear variables, not in alphabet $\mathcal{A}$. These symbols serve as placeholders for

specific subtrees. Every occurrence of a symbol X in a nonlinear tree pattern is matched with the same subtree. A nonlinear tree pattern has to contain at least one symbol from $\mathcal{A}$. A nonlinear tree pattern which contains at least two equal nonlinear variables will be called a *nonlinear tree template*.

A nonlinear tree pattern np with $k \geq 2$ occurrences of a nonlinear variable X *matches* a subject tree t at node n if there exists a subtree t_X of the tree t and subtrees $t_1, t_2, \ldots, t_m$ (not necessarily the same) of the tree t such that the tree np', obtained from np by substituting the subtree t_X for the i-th, $1 \leq i \leq k$, occurrences of X in np, and by substituting the subtree t_i for the i-th occurrence of S in p, $i = 1, 2, \ldots, m$, is equal to the subtree of t rooted at n.

Example 2. Consider a tree $t_1 = (\{a2_1,\, a2_2,\, a0_3,\, a1_4,\, a0_5,\, a1_6,\, a0_7\},\, R_1)$ from Example 1, which is illustrated in Figure 1.

Consider a tree pattern p_1 over $\mathcal{A}, p_1 = (\{a2_1, a0_2, a1_3, a0_4\}, R_{p1})$. Tree pattern p_1 in prefix notation os $pref(p_1) = a2\ a_0\ a_1\ a_0$.

$$R_{p1} = \{((a2_1, a0_2), (a2_1, a1_3)), ((a1_3, a0_4))\}$$

Consider a tree pattern p_2 over $\mathcal{A} \cup \{S\}$, $p_2 = (\{a2_1,\, S_2,\, a1_3,\, S_4\},\, R_{p2})$. Tree pattern p_2 in prefix notation is $pref(p_2) = a2\ S\ a1\ S$. Note that symbol S can occur in a nonlinear tree pattern and it serves as a linear variable.

$$R_{p2} = \{(a2_1, S_2), (a2_1, a1_3), (a1_3, S_4)\}$$

Consider a nonlinear tree pattern p_3 over $\mathcal{A} \cup \{S, X\}$, $p_3 = (\{a2_1,\, X_2,\, a1_3,\, X_4\},\, R_{p3})$. Nonlinear tree pattern p_3 in prefix notation is $pref(p_3) = a2\ X\ a1\ X$.

$$R_{p3} = \{(a2_1, X_2), (a2_1, a1_3), (a1_3, X_4)\}$$

Tree patterns p_1, p_2 and p_3 are illustrated in Figure 2. Tree pattern p_1 has one occurrence in tree t_1 – it matches at node 2 of t_1. Tree pattern p_2 has two occurrences in tree t_1 – it matches at nodes 1 and 2 of t_1. Nonlinear tree pattern p_3 has one occurrence in tree t_1 – it matches at node 2 of t_1.

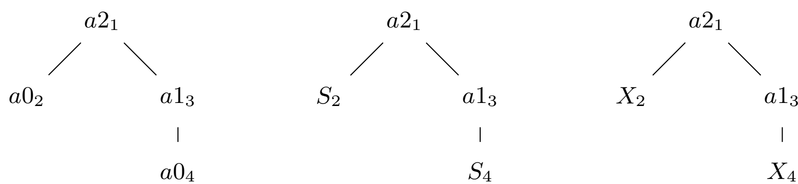


Fig. 2. Tree pattern p_1 (left), tree template p_2 (center) and nonlinear tree template p_3 (right) from Example 2

3.3. Language, finite and pushdown automata

We define notions from the theory of string languages similarly as they are defined in [2, 16].

A *language* over an alphabet $\mathcal{A}$ is a set of strings over $\mathcal{A}$. Symbol $\mathcal{A}^*$ denotes the set of all strings over $\mathcal{A}$ including the empty string, denoted by ε. Set $\mathcal{A}^+$ is defined as $\mathcal{A}^+ = \mathcal{A}^* \setminus \{\varepsilon\}$. Similarly, for string $x \in \mathcal{A}^*$, symbol x^m, $m \geq 0$, denotes the m-fold concatenation of x with $x^0 = \varepsilon$. Set x^* is defined as $x^* = \{x^m : m \geq 0\}$, $x^+ = \{x^m : m \geq 1\}$ and $x^* = x^+ \cup \{\varepsilon\}$.

A *nondeterministic pushdown automaton* (nondeterministic PDA) is a seven-tuple $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$, where Q is a finite set of *states*, $\mathcal{A}$ is an *input alphabet*, G is a *pushdown store alphabet*, δ is a mapping from $Q \times (\mathcal{A} \cup \{\varepsilon\}) \times G$ into a set of finite subsets of $Q \times G^*$, $q_0 \in Q$ is an initial state, $Z_0 \in G$ is the initial pushdown store symbol, and $F \subseteq Q$ is the set of final (accepting) states.

Triple $(q, w, x) \in Q \times \mathcal{A}^* \times G^*$ denotes the configuration of a pushdown automaton. We will write the top of the pushdown store x on its left hand side. The initial configuration of a pushdown automaton is a triple (q_0, w, Z_0) for the input string $w \in \mathcal{A}^*$. The relation $\vdash_M \subset (Q \times \mathcal{A}^* \times G^*) \times (Q \times \mathcal{A}^* \times G^*)$ is a *transition* of a pushdown automaton M. It holds that $(q, aw, \alpha\beta) \vdash_M (p, w, \gamma\beta)$ if $(p, \gamma) \in \delta(q, a, \alpha)$. The k-th power, transitive closure, and transitive and reflexive closure of the relation $\vdash_M$ is denoted $\vdash_M^k$, $\vdash_M^+$, $\vdash_M^*$, respectively.

A pushdown automaton is *input–driven* if each of its pushdown operations is determined only by the input symbol.
A language L accepted by a pushdown automaton M is defined in two distinct ways:

1. *Accepting by final state:* $L(M) = \{x : (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \gamma) \wedge x \in \mathcal{A}^* \wedge \gamma \in G^* \wedge q \in F\}$.
2. *Accepting by empty pushdown store:* $L_\varepsilon(M) = \{x : (q_0, x, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon) \wedge x \in \mathcal{A}^* \wedge q \in Q\}$.

If the pushdown automaton accepts the language by empty pushdown store, then the set F of final states is the empty set.

Unreachable states are states $p \in Q$ from automaton $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ which are not reachable from the initial state because there is no sequence of transitions from the initial state to that particular state p. Formally, there are no transitions that allow $(q_0, kw, Z_0) \vdash_M^+ (p, w, \gamma)$.

Unnecessary states are states $p \in Q$ from automaton $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ which are not connected to any final state $f \in F$ if automaton accepts by final states, or not connected to any state, where $\gamma \in G^*$ may be ε, if automaton accepts by empty pushdown store.

Pushdown automaton $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ is acyclic if it does not contain transitions $(q, x_1, \gamma_1) \vdash_M^+ (q, x_2, \gamma_2)$, where $xx_2 = x_1$, $x \neq \varepsilon$ and $q \in Q$.

3.4. String suffix and factor automata

String suffix and factor automata are finite automata that were introduced in [4, 7] as a mechanism for eliminating redundancy in string suffix trees [9, 10,

22, 25]. Given a string $s \in \mathcal{A}^*$, the suffix and factor automaton constructed for the string s accepts all suffixes and substrings, respectively, of the string s in time linear to the length of the input suffix and the input substring, respectively, and not depending on the length of the string s. In [9, 10, 25], suffix and factor automata are defined as such minimal deterministic finite automata. In [23, 22], their basic nondeterministic versions are also presented. In some literature, the deterministic suffix automaton is also called the *directed acyclic word graph (DAWG)*.

Example 3. Given a string $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is the prefix notation of tree t_1 from Example 1, the corresponding nondeterministic suffix automaton is $FM_{nsuf}(pref(t_1)) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \delta_n, 0, \{7\})$, where its transition diagram is illustrated in Figure 3. For the construction of the nondeterministic suffix automaton, see [22].

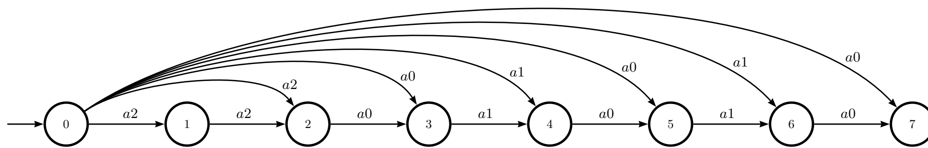


Fig. 3. Transition diagram of the nondeterministic string suffix automaton $FM_{nsuf}(pref(t_1))$ for prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ of tree t_1 from Example 1

4. Indexing trees for tree pattern matching

In this section, algorithms and theorems regarding tree pattern PDAs for trees in prefix notation are given, and the tree pattern PDAs and their construction are demonstrated on an example. A tree pattern can be either a subtree or a tree template, which contains at least one special nullary symbol S representing a subtree. Tree pattern PDAs are an extension of subtree PDAs, introduced in [18]. A subtree PDA is analogous to the string suffix automaton and it accepts a linear notation of all subtrees of a given tree. The pushdown operations are used to process the tree structure. New states and transitions, which are used for processing the special nullary symbols S in tree templates, are additionally present in the tree pattern PDA. The pushdown operations are the same.

Definition 1. *Let t and $pref(t)$ be a tree and its prefix notation, respectively. A* tree pattern pushdown automaton *for $pref(t)$ accepts all tree patterns in prefix notation which match the tree t.*

Given a subject tree, first we construct a so-called deterministic *treetop PDA* for this tree in prefix notation, which accepts all tree patterns that match the subject tree and contain the root of the subject tree. The deterministic treetop PDA is defined by the following definition. States and transitions of the treetop pushdown automaton are computed by Algorithm 1. Finally, the correctness Algorithm 1 is proved by Theorem 1.

Definition 2. *Let t, r and $pref(t)$ be a tree, its root and its prefix notation, respectively. A* treetop pushdown automaton $M_{pt}(t) = (0, 1, 2, ..., n, A \cup S, S, \delta, 0, S, \emptyset)$ *for $pref(t)$ accepts all tree patterns in prefix notation which have the root r and match the tree t.*

The construction of the treetop PDA is described by the following algorithm. The treetop PDA is deterministic.

Algorithm 1 *Construction of a treetop PDA for a tree t in prefix notation $pref(t)$.*
Input: *A tree t over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1 a_2 \ldots a_n$, $n \geq 1$.*
Output: *Treetop PDA $M_{pt}(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A} \cup \{S\}, \{S\}, \delta, 0, S, \emptyset)$.*
Method:

1. *For each state i, where $1 \leq i \leq n$, create a new transition $\delta(i-1, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$.*
2. *Create a set $srms = \{\ i\ : 1 \leq i \leq n,\ \delta(i-1, a, S) = (i, \varepsilon),\ a \in \mathcal{A}_0\}$. The abbreviation* srms *stands for Subtree Right Most States.*
3. *For each state i, where $i = n-1, n-2, \ldots, 1$, $\delta(i, a, S) = (i+1, S^p)$, $a \in \mathcal{A}_p$, create a new transition $\delta(i, S, S) = (l, \varepsilon)$ such that $(i, xy, S) \vdash^+_{M_p(t)} (l, y, \varepsilon)$ as follows:*
   If $p = 0$, create a new transition $\delta(i, S, S) = (i+1, \varepsilon)$.
   Otherwise, if $p \geq 1$, create a new transition $\delta(i, S, S) = (l, \varepsilon)$, where l is the p-th smallest integer such that $l \in srms$ and $l > i$. Remove all j, where $j \in srms$, and $i < j < l$, from $srms$. □

The treetop PDA is similar to the prefix string finite automaton. Moreover, there exists additional transitions reading symbol S, which represents a subtree, and these transitions skip over parts which are subtrees of the tree in prefix notation. The automaton uses the pushdown store for computing a checksum so that the input would be a valid prefix notation of a tree.

The construction of treetop PDA by Algorithm 1 is illustrated in the following example.

Example 4. Consider tree t_1 in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 1, which is illustrated in Figure 1. The deterministic treetop PDA, constructed by Algorithm 1, is deterministic PDA $M_{pt}(t_1) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \{S\}, \delta_1, 0, S, \emptyset))$, where mapping δ_1 is a set of the following transitions:

$$
\begin{array}{ll}
\delta_1(0, a2, S) = (1, SS) & \\
\delta_1(1, a2, S) = (2, SS) & \delta_1(1, S, S) = (5, \varepsilon) \\
\delta_1(2, a0, S) = (3, \varepsilon) & \delta_3(2, S, S) = (3, \varepsilon) \\
\delta_1(3, a1, S) = (4, S) & \delta_1(3, S, S) = (5, \varepsilon) \\
\delta_1(4, a0, S) = (5, \varepsilon) & \delta_1(4, S, S) = (5, \varepsilon) \\
\delta_1(5, a1, S) = (6, S) & \delta_1(5, S, S) = (6, \varepsilon) \\
\delta_1(6, a0, S) = (7, \varepsilon) & \delta_1(6, S, S) = (7, \varepsilon)
\end{array}
$$

The transition diagram of deterministic treetop PDA $M_{pt}(t_1)$ is illustrated in Figure 4. In this figure for each transition rule $\delta(p, a, \alpha) = (q, \beta)$ from δ the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \mapsto \beta$.

Deterministic treetop PDA $M_{pt}(t_1)$ has been constructed by Algorithm 1 as follows. We can see that the initial set $srms = \{3, 5, 7\}$. Then, new transitions, which read symbol S, are created in the following order: $\delta_4(6, S, S) = (7, \varepsilon)$, $\delta_4(5, S, S) = (7, \varepsilon)$, $\delta_4(4, S, S) = (5, \varepsilon)$, $\delta_4(3, S, S) = (5, \varepsilon)$, $\delta_4(2, S, S) = (3, \varepsilon)$, and $\delta_4(1, S, S) = (5, \varepsilon)$. □

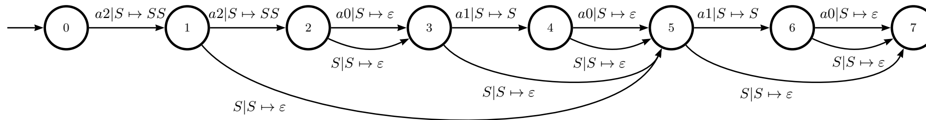


Fig. 4. Transition diagram of deterministic treetop pushdown automaton $M_{pt}(t_1)$ for tree in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 4

Theorem 1. *Given a tree t and its prefix notation $pref(t)$, the PDA $M_{pt}(t)$ constructed by Algorithm 1 is a treetop PDA for $pref(t)$.*

Proof. Let r be the root of t. The PDA $M_{pt}(t)$ is a simple extension of the PDA, which is constructed by step 1 and accepts the tree t in prefix notation. It holds for new transitions added by step 3, which read the special nullary symbol S, that $\delta(q_1, S, S) = (q_2, \varepsilon)$ if and only if $(q_1, w, S) \vdash^+_{M_{pt}(t)} (q_2, \varepsilon, \varepsilon)$ and q_1 is not the initial state 0. This means that the new added transitions reading S correspond just to subtrees not containing the root r. Thus, the PDA $M_{pt}(t)$ accepts all tree patterns in prefix notation which contain the root r and match the tree t. □

The nondeterministic tree pattern PDA for trees in prefix notation is constructed as an extension of the deterministic treetop PDA: for each state of the treetop PDA with an incoming transition which reads a symbol $a \in \mathcal{A}$ we add the same transition from the starting state to that state. This construction is described by the following algorithm.

Algorithm 2 *Construction of a nondeterministic tree pattern PDA for a tree t in prefix notation $pref(t)$.*
Input: *A tree t over a ranked alphabet $\mathcal{A}$; prefix notation $pref(t) = a_1 a_2 \ldots a_n$, $n \geq 1$.*
Output: *Nondeterministic tree pattern PDA $M_{npt}(t) = (\{0, 1, 2, \ldots, n\}, \mathcal{A} \cup \{S\}, \{S\}, \delta, 0, S, \emptyset)$.*
Method:

1. *Create $M_{npt}(t)$ as $M_{pt}(t)$ by Algorithm 1.*
2. *For each state i, where $2 \leq i \leq n$, create a new transition $\delta(0, a_i, S) = (i, S^{Arity(a_i)})$, where $S^0 = \varepsilon$.* □

The tree pattern PDA is similar to the string factor finite automaton. Its construction is based on the treetop PDA and the extension is that the tree pattern accepted by the automaton can be matched on any node of the tree. For this reason, additional transitions are created.

Example 5. Consider tree t_1 in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$ from Example 1, which is illustrated in Figure 1. The nondeterministic tree pattern PDA accepting all tree patterns matching tree t_1, which has been constructed by Algorithm 2, is nondeterministic PDA
$M_{npt}(t_1) = (\{0, 1, 2, 3, 4, 5, 6, 7\}, \mathcal{A}, \{S\}, \delta_2, 0, S, \emptyset))$, where mapping δ_2 is a set of the following transitions:

$$
\begin{array}{lll}
\delta_2(0, a2, S) = (1, SS) & & \\
\delta_2(1, a2, S) = (2, SS) & \delta_2(1, S, S) = (5, \varepsilon) & \delta_2(0, a2, S) = (2, SS) \\
\delta_2(2, a0, S) = (3, \varepsilon) & \delta_3(2, S, S) = (3, \varepsilon) & \delta_2(0, a0, S) = (3, \varepsilon) \\
\delta_2(3, a1, S) = (4, S) & \delta_2(3, S, S) = (5, \varepsilon) & \delta_2(0, a1, S) = (4, S) \\
\delta_2(4, a0, S) = (5, \varepsilon) & \delta_2(4, S, S) = (5, \varepsilon) & \delta_2(0, a0, S) = (5, \varepsilon) \\
\delta_2(5, a1, S) = (6, S) & \delta_2(5, S, S) = (6, \varepsilon) & \delta_2(0, a1, S) = (6, S) \\
\delta_2(6, a0, S) = (7, \varepsilon) & \delta_2(6, S, S) = (7, \varepsilon) & \delta_2(0, a0, S) = (7, \varepsilon)
\end{array}
$$

The transition diagram of nondeterministic tree pattern PDA $M_{npt}(t_1)$ is illustrated in Figure 5. Again, in this figure for each transition rule $\delta(p, a, \alpha) = (q, \beta)$ from δ the edge leading from state p to state q is labelled by the triple of the form $a|\alpha \mapsto \beta$. □

In the following theorem we prove the correctness of the constructed tree pattern PDA.

Theorem 2. *Given a tree t and its prefix notation $pref(t)$, the PDA $M_{npt}(t)$ constructed by Algorithm 2 is a tree pattern PDA for $pref(t)$.*

Proof. The PDA $M_{npt}(t)$ is a simple extension of the PDA $M_{pt}(t)$, which is constructed by Algorithm 1 and accepts all tree patterns in prefix notation which contain the root r of the tree t and match the tree t by empty pushdown store. The PDA $M_{npt}(t)$ contains new added transitions of the form $\delta(0, a_i, S) = (i, S^{Arity(a_i)})$. These transitions correspond just to the possibility that the first

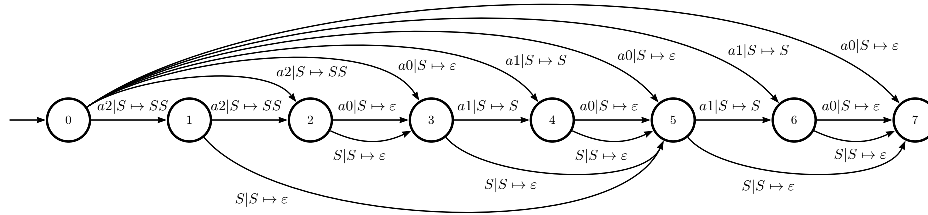


Fig. 5. Transition diagram of nondeterministic tree pattern pushdown automaton M_{npt} (t_1) from Example 5 for tree in prefix notation $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$

symbol of a tree pattern to be accepted can be any node of the tree t. Thus, the PDA $M_{npt}(t)$ accepts all tree patterns in prefix notation which match the tree t. □

Lemma 1. *Given a tree t with n nodes, the number of distinct tree patterns which match the tree t can be at most $2^{n-1} + n$.*

Proof. First, subtrees of any subtree of the tree t can be replaced by the special nullary symbol S and the tree template resulting from such a replacement is a tree pattern which matches the tree. Given a tree with n nodes, the maximal number of subsets of subtrees that can be replaced by the special nullary symbol S occurs for the case of a tree t_2 whose structure is given by the prefix notation $pref(t_2) = a(n-1)\ a_1 0\ a_2 0 \ldots a_{n-1} 0$, where $n \geq 2$. Such a tree is illustrated in Figure 6. In this tree, each of the nullary symbols $a_1 0, a_2 0, \ldots, a_{n-1} 0$ can be replaced by nullary symbol S, and therefore we can create 2^{n-1} distinct tree templates which are tree patterns matching the tree t_2.

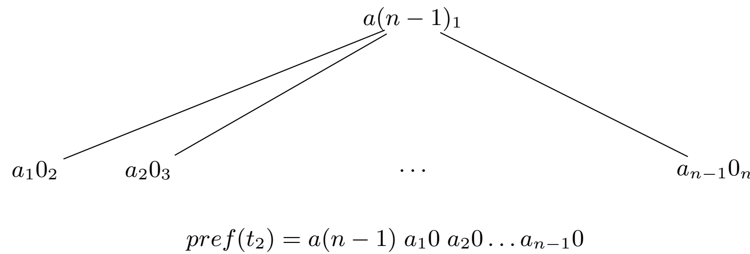


Fig. 6. A tree t_2 with $2^{n-1} + n$ distinct tree patterns matching the tree t_3 and its prefix notation

Second, the tree t itself and all its subtrees not containing the root are tree patterns which match the tree, which gives n other distinct tree patterns (provided all the subtrees are unique).

Thus, the total number of distinct tree patterns matching the tree t can be at most $2^{n-1} + n$. □

Lemma 2. *The number of states of a nondeterministic tree pattern pushdown automaton M_{ntp} is $m + 1$, where m is the number of nodes of a subject tree.*

Proof. There is one state for each symbol in $pref(t)$ plus and the initial state. Thus, the number of states is $m + 1$. □

Lemma 3. *The number of transitions of a nondeterministic tree pattern pushdown automaton M_{ntp} is $3m - 2$, where m is the number of nodes of a subject tree.*

Proof. There is one transition for each symbol in $pref(t)$, which forms the "backbone" of the automaton. There are exactly $m-1$ transitions from the initial state to every other state. Finally, there is one transition for symbol S leading from every state except the initial state. Thus, the number of states is then $3m - 2$. □

5. Indexing trees for nonlinear tree pattern matching

Definition 3. *Let t and $pref(t)$ be a tree and its prefix notation, respectively. A* nonlinear tree pattern pushdown automaton *for $pref(t)$ accepts all nonlinear tree patterns in prefix notation which have at most one nonlinear variable and match the tree t.*

5.1. Basic nonlinear tree pattern pushdown automaton

In our indexing pushdown automata for nonlinear tree pattern matching we construct new parts called tails, which represent parts of the pushdown automaton after reading nonlinear variables.

Definition 4. *Given a tree pattern pushdown automaton $M = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ and a state $q_t \in Q$, the $tail(M, q_t) = (Q_t, \mathcal{A}, G, \delta_t, q_t, S, F)$. $Q_t = Q \setminus Q_{us}$, Q_{us} is a set of unreachable states from q_t, $\delta_t = \delta \setminus \delta_{us}$, δ_{us} are transitions leading from or to state $q_n \in Q_{us}$.*

Example 6. Consider a tree pattern pushdown automaton $M_{npt}(t_1)$ from Example 5, which is an index of tree t_1 from Example 1. The tail of automaton with initial state $q_t = 3$ is $tail(M_{npt}(t_1), 3) = (Q, \mathcal{A} \cup \{S\}, \{S\}, \delta, 3, S, \varnothing)$ constructed from tree pattern pushdown automaton shown in Figure 5. The corresponding transition diagram is illustrated in Figure 7. □

We note that every node of a tree t is the root of just one subtree, which is represented by symbol S. The prefix notation of such subtree is a factor of $pref(t_1)$. These factors are in the tree pushdown automaton "skipped" by transitions for input symbol S.

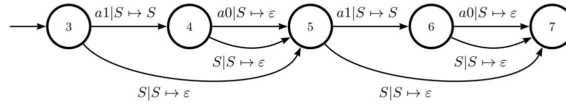


Fig. 7. Tail of tree pattern pushdown automaton $tail(M_{npt}(t_1), 3)$ from Example 6

Definition 5. *Given a tree pattern pushdown automaton $M_{npt}(t) = (Q, \mathcal{A}, G, \delta, q_0, Z_0, F)$ and a state $q \in Q$, the* subtree skipped by transition $sst(q) = b_1 b_2 \ldots b_m$*, where $b_1, b_2, \ldots, b_m \in \mathcal{A}$, is given by a labelled path $b_1, b_2, \ldots, b_m$ in the PDA $M_{npt}(t)$ between states q and q_t, where $(q_t, \varepsilon) \in \delta(q, S, S)$.*

Informally, $sst(q)$ is the prefix notation of the subtree which is skipped by transition S leading from the state q. $sst(q)$ is used in Algorithm 3 to determine which subtree of the subject tree was "assigned" to a particular automaton tail.

Example 7. Consider a tree pattern pushdown automaton $M_{npt}(t_1)$ from Example 5, which is an index of tree t_1 from Example 1. The subtree skipped by transition $sst(1) = a2\ a0\ a1\ a0$. □

The construction of basic nonlinear tree pattern PDA consists of two algorithms. Algorithm 3 constructs tails from the original tree pattern pushdown automaton. Algorithm 4 recursively connects these created tails to the pushdown automaton being created.

Algorithm 3 Recursive construction of tail of nondeterministic basic nonlinear tree pattern automaton.
Input: Tail of nondeterministic tree pattern pushdown automaton M_{tnpt}, string representing subtree skipped by transition $x = sst(q)$.
Output: Recursively created tail $nta(M_{tnpt}, x)$.
Method:

1. For each transition $(q_t, \varepsilon) \in \delta(q, S, S)$ in automaton M_{tnpt} where $sst(q) = x$ do:
   1.1. Create $M_{tmp} = nta(tail(M_{tnpt}, q_t), x)$ using Algorithm 3.
   1.2. Add new state q_{id} to M_{tnpt} where q_{id} is copy of state q_t.
   1.3. Add new transition $(q_{id}, \varepsilon) \in \delta(q, X, S)$ to M_{tnpt}.
   1.4. Add M_{tmp} to M_{tnpt} and merge initial state of M_{tmp} with q_{id}.
2. $nta(M_{tnpt}, x)$ is M_{tnpt}.

Algorithm 4 Construction of nondeterministic basic nonlinear tree pattern pushdown automaton.
Input: Nondeterministic tree pattern pushdown automaton $M_{npt}(t)$.
Output: Nondeterministic basic nonlinear tree pattern pushdown automaton $M_b(t)$.
Method:

1. For each transition $(q_t, \varepsilon) \in \delta(q, S, S)$ in automaton $M_{npt}(t)$ do:

1.1. Create $M_{tmp} = nta(tail(M_{npt}(t), q_t), sst(q))$ using Algorithm 3.
1.2. Add new state q_{id} to $M_{npt}(t)$ where q_{id} is copy of state q_t.
1.3. Add new transition $(q_{id}, \varepsilon) \in \delta(q, X, S)$ to $M_{npt}(t)$.
1.4. Add M_{tmp} to $M_{npt}(t)$ and merge initial state of M_{tmp} with q_{id}.

2. $M_b(t)$ is $M_{npt}(t)$.

The nonlinear tree pattern PDA is similar to the tree pattern PDA. The difference between Algorithm 3 and Algorithm 4 is that Algorithm 3 calls itself only when processing transition for symbol S leading from state q, where $sst(q)$ equals its subtree parameter. On the other hand, Algorithm 4 calls Algorithm 3 for each transition for symbol S.

Example 8. Given a string $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is a prefix notation of tree t_1 from Example 1, the corresponding nondeterministic basic nonlinear tree pattern pushdown automaton is
$M_b(t_1) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \emptyset)$, where its transition diagram is illustrated in Figure 8. □

5.2. Nonlinear tree pattern pushdown automaton

Some states of the pushdown automaton constructed by Algorithm 4 can be merged so that states in nondeterministic nonlinear tree pattern pushdown automaton M_{nntp} for a subject tree t $M_{nntp}(t) = (\{0,\ 1,\ 2,\ \ldots,\ n,\ x_1,\ \ldots,\ n_1,$ $y_2,\ \ldots,\ n_2,\ \ldots,\ z_m,\ \ldots,\ n_m\}, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0,\ S,\ \emptyset)$ would still track both assigned subtree and the same number of nonlinear variables read from the pattern. Merged states are those from tails with the same assigned subtree and the same number of nonlinear variables read.

Definition 6. Let $M_b(t) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, q_0, S, \emptyset)$ be a basic nondeterministic nonlinear tree pattern PDA constructed by Algorithm 4. Let x be the longest string over alphabet $\mathcal{A}$, where $(q, x, \alpha) \vdash^*_{Mb} (q_f, \varepsilon, \beta)$. The *tree node state label* $tnsl(q)$ is defined $tnsl(q) = |pref(t)| - |x|$.

Algorithm 5 Algorithm for counting the $tnsl$.
Input: Nondeterministic basic nonlinear tree pattern pushdown automaton $M_b(t)$ and state q for which the $tnsl$ is counted.
Output: Number representing $tnsl$.
Variables: Temporary number n, State $initial$.
Method:

1. n = 0. $initial$ is the starting state of $M_b(t)$.
2. Do:
   2.1. If exists transition $(q, S^{arity(a)}) \in \delta(q_{prev}, a, S)$ where $a \in \mathcal{A}$ and $q_{pref} \neq initial$ do:
      2.1.1. $n = n + 1$, $q = q_{prev}$.
      2.1.2. Goto step [2.].

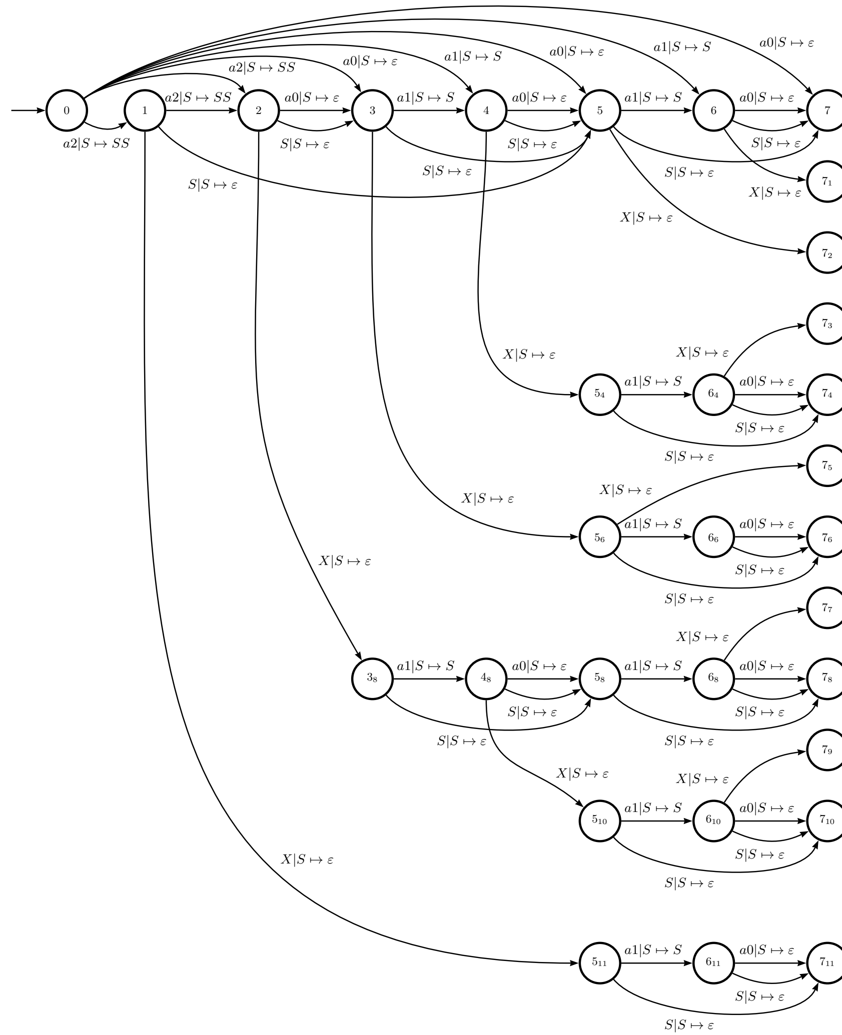


Fig. 8. Basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t_1)$ from Example 8 constructed for tree t_1 shown in Figure 1

2.2. If exists transition $(q, \varepsilon) \in \delta(q_{prev}, X, S)$ where X is nonlinear variable do:
 2.2.1. $n = n + |sst(q_{prev})|$, $q = q_{prev}$.
 2.2.2. Goto step [2.].
2.4. Output $n + 1$.

Example 9. Given a basic nondeterministic nonlinear tree pattern pushdown automaton is $M_b(t_1) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \emptyset)$, its transition diagram is shown in Figure 8.

Then, $tnsl(3) = 3$, $tnsl(5_4) = 5$, $tnsl(7_{11}) = 7$, $tnsl(7_9) = 7$. □

Definition 7. Given a basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \emptyset)$ created by Algorithm 4, the *number of nonlinear variable transitions* $nnv(q, X)$ is the number of transitions reading nonlinear variable X on the path from the initial state q_0 to state q, where q and $q_0 \in Q$.

Algorithm 6 Algorithm for counting the nnv.
Input: Basic nondeterministic nonlinear tree pattern pushdown automaton $M_b(t)$ and state q for which the $tnsl$ is counted.
Output: Number representing nnv.
Variables: Temporary number n, State $initial$.
Method:

1. n = 0. $initial$ is the starting state of $M_b(t)$.
2. Do:
 2.1. If exists transition $(q, S^{arity(a)}) \in \delta(q_{prev}, a, S)$ where $a \in \mathcal{A}$ and $q_{pref} \neq initial$ do:
  2.1.1. $q = q_{prev}$.
  2.1.2. Goto with step [2.].
 2.2. If exists transition $(q, \varepsilon) \in \delta(q_{prev}, X, S)$ where X is nonlinear variable do:
  2.2.1. $n = n + 1$, $q = q_{prev}$.
  2.2.2. Goto with step [2.].
 2.3 Output n.

Definition 8. Given a nondeterministic basic nonlinear tree pattern pushdown automaton $M_b(t)$ created by Algorithm 4, the *mergeable states* $ms(M_b(t))$ is a collection of pairs (key, $value$), where key is a triplet ($sst(q)$, $nnv(u, X)$, $tnsl(u)$) and $value$ is a set of states. $ms(M_b(t))$ stores sets of states with the same number of transitions reading nonlinear variable X $nnv(q, X)$ and subtree skipped by transition $sst(q)$.
$ms(M_b(t)) = \{(sst(q_x), nnv(s_{a1}, X), tnsl(s_{a1})), \{s_{a1}, s_{a2}, \ldots\}),$
$(sst(q_y), nnv(s_{b1}, X), tnsl(s_{b2})), \{s_{b1}, s_{b2}, \ldots\}), \ldots\}$, where the first state s_1 from each set is the main state. State v is $sst(v)$ denoting state for state s_1 given by $(v, X\omega, S\gamma) \vdash (s_1, \omega, \gamma)$, where $\omega = (\mathcal{A} \cup \{S, X\})^*$. All states from that set are given by following: $\{\forall s : nnv(s, X) = nnv(s_1, X)$ and $sst(v) = sst(u)$ and $tnsl(s) = tnsl(s_1); s, s_1, u, v \in Q\}$, where state u is $sst(u)$ denoting state for state s given by $(u, X(\mathcal{A} \cup \{S\})^*\omega, S^\alpha) \vdash^* (s, \omega, S^\beta)$, where $\omega = (\mathcal{A} \cup \{S, X\})^*$.

Each set from the collection of sets of mergeable states $ms(M_b(t))$ defines states from nondeterministic basic nonlinear tree pattern pushdown automaton $M_b(t)$ that can be merged and the resulting automaton is called nondeterministic nonlinear tree pattern pushdown automaton $M_{nntp}(t)$. All states from each set defines the start of a merging process so that states that are reachable by the same sequence of transitions are also merged.

Example 10. Given a string $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is the prefix notation of tree t_1 from Example 1. The corresponding nondeterministic basic nonlinear tree pattern pushdown automaton is $M_b(t_1) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \varnothing)$, where its transition diagram and states are illustrated in Figure 8.

All states that occur in one of the set in the collection $ms(M_b(t_1))$ are target states from all transitions for a symbol X and the transitions for a symbol S which shares the source state.

$$ms(M_b(t_1)) = \{((a0, 1, 5), \{5_4, 5_8\}), ((a0, 1, 7), \{7_1, 7_4, 7_8\}), \\ ((a0, 2, 7), \{7_3, 7_7, 7_{10}\}), ((a1a0, 1, 7), \{7_2, 7_6\})\}.$$

Algorithm 7 Construction of the nondeterministic nonlinear tree pattern pushdown automaton.
Input: Nondeterministic basic nonlinear tree pattern pushdown automaton $M_b(t)$.
Output: Nondeterministic nonlinear tree pattern pushdown automaton $M_{nntp}(t)$.
Variables: Collection of sets of states $ms(M_b(t))$.
Method:

1. For all transitions $(u_1, \varepsilon) \in \delta(q, X, S)$ do:
   1.1. If the collection $ms(M_b(t))$ does not contain a set on a key ($sst(q)$, $nnv(u_1, X)$, $tnsl(u_1)$) create that set as an empty set.
   1.2. Add u_1 to the collection $ms(M_b(t))$ to the set on the key ($sst(q)$, $nnv(u_1, X)$, $tnsl(u_1)$).
2. For all transitions $(u_2, \varepsilon) \in \delta(q, S, S)$, where exists a transition $(u_1, \varepsilon) \in \delta(q, X, S)$ do:
   2.1. If $nnv(u_2, X) \neq 0$ and the collection $ms(M_b(t))$ does not contain a set on a key ($sst(q)$, $nnv(u_2, X)$, $tnsl(u_2)$) create that set as an empty set.
   2.2. Add u_2 to the collection $ms(M_b(t))$ to the set on the key ($sst(q)$, $nnv(u_2, X)$, $tnsl(u_2)$).
3. For each set in the collection $ms(M_b(t))$ do:
   3.1. Merge all states in this set, along with all states that follows-up.

Example 11. Given a string $pref(t_1) = a2\ a2\ a0\ a1\ a0\ a1\ a0$, which is the prefix notation of tree t_1 from Example 1, the corresponding nondeterministic nonlinear tree pattern pushdown automaton is
$M_{nntp}(t_1) = (Q, \mathcal{A} \cup \{S, X\}, \{S\}, \delta, 0, S, \varnothing)$, where merged states are in Example 10 and its transition diagram and states are illustrated in Figure 9. □

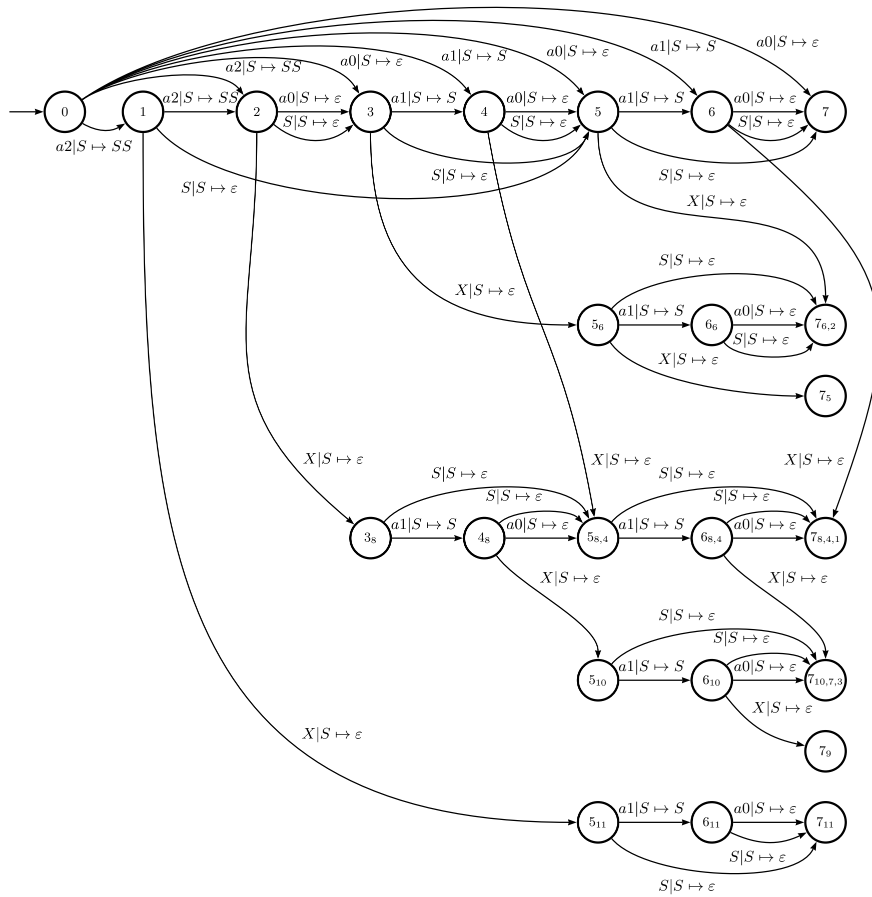


Fig. 9. Nondeterministic nonlinear tree pattern pushdown automaton $M_{nntp}(t_1)$ from Example 11 constructed by Algorithm 7 for subject tree shown in Figure 1

The nondeterministic nonlinear tree pattern pushdown automaton can be even minimalised by omitting the $nnv(q, X)$ part of the key value pairs of the collection $ms(M_b(t))$. The resulting automaton would represent an index of the subject tree for nonlinear tree pattern matching but would not be able to say how many nonlinear variables has been read during processing the nonlinear tree pattern.

5.3. Time and Space Complexity Analysis

Lemma 4. *Time complexity of accepting the nonlinear tree template by automaton created by Algorithm 7 is $\mathcal{O}(\sum_K k_i)$, where K is the set of all prefixes except ε, and k_i is the number of distinct sequences of transitions in automaton $M_{nntp}(t)$ for $k_i \in K$ which ends in a state of automaton M_{nntp}.*

Proof. Automata have to try all possible sequences of transitions according to tree template which occur in the nondeterministic nonlinear tree pattern automaton. Sequences of symbols of these transitions form a prefix of tree template. Prefix of the size of one symbol from tree template is handled by exactly n steps, where n is the number of all possible sequences of transitions in the automaton for that prefix. Prefix of the size of two symbols is handled by $n + m$ steps, where m is the number of all possible sequences of transitions in the automaton for that prefix. Note that handling two symbols prefix requires two transitions to be processed, however the first transition is already accounted by prefix of size of one symbol.

Exact time complexity is then the sum of all possible sequences of transitions in the automaton for all prefixes of nonlinear tree template, which is $\mathcal{O}(\sum_S rs_i)$. □

Lemma 5. *The number of states of nondeterministic nonlinear tree pattern pushdown automaton $M_{nntp}(t)$ created by Algorithm 7 is $\mathcal{O}(n(\sum_{i=0}^{s} r_i)) = \mathcal{O}(n^2)$, where n is the number of nodes of a subject tree and $\sum_{i=0}^{s} r_i$, where s is the number of distinct subtrees, and r_i is the number of repetitions of each subtree.*

Proof. Each occurrence of each unique subtree in tree increments the number of automaton tails, that were created for this subtree. The exact number of tails created for particular subtree is then r_i, where r_i is the number of repetitions of that subtree. Then the total number of tails for one nonlinear variable in automaton is the number of tails created for each unique subtree of indexed tree which is $\sum_{i=0}^{s} r_i$. The total number of tails does not count original automaton. The exact number of states of the automaton for one nonlinear variable is $\mathcal{O}(n(\sum_{i=0}^{s} r_i + 1))$ = $\mathcal{O}(n(\sum_{i=0}^{s} r_i))$. □

Lemma 6. *The number of transitions of nondeterministic nonlinear tree pattern pushdown automaton $M_{nntp}(t)$ created by Algorithm 7 is $\mathcal{O}(n^2 + n + \sum_{i=0}^{s}(\frac{r_i^2 + r_i}{2})) = \mathcal{O}(n^2)$, where n is the number of nodes of a subject tree, s is the number of distinct subtrees and r_i is the number of repetitions of each unique subtree.*

Proof. For all tails for one nonlinear variable, there are transitions reading symbol X between these tails. There is one transition heading to the last tail. There are two transitions heading to the previous tail, and so on. The number of transitions reading symbol X is $\sum_{i=0}^{s}(\frac{r_i^2+r_i}{2})$.

Using Lemma 5 the number of transitions for symbol S is $\frac{1}{2}n^2$ and the number of transitions for symbol $a \in \mathcal{A}$ is $\frac{1}{2}n^2 + n$.

The number of transitions then is $\mathcal{O}(\sum_{i=0}^{s}(\frac{r_i^2+r_i}{2}) + n^2 + n)$. □

Lemma 7. *Given a tree t with n nodes, the number of distinct nonlinear tree patterns which match the tree t can be at most $3^{n-1} + 2$.*

Proof. First, subtrees of any subtree of the tree t can be replaced by the special nullary symbol S and the tree template resulting from such a replacement is a tree pattern which matches the tree. Second, subtrees of any subtree of the tree t can be replaced by the special nullary symbol X and the nonlinear tree template resulting from such replacement is a nonlinear tree pattern which matches the tree. Given a tree with n nodes, the maximal number of subsets of subtrees that can be replaced by the special nullary symbol S, X occurs for the case of a tree t_3 whose structure is given by the prefix notation $pref(t_3) = a(n-1)\ a_1 0\ a_2 0 \dots a_{n-1} 0$, where $n \geq 2$. Such a tree is illustrated in Figure 6. In this tree, each of the nullary symbols $a_1 0, a_2 0, \dots, a_{n-1} 0$ can be replaced by nullary symbol S or X, and therefore we can create 3^{n-1} distinct tree templates which are tree patterns matching the tree t_3.

Third, the tree t itself and all its subtrees not containing the root are tree patterns which match the tree. Subtrees of the tree t must be the same so that the nonlinear tree pattern matched the tree in the first place. These gives 2 other distinct tree patterns.

Thus, the total number of distinct tree patterns matching the tree t can be at most $3^{n-1} + 2$. □

6. Processing more nonlinear variables in nonlinear tree patterns

Indexing for nonlinear tree pattern matching with more than one nonlinear variable can be done by a pushdown automaton created as a pushdown automaton for the intersection of languages. Automaton for two nonlinear variables would be constructed on the basis of two automata – each of them for one nonlinear variable. The disadvantage of this approach would be increasing space complexity.

Another approach is represented by a nondeterministic nonlinear tree pattern pushdown automaton $M_{nntp}(t)$ for one nonlinear variable that can be used as an indexing data structure also for nonlinear tree patterns with more variables. The idea is to compare which transitions of more runs of this single automaton were used to match the pattern. The input pattern needs to be modified because it contains symbols representing the nonlinear variables that the nondeterministic nonlinear tree pattern pushdown automaton can't handle.

Example 12. Consider a ranked alphabet $\mathcal{A} = \{a4, a3, a2, a1, a0\}$. Consider a nonlinear tree template p_4 over $\mathcal{A} \cup \{S, Y, Z\}$ $p_4 = (\{a4_1, X_2, X_3, Y_4, Y_5\}, R_{p4})$ over $\mathcal{A}$, where R_{p4} is a set of the following ordered pairs:

$$R_{p4} = \{(a4_1, X_2), (a4_1, X_3), (a4_1, Y_4), (a4_1, Y_5)\}.$$

Nonlinear tree template p_4 is illustrated in Figure 10.

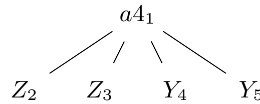


Fig. 10. Nonlinear tree template p_4 from Example 12

Nonlinear tree template p_4 can be decomposed to nonlinear tree templates for one nonlinear variable. These nonlinear tree templates will be over alphabet $\mathcal{A} \cup \{S, X\}$ and are illustrated in Figure 11. □

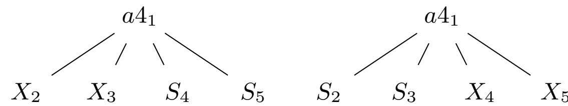


Fig. 11. Decomposition of nonlinear tree template p_4 from Example 12

In the beginning the algorithm decomposes a given nonlinear tree template to nonlinear tree templates of one nonlinear variable. Then, the accepting sequences of transitions are computed using nondeterministic nonlinear tree pattern pushdown automaton $M_{nntp}(t)$ and each decomposed nonlinear tree pattern. These accepting sequences of transitions can be used for filtering real occurrences out of the original tree template for more nonlinear variables.

Algorithm 8 Algorithm of nonlinear tree pattern matching with more nonlinear variables using nondeterministic nonlinear tree pattern pushdown automaton $M_{nntp}(t)$

Input: Nondeterministic nonlinear tree pattern pushdown automaton $M_{nntp}(t)$. Nonlinear tree pattern with more variables p, set $vars$ of variables used in the template p.

Output: Occurrences of the pattern.

Method:

1. Collection of nonlinear templates for one variable po is an empty collection.
2. For each variable var in $vars$ do:
   2.1. pd is a clone of nonlinear tree template p.
   2.2. Change a symbol in leaf nodes to nullary symbol S, where node label $l \in (vars \setminus var)$.
   2.3. Change a symbol in leaf nodes to nullary symbol of nonlinear variable X, where node label $l = var$.
   2.4. Add pd to po.
3. Set Occ contains $\{0, 1, \ldots, n\}$ where n is the size of the tree t.
4. For each nonlinear tree template pd in po do:
   4.1. Determine accepting sequences of transitions ts of tree template t using $M_{nntp}(t)$.
   4.2. Compute a set Occ_{pd} as set of $tnsl(q)$, where q is a target state of the first transition from all ts.
   4.2. Remove all items in Occ which are not in Occ_{pd}.
5. Output Occ.

6.1. Time and Space Complexity Analysis

Lemma 8. *Time complexity of accepting the nonlinear tree template for more nonlinear variables by nondeterministic nonlinear tree pattern pushdown automaton is $\mathcal{O}(v \times m + \sum run(pd) + \sum Occ_{pd})$, where v is the number of nonlinear variables, m is the size of the nonlinear template, $run(pd)$ is the time of locating all accepting transition sequences of each of decomposed templates pd in automaton $M_{nntp}(t)$ and Occ_{pd} is the size of occurrences of decomposed template pd.*

Proof. The nonlinear tree template needs to be decomposed to nonlinear tree templates for one nonlinear variable. This takes $v \times m$ time.

Occurrences of each nonlinear tree template from decomposed nonlinear tree template p are computed in time $\sum run(pd)$.

Composition of partial occurrences Occ_{pd} to Occ can be done in $\sum Occ_{pd}$ time. □

Lemma 9. *Given a tree t with n nodes, the number of distinct nonlinear tree patterns (with more nonlinear variables) which match the tree t can be at most $(2 + v)^{n-1} + 2$.*

Proof. First, subtrees of any subtree of the tree t can be replaced by the special nullary symbol S and the tree template resulting from such a replacement is a tree pattern which matches the tree. Second, subtrees of any subtree of the tree t can be replaced by the special nullary symbols of variables and the nonlinear tree template resulting from such replacement is a nonlinear tree pattern which matches the tree. Given a tree with n nodes, the maximal number of subsets of subtrees that can be replaced by the special nullary symbols of variables occurs for the case of a tree t_3 whose structure is given by the prefix notation $pref(t_3) = a(n-1)\ a_1 0\ a_2 0 \ldots a_{n-1} 0$, where $n \geq 2$. Such a tree is illustrated

in Figure 6. In this tree, each of the nullary symbols $a_1 0, a_2 0, \ldots, a_{n-1} 0$ can be replaced by nullary symbol of variables and therefore we can create $(2+v)^{n-1}$ distinct tree templates which are tree patterns matching the tree t_3.

Third, the tree t itself and all its subtrees not containing the root are tree patterns which match the tree. Subtrees of the tree t must be the same so that the nonlinear tree pattern matched the tree in the first place. These gives 2 other distinct tree patterns.

Thus, the total number of distinct tree patterns matching the tree t can be at most $(2+v)^{n-1}+2$. □

7. Some empirical results

We have implemented the nondeterministic tree pattern pushdown automaton and the nonlinear tree pattern pushdown automaton using the bit-parallelism technique, which was introduced in [27]. For our implementations of transitions reading symbols from $\mathcal{A}$ we use the same approach as it is used in the implementation of nondeterministic string suffix automata in [23]. The transitions reading the special variable nullary symbols must be treated in a special way. When processing these nullary symbols the algorithm takes indexes of each one in configuration bit vector of the bit–parallelism simulation and recompute these indexes according to transitions reading variable symbols in the simulated automaton. A unique id of subtree is stored for each location in the bit vector so that the matching of nonlinear tree patterns is possible.

Our implementations were written in Java programming language; all timings were conducted on a 2 GHz Intel Core i7 with 8 GB of RAM running OpenSUSE GNU/Linux version 12.1 and Java 1.6.0. In Figures 12, 13, and 14, it is shown that the running time for a given tree is linear with the size of the input (nonlinear) tree pattern. In Figures 15, 16, and 17, it is shown that the running time for binary trees is also linear in general but for very small input patterns there is a slowdown caused by recomputing the configuration vector of bit–parallelism for nullary variable symbols.

8. Conclusion

We have presented the tree pattern pushdown automaton and the nonlinear tree pattern pushdown automaton, a new kind of pushdown automata which represent a complete index of a given ordered tree for tree patterns and nonlinear tree patterns, respectively. We have discussed the time and space complexities and have shown timings of our implementations using the bit-parallel technique. The timings are similar to those for the existing bit-parallel implementation of nondeterministic string suffix automata.

Since the presented pushdown automata are input–driven, they can be determinised. However, the space complexities of their deterministic versions are open problems.

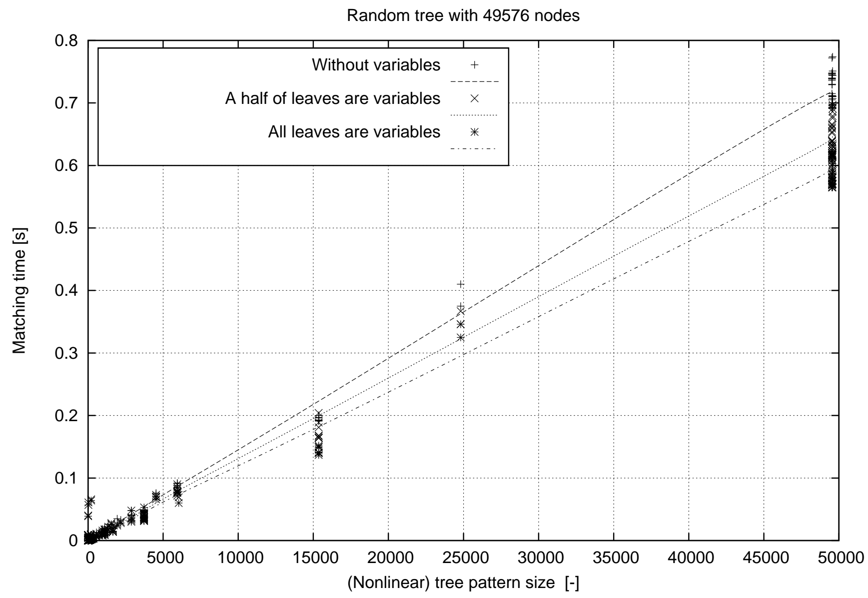


Fig. 12. Random tree with 49576 nodes

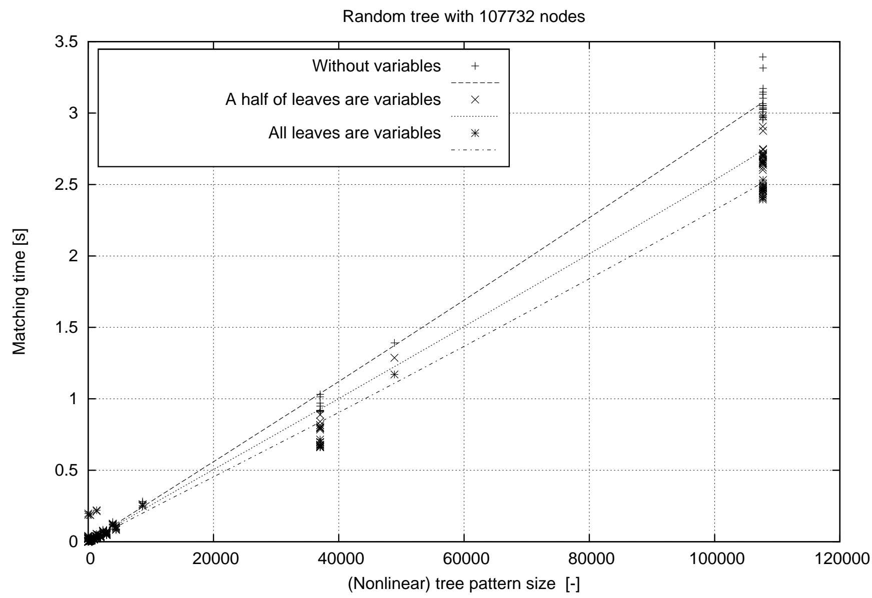


Fig. 13. Random tree with 107732 nodes

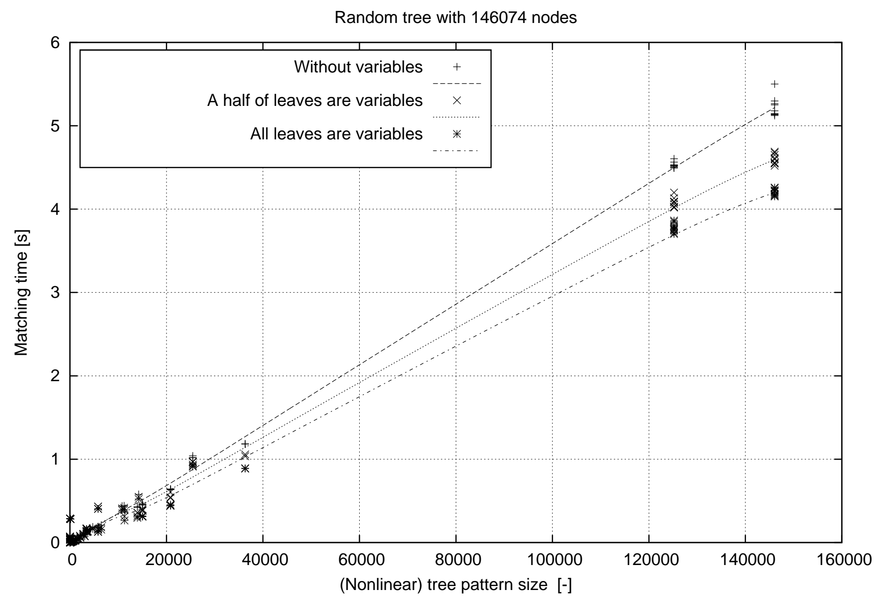


Fig. 14. Random tree with 146074 nodes

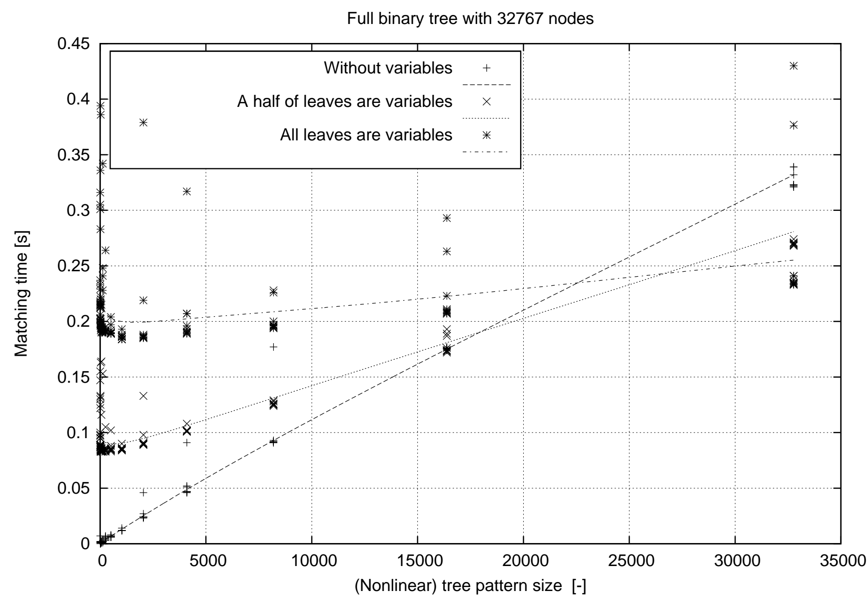


Fig. 15. Full binary tree tree with 32767 nodes

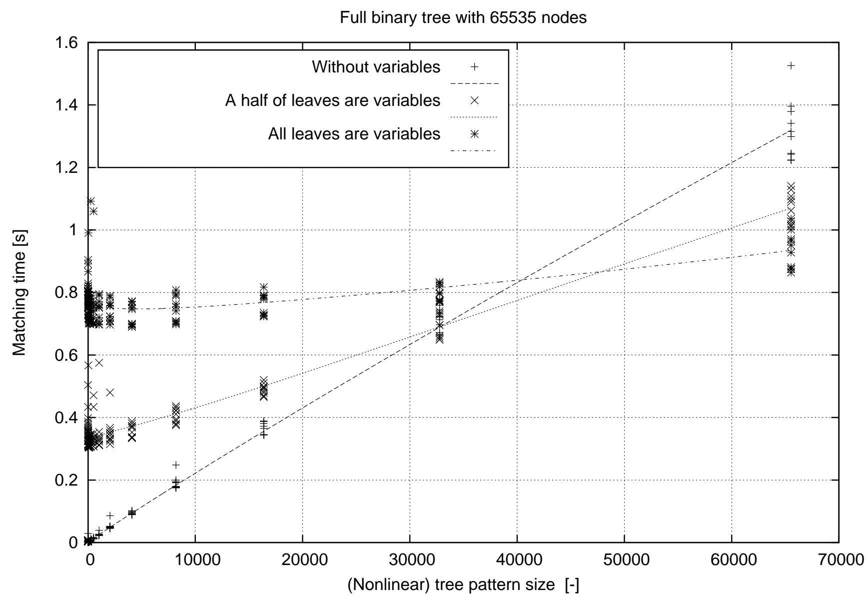


Fig. 16. Full binary tree tree with 65535 nodes

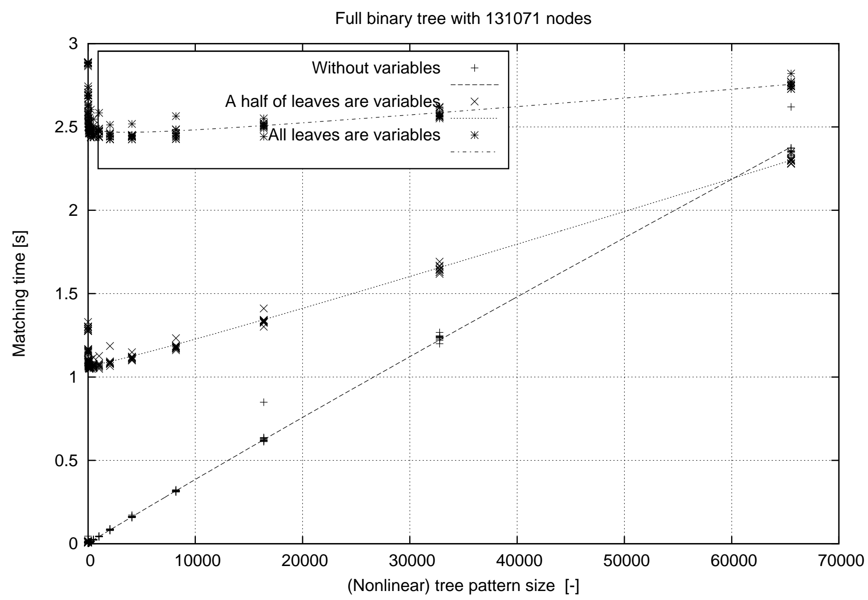


Fig. 17. Full binary tree tree with 131071 nodes

References

1. Aho, Alfred V.; Margaret J. Corasick: Efficient string matching: An aid to bibliographic search. In: *Communications of the ACM*, 18 (6), pp. 333340, 1975.
2. Alfred V. Aho and Jeffrey D. Ullman. *The theory of parsing, translation, and compiling*. Prentice-Hall Englewood Cliffs, N.J., 1972.
3. *Arbology www pages*, Available at: http://www.arbology.org, June 2012.
4. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M. T., Seiferas, J. I., 1985. The smallest automaton recognizing the subwords of a text. Theor. Comput. Sci. 40, 31–55.
5. Christou, M., Crochemore, M., et al., 2011. Computing All Subtree Repeats in Ordered Ranked Trees. In *String Processing and Information Retrieval*, Vol. 7024, pp. 338-343.
6. L.G.W.A. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit.* PhD Thesis, Department of Mathematics and Computer Science, Eindhoven University of Technology, Eindhoven, April 2008.
7. Crochemore, M., 1986. Transducers and repetitions. Theor. Comput. Sci. 45 (1), 63–86.
8. H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.
9. Crochemore, M., Hancart, C., 1997. Automata for matching patterns. In: Rozenberg, G., Salomaa, A. (Eds.), Handbook of Formal Languages. Vol. 2 Linear Modeling: Background and Application. Springer–Verlag, Berlin, Ch. 9, pp. 399–462.
10. Crochemore, M., Rytter, W., 1994. Jewels of Stringology. World Scientific, New Jersey.
11. Domenico Cantone, Simone Faro and Emanuele Giaquinta: A Compact Representation of Nondeterministic (Suffix) Automata for the Bit-Parallel Approach, In: *CPM 2010, LNCS 6129*, Springer, Berlin, 2010.
12. Toms Flouri, Jan Janousek, Borivoj Melichar, Costas S. Iliopoulos, Solon P. Pissis: Tree Template Matching in Ranked Ordered Trees by Pushdown Automata. In: *CIAA 2011, LNCS 6807*, Springer, Berlin, pp. 273-281, 2011.
13. Olivier Gauwin, Joachim Niehren: Streamable Fragments of Forward XPath. In: *CIAA 2011, LNCS 6807*, Springer, Berlin, pp. 3-15, 2011.
14. F Gecseg and M. Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3 Beyond Words. Handbook of Formal Languages, pages 1–68. Springer–Verlag, Berlin, 1997.
15. Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *J. ACM*, 29(1):68–95, 1982.
16. Hopcroft, J. E., Motwani, R., Ullman, J. D., 2001. Introduction to automata theory, languages, and computation, 2nd Edition. Addison-Wesley, Boston.
17. J. W. Klop. *Term Rewriting Systems*, Handbook of Logic in Computer Science, 1992.
18. Janousek, J. String Suffix Automata and Subtree Pushdown Automata. In: *Proceedings of the Prague Stringology Conference 2009*, pp. 160–172, Czech Technical University in Prague, Prague, 2009.
19. Janousek, J.: *Arbology: Algorithms on Trees and Pushdown Automata*. Habilitation thesis, TU FIT, Brno, 2010.
20. Janousek, J., Melichar, B. On Regular Tree Languages and Deterministic Pushdown Automata. In *Acta Informatica*, Vol. 46, No. 7, pp. 533-547, Springer, 2009.

21. Melichar, B. Arbology: Trees and pushdown automata. In: *LATA 2010 (LNCS 6031)*, invited paper, pp. 32-49, Springer, 2010.
22. Melichar, B., Holub, J., Polcar, J., 2005. Text searching algorithms. Available at: http://stringology.org/athens/, release November 2005.
23. Gonzalo Navarro, Mathieu Raffinot: A Bit-Parallel Approach to Suffix Automata: Fast Extended String Matching. In: *CPM, LNCS 1448*, Springer, Berlin, pp. 14-33, 1998.
24. R. Ramesh, I. V. Ramakrishnan. *Nonlinear Pattern Matching in Trees*, Journal of the Association for Computing Machinery, Vol 39, No 2, April 1992.
25. Smyth, B., 2003. Computing Patterns in Strings. Addison-Wesley-Pearson Education Limited, Essex, England.
26. Travnicek, J. , Janousek, J., Melichar, B. Nonlinear Tree Pattern Pushdown Automata. In*Proceedings of the FEDCSIS 2011*, IEEE Computer Society Press, pp. 871-878, 2011.
27. R. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In: *Commun. ACM*, 35(10), pp. 7482, 1992.

Jan Trávníček has been a PhD student at the Department of Theoretical Computer Science, Czech Technical University in Prague, Faculty of Information Technology. His research interests are focused on algorithms on trees (Arbology).

Jan Janoušek has been an associate professor at the Department of Theoretical Computer Science, Czech Technical University in Prague, Faculty of Information Technology. His research interests include algorithms on trees (Arbology), parsing algorithms, compiler construction, attribute grammars and formal languages and automata theory.

Borivoj Melichar has been a full professor at the Department of Theoretical Computer Science, Czech Technical University in Prague, Faculty of Information Technology. His research interests include algorithms on strings (Stringology), algorithms on trees (Arbology), parsing algorithms and compiler construction.

Received: December 20, 2011; Accepted: June 4, 2012.

DOI: 10.2298/CSIS120104026G

A Programming Language Independent Framework for Metrics-based Software Evolution and Analysis

Črt Gerlec[1], Gordana Rakić[2], Zoran Budimac[2], Marjan Heričko[1]

[1] Institute of Informatics
Faculty of Electrical Engineering and Computer Science
University of Maribor
Smetanova ulica 17, 2000 Maribor, Slovenia
{crt.gerlec,marjan.hericko}@uni-mb.si
[2] Department of Mathematics and informatics
Faculty of Science
University of Novi Sad
Trg Dositeja Obradovića 4, 2100 Novi Sad, Serbia
{goca,zjb}@dmi.uns.ac.rs

Abstract. Knowledge about different aspects of software quality during software evolution can be valuable information for developers and project managers. It helps to reduce the number of defects and improves the internal structure of software. However, determining software's quality and structure in heterogeneous systems is a difficult task. In this paper, a programming language independent framework for evaluating software metrics and analyzing software structure during software development and its evolution will be presented. The framework consists of the SMIILE tool for calculation of software metrics, extended with an analysis of software structure. The data are stored in a central repository via enriched Concrete Syntax Tree (eCST) for universal source code representation. The framework is demonstrated in a case study. The development of such a framework is a step forward to consistent support for software evolution by providing a change analysis and quality control. The significance of this consistency is growing today, when software projects are more complex, consisting of components developed in diverse programming languages.

Keywords: Software evolution, software development, software quality, software structure, software metrics, syntax tree

1. Introduction

From the beginning of the application of software engineering, engineers have been striving to develop quality and maintainable software products. Therefore, software evolution and its quality have become an important research discipline. Early versioning systems like the Source Code Control System made it possible to record the sequential versions of software products [41]. Such software

history has been important to understand what, where and when a change was applied. Beside versioning systems, software quality measures have been researched. The first published book that describe software metrics appeared in 1976 [20] but the first attempts at applying software metrics had already taken place in the late 1960s [17]. With the spread of software metrics, a need for the appropriate storage of such data grew. In 1993, Pfleeger described the importance of data collecting and determined their success in a metrics program [37].

Various approaches have been used to analyze software evolution. The majority of them are programming language specific. Their meta-models are not general and do not enable a structural software comparison between different systems. Thus, making a comparison between the structural software evolution of two systems or of two components with the same system that are written in different programming languages (e.g. Java and C#), is not possible. A similar problem can be found in the field of software metrics. Existing approaches that define the metrics and its algorithms are programming language specific. Furthermore, the algorithms usually differ between the tools. Thus, a software metric comparison in heterogeneous systems is not accurate.

The purpose of this study is to deal with the problems surrounding programming language dependent frameworks and approaches that describe software metrics and software structure. Thus, a general framework that allows a programming language independent representation and evaluation of software artifacts has been developed. It consists of three major components. The first component of the framework is a language independent meta-model for representing a source code structure. The main purpose is to provide sufficient data for a further evolutionary analysis based on software structures (e.g. detecting structural source code changes between sequential software versions). The second one is the SMIILE (Software Metrics Independent of Input LanguagE) tool. Its main advantage is to define an universal implementation of metric algorithms (e.g. algorithm for the cyclomatic complexity) built upon the meta-model. Both components are based on the enriched Concrete Syntax Tree (eCST) that represents an internal representation of source code. The eCST is built on "universal" nodes that are common for all programming languages. However, in order to store the software artifacts and conduct a deeper analysis, an appropriate repository is needed. To fulfill this demand, a specific repository was build and integrated as the third component in the framework.

In our case study, the application of the framework will be shown. Its goal is to apply the framework (i.e. meta-models) in practice by using the SMIILE tool for defining and calculating software metric values and by using structural source code representation from different programming languages.

The contribution of this paper is the development of a programming language independent framework for metrics-based software evolution and analysis. This goal was achieved by (1) adjusting the eCST concept in order to support a language-independent source code structure representation that en-

ables evolutionary analysis based on software structure and by (2) integrating the SMIILE tool with a central repository that supports eCST.

This paper is organized as follows: The background, needed for understanding the study, and the motivation are described in section 2. Then, the preliminary works are introduced in section 3. The programming language independent framework for metrics and structural analysis in software evolution is briefly described in section 4. In the section 5, the framework application is presented with a case study. The validity and limitations of this research are stated in section 6. In the section 7, state-of-the-art tools and approaches for analyzing software metrics and software structure are introduced. In the last section, the conclusion and ideas for future work are provided.

2. Background and motivation

This section describes three important notions of the study (i.e. software evolution, software metrics and software repository) and the motivation.

2.1. Software evolution

The field of software evolution has become an interesting area over the last decade, leading to an increase in the amount of research on the subject [14]. Lehman et al. [31] describes two perspectives on software evolution. The first perspective focuses on the questions of "what and why" and describes the nature of software evolution and its properties. On the other hand, the second perspective is focused on the word "how" and covers areas like the theories, abstractions, languages, activities, methods and tools required to evolve software.

Software evolution could also be understood as continuous adaptation. Software changes, that are caused by an adaptation process, are usually partitioned into three general classes [33]. The first class includes corrections that tend to be fixes of source code errors. However, there are also some other error fixes that are related to software design, architecture and requirements. The next class consists of improvements. They tend to include things like increases in performance, usability, maintainability, etc. The last class comprises enhancements that represent new features or functions that are visible to the users of the end system.

Software systems evolve continuously in order to satisfy all users' needs and requirements. The research in [26] showed that the software history is a good indicator for its quality. Therefore, it is vital for companies to ensure mechanism that tracks the changes during the development in order to minimize the risk for potential new bugs.

Software changes are part of software evolution. Thus, it is important to analyze these changes from a structural and qualitative point of view and then compare the results.

Structural source code changes are constant during software development. They are usually made when new functionality is added to the existing software product or during the updates. Moreover, changes are also made in refactoring and debugging processes. In this paper, a structural source code change is defined as an object-oriented change on a class (e.g. the add/remove method) between two sequential versions. The examples of structural source code changes are:

– add parameter, field and method,
– remove parameter, field and method,
– hide and unhide method,
– rename method,
– move attribute, method and class,
– extract superclass, interface and class,
– pull up field and method,
– push down field and method, and
– inline class.

In the sense of software evolution, our study focuses on defining a programming language independent meta-model that is based on the time (i.e. version) component. Its intent is to collect sufficient data about software structure and its quality properties. Such a meta-model enables further evolutionary analysis upon the collected data. However, the change detection process uses several rules in order to identify structural changes between two source code versions. Each rule represents one change type (e.g. add method) and usually accepts two parameters. For example, the first parameter is metadata for a class in version n and the second parameter represents the same class in the next version (i.e. n+1). If the metadata for the same class in two sequential versions fulfills the demands of the rules, the change type that the rule represents, was used on the class. Even though the change detection process has already been implemented, it is out of the scope of this paper.

2.2. Software product metrics

The measuring and continual monitoring of a software product is crucial for success in the software development process. From this perspective, software metrics, the software metrics tool and the software metrics repository are crucial notions.

Software metrics can be defined as numerical values that reflect the properties of a software development processes and software products [34]. There are numerous categorizations of software metrics but when considering the measurements, target metrics can be divided into three main categories: product metrics, process metrics and project metrics [29]. In the rest of the paper, we will deal with product metrics and especially code metrics as a sub-category of product metrics. First, we will specify some of the product metrics used in the rest of the paper:

- Cyclomatic Complexity (CC) - reflects structure complexity based on control-flow structures in the program.
- Halstead Metrics (H) - reflects the complexity of the program based on number of operators and operands.
- Lines of Code (LOC) - represents the length of the source code expressed in the number of lines of source code. It is common to differentiate between the number of lines of comment (CLOC), source code (SLOC), etc.
- Object Oriented metrics (OO) - the family of metrics related to the object orientation of software. On the other hand, the term *design metrics* is often used and usually describes metrics related to characteristics of object oriented development and design. However, some examples of the metrics used in this paper are:
  - Number of Classes (NOC) - reflects the number of classes contained in the package, namespace, project, etc.
  - Number of Interfaces (NOI) - reflects the number of interfaces contained in the package, namespace, project, etc.
  - Number of Methods (NOM) - reflects the number of methods declared in the unit (class, interface, etc).
  - Number of Properties (NOP) - reflects the number of properties declared in the unit (class, interface, etc).
  - Number of Attributes (NOA) - reflects the number of attributes declared in the unit (class, interface, etc).

Nowadays, various software metrics tools are used for automatic calculations of software metrics. However, achieveing accuracy of the gathered metric values and the appropriate interpretation of extracted data is often the hardest step.

In section 7, problems in the area of consistent and systematic application of software metrics will be presented. During the exploration, the strong dependency of the applicability of software metrics on an input programming language was recognized as one of the main weaknesses in this field. Introducing an enriched Concrete Syntax Tree (eCST) for intermediate representation of the source code resulted in a step towards programming language independence.

2.3. Software repositories

In order to perform a detailed measurement and analysis and interpretation of numerous software metric values, a repository is needed. Its aim is to collect [24], store and enable access to a wide range of metric values (e.g. product, process and resource metric values) collected from software products, software development processes and project management tools. The collected data, extracted with different tools, helps project leaders and development teams get a better overview of a project.

Software repositories have been recognized as an important tool in the past. Carnegie et al. [27] suggested that software organizations should implement systems to define, collect, store, analyze and use process data. Furthermore,

Basili [12] suggested that data analysis routines should be implemented in order to extract derived data from the row data. Then, all collected data should be stored in a computerized database. In the study conducted by Goeminne et al. [22] the term "repository" is defined as follows: "A data source containing information that is relevant to the software product or process, and that can be accessed and modified by different persons by using their identity." Repositories collect various properties of software systems (e.g the version of the source code). As mentioned earlier, metrics repositories store data about a software product (i.e. software metrics) while other repositories store different data (e.g. properties of software processes). However, with historical insight over the software properties, users become familiar with changes that were made over time. With such knowledge, users are able to predict changes in the future and act if the negative trend is detected. Thus, the establishment of software repositories is sensible in organizations.

2.4. Motivation

Related research has also shown that there is no fully consistent tool support for measurement and analysis during software development and maintenance. The tools used for these purposes have some limitations (e.g. limited programming language support, weak and inconsistent usage of metrics and/or testing techniques, etc).

Large software systems are written in several programming languages. In order to ensure a high level of software quality, we have to know the condition of every part of a system. Furthermore, in order to evaluate such systems, different tools have to be used. However, these tools usually provide inconsistent values for software metrics [36], [32], [43] and therefore, a comparison between different parts of a system, written in different programming languages, is not applicable.

In the field of software evolution, which enforces techniques such as advising, recommending and the automating of refactoring and reengineering, solutions that are based on a common intermediate structure can be a key supporting element. This support could be based on metrics, testing and deeper static and structure analysis. The development of such support would introduce new values into the field of software engineering. For all of these reasons, a proposed universal tree could be an appropriate internal representation applicable toward all stated goals. Universality of internal structure is important for meeting consistency in all fields.

By realization of this idea a key benefit could be made from language independence of eCST and its universality and broad applicability.

3. Preliminary work

In this section, the preliminary work for developing a tool for change analysis during software evolution will be described. Furthermore, a description of eCST

and the original idea of an application of underlying trees in the development of the SMIILE tool will be presented.

3.1. A tool for mining software repositories

The tool for identifying structural source code changes was presented in [19]. Its aim is to extract data from software repositories (e.g. subversion) and store them into the meta-model in order to identify structural source code changes between sequential versions. The change identification process is based on a set of change rules. They are applied between different versions represented by the meta-models. If demands of the rule are fulfilled, the change type is found. In this study, 26 different rules for detecting change types were used. The results showed that the tool could be used to analyze source code changes in software repositories. On the other hand, the tool also has some limitations. The main weakness is a programming language dependency. The current tool only supports C# and VisualBasic programming language. The main problem is direct relation between source-code and the meta-model. In order to overcome this limitation, a universal intermediate representation of source code is needed.

3.2. Introducing of eCST

The motivation for introducing eCST as a new intermediate representation of the source code is described in section 2.4.

Originally, tools used a Concrete Syntax Tree (CST) for the representation of source code. This tree is usually an intermediate product of a parser generator. It takes language grammar as an input and returns a language scanner and parser as output. The grammar rules determine the manner in which the syntax tree, as an intermediate structure, will be generated [23].

A CST represents concrete source code elements attached to a corresponding construction in a language syntax. Although this tree is quite rich, it is still unaware of sophisticated details about the meaning of syntax elements and their role in certain problems (e.g. algorithms for the calculation of software metrics). We enriched CST by adding universal nodes to mark elements to become a recognizable independent for input programming language. The catalog of universal nodes used in the prototype can be found in the appendix, in table 8.

To illustrate this technique and to achieve the independence of a programming language, we provide the following simple example [38]. It illustrates the problems in the calculation of a CC metric via the predicate counting method.

The simple loop statement (REPEAT), written in Modula-2, and the corresponding one (do-while), written in Java, are stated as in table 1.

Although the given statements have different syntax, they express the same functionality: some statements in the code will be repeated until parameter i becomes greater than parameter j. In addition to the different syntax, a condition for leaving the loop is oppositely stated. First, the condition expresses what

Table 1. Loop statements

REPEAT ... Some statements ... UNTIL(i > j);	do{ ...Some statements... }while(i <= j);
(a) REPEAT (Modula-2)	(b) do-while (Java)

condition should be fulfilled to leave the loop, while the second one states the condition to continue looping.

Simplified syntax trees representing these given statements are illustrated in Figure 1.

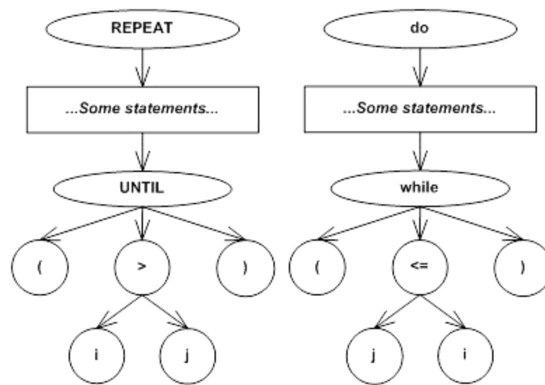


Fig. 1. Simplified CST for REAPEAT-UNTIL (left) and do-while (right) statements

For the implementation of a CC algorithm, a *REPEAT* and a *WHILE* loop have to be recognized and then increment the current CC value by 1. It is clear that by using CST for source code representation, two implementations or at least two conditions to recognize these loops in the tree are needed. By adding universal nodes (i.e. *LOOP_STATEMENT*) as a parent of sub-trees, that represent these two segments of source code, the goal by only one condition in the implementation of the CC algorithm is met. A universal node, CONDITION, was also added in order to mark the condition for leaving the loop repetition (Figure 2).

By adding all the needed universal nodes [40], the algorithms for the CC metric could be implemented independently of a programming language. The only requirement is that there is a language grammar to modify and generate an appropriate parser that is then used for generating eCST.

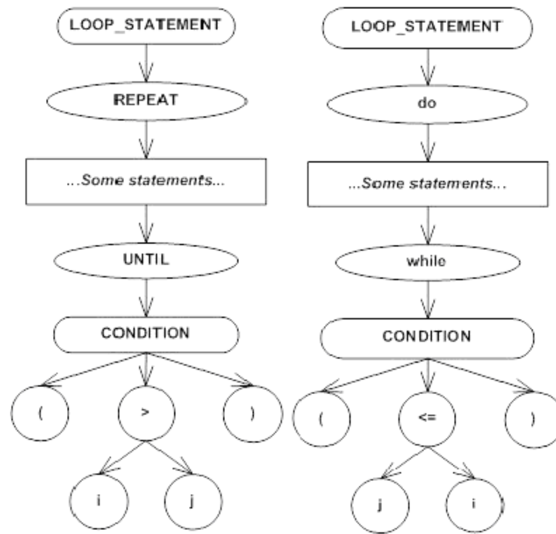


Fig. 2. Simplified eCST for REAPEAT-UNTIL (left) and do-while (right) statements

Enriching the CST by adding universal nodes is done at a grammar level. In the input language grammar, in the corresponding rule, we simply build in an imaginary node. For example, in the rule where a control structures (e.g. if, case, switch, etc.) are defined, an appropriate universal node is created. In the ANTLR [9] (the compiler generator that was used in our study) this is possible via a simple extension of the appropriate rule in the form of a declaration of a new node that is automatically added to the syntax tree during its creation process. The list of universal nodes required for implementing CC algorithm is given in [40], while a full description of the eCST, generating process and storing is presented in [39]. The possible broader applicability of eCST in different software engineering fields is described in [38].

It should be noted that the CC metric was chosen as a characteristic example for presenting the usefulness of the eCST in the sense of language independence. The LOC metric is less sensitive to the syntax of a programming language. However, a generated eCST is stored in an XML file based on a recursive definition [39]. Each node contains information about the location of the element in the source code (line and column), its text and node name, an index of the element and nodes that contain children (i.e. sub-trees). In such a structure, we can find all the necessary data for calculating the LOC metric. From the first element in the underlying sub-tree we can identify the starting line number, and from the last element in the last sub-tree we can identify the ending line number. Using the first and last line, we can easily calculate the LOC metric for a certain unit.

3.3. The SMIILE Tool

The SMIILE tool is a software metrics tool with the following general goals:

- independence of an input programming language,
- broad set of software metrics supported and
- support of software metrics history.

For an input source code, the SMIILE tool will execute the steps in two phases (Figure 3).

- Phase 1:
  - Recognition of the input programming language based on the input file extension.
  - Reading data about the language.
  - Calling an appropriate scanner and parser. Scanner and parser is generated by an ANTLR parser generator [9] from grammar containing rules for extending CST to eCST.
  - Tree generation that represents the provided source code and translates it into XML format. This process forms the basis for applying different algorithms (e.g. algorithms for the calculation of software metrics)
- Phase 2:
  - Reading the tree structure form XML to eCST.
  - Calculating software metric values.
  - Storing software metric values in XML.

The SMIILE was used on several different programming languages (object-oriented Java and C#, procedural Module-2 and Pascal and legacy COBOL). Furthermore, several metrics were used and implemented. We have chosen two of them (LOC and CC) in order to demonstrate the universality of the model. The LOC metric calculation algorithm is executable on a lexical level, while the CC metric is sensitive to input language syntax (illustrated by the example in the previous subsection). To implement algorithms for calculating the CC by predicate counting and at the same time to meet the language independence of this implementation, we introduced universal nodes for each element of language syntax figuring in the algorithm. However, the eCST is designed in such a way as to support any programming languages.

The catalog of universal nodes used in the implementation of the CC metric is specified in [40]. The full catalogue of universal nodes used in the current prototype of SMIILE tool can be seen in the appendix (in table 8). In the following table (table 2) we will only introduce those universal nodes referred to in this paper.

The storage of the SMIILE tool's source code representation and metrics history can be divided into two parts. In the first part, the eCST representation of a source code is stored in an XML file that represents the basis for metric calculations. In the next part, software metrics are calculated and stored in a separate XML file that contain metric values. In other words, for each version of a software the SMIILE tool generates two xml files (i.e. eCST representation and metric values). However, the aim of this study is to integrate software metrics history with a repository.

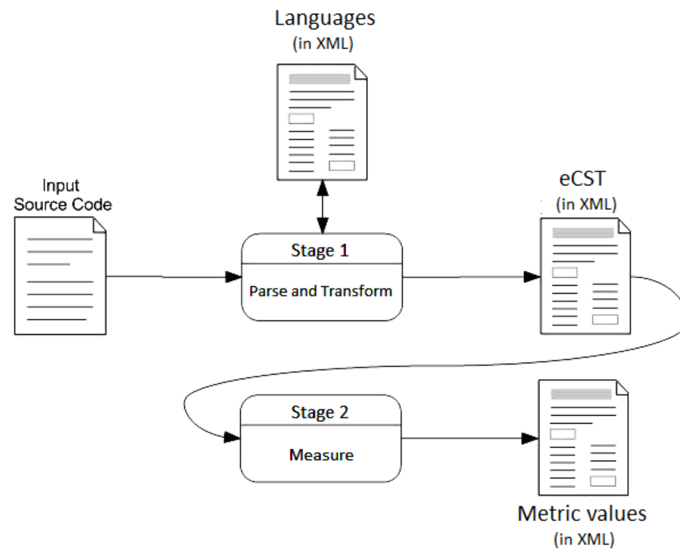


Fig. 3. SMIILE Tool Architecture

Table 2. Catalog of universal nodes used for integration eCST with the framework.

Universal node	coresponding element of language syntax
PACKAGE_DECL	package, workspace,...
CONCRETE_UNIT_DECL	class, implementation module,...
ABSTRACT_UNIT_DECL	abstrat class, etc.
INTERFACE_UNIT_DECL	interface, definition module,...
EXTENDED_BASE_UNITS	extended class
IMPLEMENTED_INTERFACE_UNITS	implemented interface, corresponding definition module,...
ATTRIBUTE_DECL	attribute, field,...
PROPERTY_DECL	property
FUNCTION_DECL	method, procedure, function
PARAMETERS_DECL	parameters of the method, procedure, function,...
NAME	name of any element (unit, function, attribute,...)
TYPE	type of any element (unit, function, attribute,...)

4. Framework for analyzing software evolution

In this section, the programming language independent framework for analyzing software structure and metrics is presented (figure 4). In order to overcome the existing problems of meta-models and approaches for software evolution analysis, our framework focuses on the following aspects:

- A programming language independent framework for analyzing software evolution built on the eCST.
  - An eCST-based meta-model that provides sufficient meta-data for analyzing software structure and its changes.
  - An eCTS-based meta-model for representing software syntax that enables metric calculations based on universal nodes.
- A software repository that stores meta-data and enables further analysis.

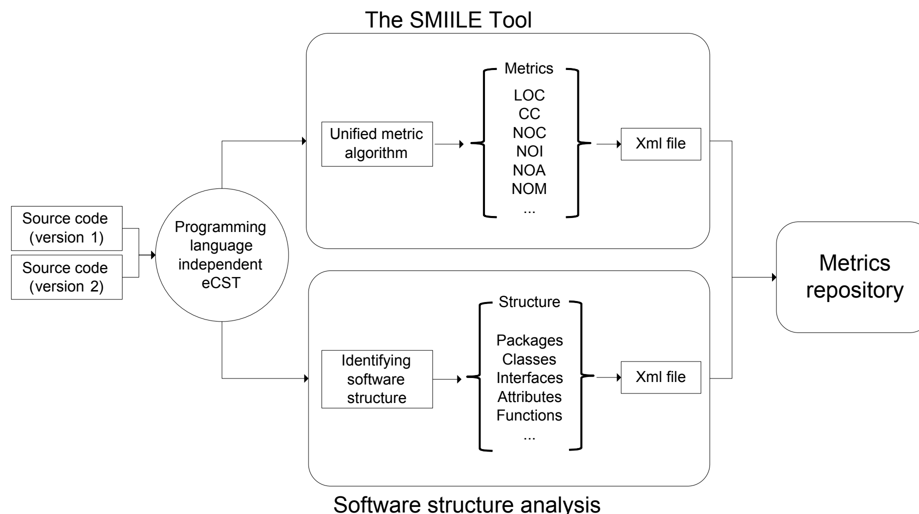


Fig. 4. The programming language independent framework for analyzing software structure and metrics.

The framework is built upon the eCST that includes "universal" nodes, which are common for various programming languages. It consists of three components. The first component is responsible for defining time in the software development life cycle and is represented with the *version* entity. The second component deals with a software structure. It describes the structure of software at a certain time in the software development process and is represented in a dedicated part of the eCST. The last entity deals with software *metrics* and provides a mechanism for the software quality analysis. Similar to a structure definition,

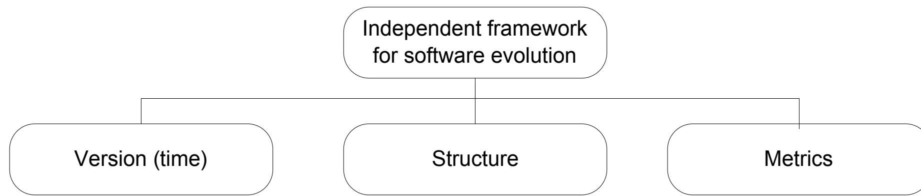


Fig. 5. Three fundamental components of the framework.

the properties needed for evaluating software metrics are also defined in the special part of eCST. All tree core components are shown in Figure 5.

The version entity is a central part of our framework. It determines the certain state of software in a development life cycle. In order to provide meaningful data to our meta-model and to enable a reasonable analysis in the future, the version entity consists of two elements: *DateTime* for determining the date and time of the software snapshot and *VersionName*, which describes the version with a unique identifier (e.g. version number, release name). The latter is usually specified by the product owner (e.g. ”my software version 1.5.3”).

4.1. Programming language independent meta-model for describing software structure

In order to detect structural software changes between software versions, a special meta-model is needed. However, one of the purposes of this study was to build a programming language independent meta-model that ensures sufficient data and represents the basis for approaches that deal with techniques for detecting structural software changes.

The basis for defining this meta-model were changes defined by Fowler et al. [18]. The authors actually defined refactoring techniques that are similar to source code changes. By their definition, refactoring improves the internal structure of a software system via source code changes. On the other hand, new functionality is not allowed to be added to the end system during the refactoring process [18]. However, each refactoring is a source code change while the opposite relation is not true. Our programming language independent meta-model is defined to provide sufficient data for detecting the changes below.

– Add parameter, field and method
– Remove parameter, field and method
– Hide and unhide method
– Rename method
– Move attribute, method and class
– Extract superclass, interface and class
– Pull up field and method
– Push down field and method
– Inline class

Programming languages differ from each other. Besides object-oriented constructs that are similar, they also have some that are unique or different between languages. For example, Java and C# use properties. In Java, they are implemented with get and set methods. On the other hand, the C# programming language has a unique construct for the same functionality. However, the idea behind a source code representation is to take a snapshot of the software's structure as it is. No additional logic is used that could identify, for example, properties (i.e. getter and setter methods) in Java code. In order to cover as many programming languages as possible, additional changes have been added to the list above. Additional types are written bellow.

- Add property
- Remove property
- Move property
- Pull up property
- Push down property
- Method body change

In order to provide sufficient data for approaches that deal with identifying code changes, the appropriate extent of data should be extracted from the raw source files. This extent of data is called the information level and represents the minimal amount of data that is necessary in order to identify structural changes. However, changes from the list were analyzed and for each change an information level for detecting it from two sequential versions were defined.

For example, figure 6 shows the extract interface change type. In the *version 1*, the class *Employee* has 3 methods: *getRate, getName and getSurname*. After the change process in *version 2*, the *Employee* class implements a new interface *Billable*. The new interface contains a method *getRate* that was "transferred" from the *Employee* class.

To be able to automatically detect such changes from software history, the appropriate data (i.e. information level) should be stored into the meta-model. The information level for the extract interface type is shown in table 3.

Table 3. Information level for extract interface change detection.

	Version 1	Version 2	State (Added/Deleted/Updated)
Package	x	x	x
Class	x	x	x
Base class	-	-	-
Interface	x	x	x
Method	x	x	x
Properties	-	-	-
Attributes	-	-	-

The adequate information for detecting the extract interface change type between *version 1* and *version 2* are *package*, *class*, *interface* and *method*.

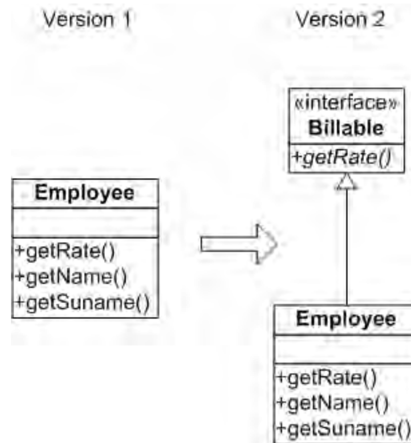


Fig. 6. The extract interface change type.

In addition, the information about the state of this construct is also needed. The state defines if a construct was added, deleted or updated in an observed version. This information is needed in order to identify parts that were actually changed.

The information level needed to fulfill all demands of our list of changes are shown in the table 3. The root element is *concrete_unit_decl* which represents the main entity. From an object-oriented perspective, this element corresponds to the *class* construct. The next important elements are *unit_state*, *name* and *package_decl*. The first one describes the state of an entity in the observed version and identifies if it was added, deleted or updated. For example, if a method is added to an existing class in a version, the *unit_state* node will be set to *updated*. On the other hand, the latter two elements identify a name and a package for a *concrete_unit_decl*. *Extended_base_units* and *implemented_interface_units* represents the lists of extended classes and implemented interfaces of the observed entity.

An additional set of meta-data elements are *attribute_decl*, *property_decl* and *function_decl*. The first two elements represent the attributes and properties of a class. They are composed by two sub-elements that identify their types (*type*) and names (*name*). The last element is *function_decl*, which represents the functions or methods. It consists of the function name (*name*), the access element (*access_decl*), the return type that is represented by *type*, a list of parameters (*parameters_decl*) with underlying elements (*type*, *name*) and tokens that describes the function body. However, tokens are represented in the eCST and therefore they are extracted from it.

Interfaces are described with the *interface_unit_decl* node, which is similar to *concrete_unit_decl*. However, the interface declaration has less universal nodes.

4.2. Software repository for storing framework meta-data

A software repository is a fundamental tool for analyzing software evolution. Software products are represented by raw source code that has to be reshaped in order to achieve a deeper analysis. Special techniques are required to extract meta-data from software products and store them into central storage. In this research, a special software repository was built. Its intent is to fully support the whole process for analyzing the software structure and evaluation of software metrics. In addition to this, a special mechanism for evaluating software quality has been added. The mechanism based on a composed metric, called the Quality index [25].

The repository consists of several modules (Figure 7).

– Basic metrics list
– Composed metrics list
– Defining a Quality index
– Data import
– Metric values presentation through versions

The fundamental module of our repository is a *basic metric list*. Its role is to mark each metric with a unique internal identifier. If the metric is not defined, it is skipped during the data import process. The reason for this is to unify metric names across the repository and to ease further analysis. The next module is *composed metrics list* where custom metrics are defined. For example, C is a composed metric derived from A and B (basic metrics). Furthermore, the repository enables custom calculations using simple mathematical operations. Currently, the repository supports all basic arithmetic operations (e.g. addition, subtraction, multiplication, divisions). However, if we would like to calculate the ratio between the CC and the LOC, then we can manually define the ratio metric as follows: Ratio = CC/LOC. The repository takes all the necessary metrics' data from out of storage and then applies mathematical operations on them. Beside the composed metrics, the repository supports calculating a quality index and its underlying parameters.

The data import process transfers data from outer sources into the repository. The process uses a special mechanism that transforms the original structure of a source into the internal (xml notation). For example, meta-data (represented in meta-models) goes through the process of reshaping its structure in order to import the data into the repository. On the other hand, the repository also allows for the importing of data from third-party sources. The only requirement is to provide the data in an appropriate structure. However, such a procedure guarantees that the data is always imported in the same way.

The last module is responsible for visualizing metric values. The repository supports the storage of software metrics for different project versions. Thus, the metric values could be historically analyzed and shown on a graph. With such a representation, researchers and project managers can analyze the history of the metrics and observe their changes between project versions. For example, if the rise of the cyclomatic complexity is recognized, additional actions may be required to lower the complexity in the next version.

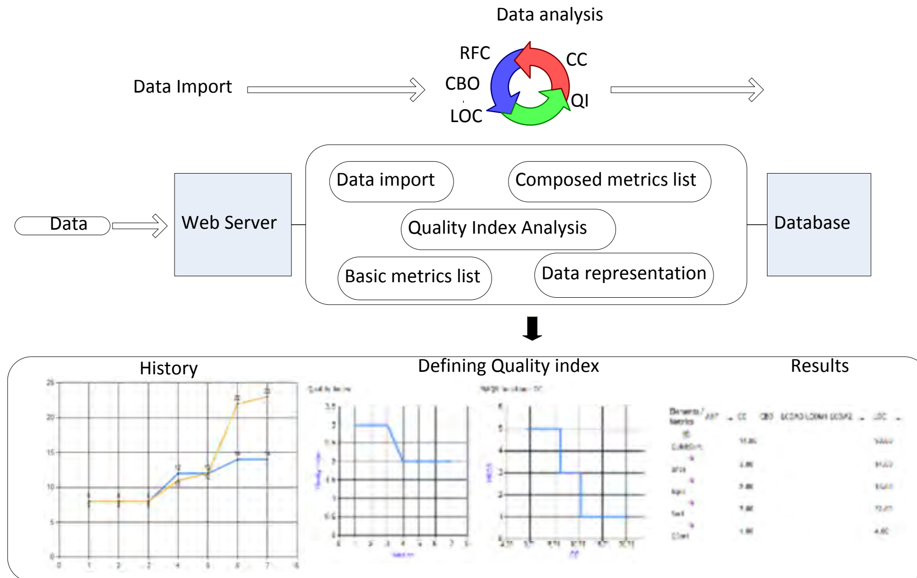


Fig. 7. The metrics repository framework.

5. A case study

In this section, the applicability of the framework is shown. The application is divided into two parts. In the first part, the programming-language independent mechanism for extracting meta-data from raw source code files is described. The mechanism is based on the meta-model that represents a software structure. In the second part, a general approach for calculating software metrics is presented. However, the meta-model with universal nodes is the frame for the software metrics' evaluation.

5.1. Extracting meta-data from a source code

In the case study, two object-oriented programming languages (Java and C#) were used in order to show the applicability of the meta-model for representing software structure. To extract data from raw source code and to fulfill demands of the meta-model, a special tool was developed. The fundamental part of the tool is a mechanism for analyzing source code files. It uses the ANTLR language tool[9] for language recognition and manipulation. However, Java and C# grammars were used in order to construct an abstract syntax tree from the source code. Then, a tree is used to identify meta-data and to fulfill demands of the meta-model.

The example below shows two classes implemented in C# and Java programming language. Both classes (*Student*) have the same behavior and the only distinction is the programming language syntax.

```
/* C# example */
namespace CSharpExample
{
  using System;
  public class Student : Person, IStudent
  {
    private Mark _mark;
    public int StudentNumber { get; set; }
    private decimal CalculateAverageMark(int level){
    ...
    }
  }
  ...
}
```

```
/* Java example */
package JavaExample;
import java.util.ArrayList;
public class Student extends Person implements IStudent
{
  private Mark _mark;
  private int studentNumber;
  public int getStudentNumber(){
  ...
  }
  public void setStudentNumber(int studentNumber){
  ...
  }
  private decimal calculateAverageMark(int level){
  ...
  }
}
...
```

The difference between the classes is the implementation of the *student number* property. In the C# programming language, a special construct is used in order to describe the property. On the other hand, in the Java programming language, the property is implemented with the attribute *studentNumber*, which actually stores a value, and two additional methods. The first method is *getStudentNumber* that returns the value and the second method is *setStudentNumber* that sets the value.

The table 4 shows the meta-models for the extracted source files. However, the meta-models' elements are similar for both languages. The *unit_state* elements were set to 'added' because the analyzed classes were treated as new

Table 4. The meta-model for classes implemented in C# and Java.

	C# class	Java class	is eCST node
UNIT_STATE	Added	Added	no
PACKAGE_DECL	CSharpExample	JavaExample	yes
CONCRETE_UNIT_DECL	x	x	yes
NAME	Student	Student	yes
EXTENDED_BASE_UNITS	Person	Person	yes
IMPLEMENTED_ _INTERFACE_UNITS	IStudent	IStudent	yes
ATTRIBUTE_DECL	x	x	yes
- TYPE	Mark	Mark/ int	yes
- NAME	_mark	_mark/ _studentNumber	yes
PROPERTY_DECL	x	/	yes
- TYPE	int	/	yes
- NAME	StudentNumber	/	yes
FUNCTION_DECL	x	x	yes
- NAME	CalculateAverageMark	CalculateAverageMark/ getStudentNumber/ setStudentNumber	yes
- ACCESS_DECL	private	private/ public/ public	yes
- RETURN_TYPE	x	x	yes
- - TYPE	decimal	decimal/ int/ void	yes
- PARAMETERS_DECL	x	x	yes
- - TYPE	int	int/ - / int	yes
- - NAME	level	level/ - / studentName	yes
- TOKENS	...	... / ... / ...	no

classes in the case study. Similar to previous elements, the *name* elements also have the same value for both languages. On the other hand, the *package_decl* elements are different and are set to 'CSharpExample' and 'JavaExample'. The extended class (*base_unit_decl*) and implemented interface (*interface_unit_decl*) have the same values for both meta-models ('Person' and 'IStudent'). Both classes have one attribute and therefore the *attribute_decl* elements are set with the name '_mark' and type 'Mark'. As expected, the differences are in the definition of the *property_decl* and *function_decl* elements. The C# programming language has special constructs for properties. Therefore the *property_decl* element is set to 'int' for the *type* and 'StudentNumber' for the *name* sub-element. In Java, one additional *attribute_decl* and two additional *function_decl* are defined. Besides the 'getStudentNumber' and 'setStudentNumber' definitions in *function_decl*, the 'calculateAverageMark' method is also defined. It has the 'private' access modifier (*access_decl*), void return type (*return_type*) and one method parameter (*parameter_decl*) with the type 'int' and name 'level'. A similar method is also defined in the meta-model that is the basis of the C# programming language.

The difference between the languages related to properties are reflected in the calculated values of the LOC metric. Therefore, the source code written in Java is approximately twice as long as one written in C# (tables 5 and 6). Furthermore, the properties in C# account for the difference in the number of attributes, properties and methods (the so-called functions in the universal model). These values are presented in table 6. The number of classes (i.e. concrete units) and interfaces (i.e. interface units) are the same in both examples 5. In this case study, the CC metric has no importance because the CC values for all methods are equal to one.

Software metrics have the same relation between different programming languages. For example, the CC metric counts the branches in a source code. If the source code consists of more branches, its complexity will be higher. Therefore, a high complexity will always be indicated with a high CC value in all programming languages and the opposite relation does not exist.

Table 5. Metrics related to the workspaces.

	C#	Java
Number of concrete units	2	2
Number of interface units	1	1
LOC	32	66

When the source files are analyzed, the meta-models and the results are exported in an xml format. Then, only the final step is required. This step imports the prepared meta-models into the software repository.

Table 6. Metrics related to the class Student.

class Student	C#	Java
Number of attributes	1	2
Number of properties	1	0
Number of functions	1	3
LOC	15	22

6. Validity and limitations

This section is focused on an internal and external validity [28] and on the limitations of the study. In order to guarantee the accuracy of the extracted data from the raw source files, the ANTLR language tool was used. Furthermore, the C# and Java grammar, which describes a programming language, were used. The tool requires a syntactically correct source code in order to build an abstract syntax tree. However, the tree constitutes the basis for extracting the meta-data into the meta-models used by framework.

The metrics' value calculations rely on the nodes defined in the abstract syntax tree. Therefore, the metrics' equations are unified and defined only once in the higher level of abstraction. For example, if the metric for calculating cyclomatic complexity is defined upon the universal nodes in a syntax tree then their values could be easily calculated for C# and Java programming language without knowing the specifics of the programming language. Furthermore, such an approach enables the calculation of software metrics in the same way between programming languages. Thus, comparing the quality of heterogeneous systems in such an environment is more accurate.

In this research, a programming language independent framework for analyzing software evolution was successfully applied in a case study and partly in the preliminary work. In the first part, the source code of two programming languages (i.e. C# and Java) were used in order to analyze its structure. The results showed that the meta-model for describing software structure was successfully populated with the meta-data. In the second part, the SMIILE tool that evaluates software metrics was used. Several software metrics were defined using the eCST and tested with different programming languages.

The metric values were correctly calculated for all cases. This was proven by using independent and language-specific software metrics tools.

The research showed that the framework is general (i.e. programming language independent) and can be used for more programming languages.

The limitation of the framework is the extent of meta-data that are described in the meta-model for representing software structures. The study has been limited to the subset of changes defined in Fowler's book [18]. However, the book contains numerous refactorings for resolving bad smells in code and design. From this perspective, this limitation is not a real limitation. Furthermore, software metrics' support gives additional value to the framework because the quality of changes can be tracked.

Different structural source code changes cannot always be compared among programming languages. Some of them have special object-oriented constructs (e.g. properties in C#) or behaviour (e.g. some languages support multiple inheritance). Another limitation is that the LOC metric has to be compared with caution. For example, a high LOC value in a class (e.g. 10,000 lines of code) will always indicate a large class and vice versa. On the other hand, we cannot predict that the LOC values of the classes with the same behaviour, written in C# and Java, will be equal. However, even if we can not strictly compare every aspect among languages, we still provide consistent monitoring of software evolution. Furthermore, adding weights to some metric values (e.g. the LOC metric) can lead to better comparability among languages. However, such an approach can improve the framework in the future.

7. Related work

In the last decade, different approaches for evaluating software artifacts have been used. Therefore, this section will focus on existing tools and approaches for evaluating software metrics and software structure through the software development process. As the analisys of related work will show, the integrated approach to application of software metrics algorithms and analysis of software evolution that are independent on programming languages are not existant in usable form. The first part describes the approaches for evaluating software metrics, the second part describes the approaches for representing software structure and the third part is focused on programming language independence.

The majority of problems are related to programming language dependency. However, the last part of the analysis describes some more or less successful approaches in order to overcome this issue. To improve existing approaches, new framework that is based on internal representation of source code with improved characteristics was developed.

7.1. Software metric approaches

In this section, the findings of current problems in the application of software metrics in practice are described [39]. Some preliminary observations of the field show that the main problem lies in the weaknesses of available metric tools and techniques. These observations are based on numerous reports on the weaknesses of existing tools in both practice and in the academic world ([30] and [32]).

Our analysis included 20 tools, with six of them being representative examples. The tools were analyzed with respect to two groups of criteria.

The first group of criteria is related to the usage range of a tool and by the nature and structure of the software product being measured. However, the group of criteria consists of: platform independence, input language independence and a list of supported metrics. The following metrics were considered:

- the cyclomatic complexity - CC,

– the Halstead metrics - H,
– lines of code - LOC (if a tool calculates any of the LOC (SLOC, CLOC, etc.) metric, then the corresponding cell contains the '+' symbol),
– the object-oriented metrics - OO (if a tool supports any of the OO metrics, then the corresponding cell contains the '+' symbol. The mark '*' next to the symbol '+' means that a tool only partially satisfied specified criteria) and
– the others (if a metric is supported and it does not belong on the list above, then the criteria is marked with a '+').

The results for six representative tools can be seen in Table 7.

Table 7. Software metric tools and observed criteria

Tool	Producer [see ref]	Platform indep.	Language indep.	CC	H	LOC	OO	Other metrics
SLOC	D. Wheeler [47]	-	+	-	-	+	-	-
Code Counter Pro	Geronesoft [2]	-	+	-	-	+	-	-
Source Monitor	Campwood Software [8]	-	-	+	-	+	+	-
Understand	ScientificToolworks [1]	+	-	+	+	+	-	-
RSM	MSquared Technologies [7]	+	-	+	+	+	-	-
Krakatau	Power Software [5], [6]	-	+*	+	+	+	+	-

The important conclusions of this analysis are below.

– The analyzed tools could be divided into two categories.
  • The first category includes tools that only calculate simple metrics (i.e. the LOC metrics) but for a wide set of programming languages.
  • The second category of tools is characterized by a wide range of metrics but limited to a small set of programming languages. There were attempts to bridge the gap between these categories, but without success. This is a limitation because there are many legacy software systems written in ancient languages, whereas modern metric tools cannot be applied uniformly.
– Even if the tools support some object-oriented metrics, the amount of supported metrics is fairly small. This is especially true when compared to the broad application of the object-oriented approach within current software development.

However, we have demonstrated on representative languages that the SMI-ILE is language independent for currently implemented software metrics (section 3.3). The process of calculating them can be strictly connected with the language syntax (e.g. the CC metric) or it can be less sensitive to its syntax and lexical analysis because we have enough data in the universal nodes (e.g. the LOC metric with the first and the last line). The object-oriented metrics are still decently supported in tools. However, we have an internal representation of the source code and its design. This is the basis for metric calculation and our next task is to extend the set of algorithms for calculating software metrics[38].

Furthermore, the analysis considered support for processing and interpreting the calculated metric results via the given tools. The criteria were: the history of the source code, the metric results' storing facility, a graphical representation of the calculated values and an interpretation of the calculated values including suggestions for improvements based on the calculated values. The general conclusion was that many techniques and tools compute numerical results with no real interpretation of their meaning. The only interpretations of numerical results that can be found are graphical. These results possess little or no value for practitioners, who need suggestions or advice on how to improve their project based on the metrics' results. Recommendations for an improvement, or even the automation of an improvement based on the obtained metrics results, would be significantly useful for the way to the real practical usability of software metrics.

Today, complex software projects are developed in several programming languages while available software metric tools are not language independent. When taking these facts into account, we can conclude that the use of several software metric tools in one project is required. An additional problem is that different software tools often provide different values for the same metric, calculated on the same product or its component [36],[32]. One of the reasons for this is the fact that the rule for metrics calculations could be differently interpreted and implemented with different tools [43]. On the other hand, our approach uses a common internal representation of the source code and meta-model for all programming languages that represents a basis for metrics calculation. Such an approach enables the same metrics calculation algorithms across different programming languages.

7.2. Approaches for analyzing software structure

Software evolution analysis covers different aspects of software development. ¿From the granularity level, two major approaches exists. The version-centered approach considers versions to be a representation of granularity, while the history-centered approach considers history to be a representation of granularity [21].

The research conducted by Gîrba et al. [21] focused on the set of requirements that an evolution meta-model should have. Therefore, they defined a meta-model where history is modeled as an explicit entity. A time component was set as the basis for structural information, which thus provides a common

infrastructure for expressing and combining evaluation analysis and structural analysis. The authors also focused on different abstraction and detailed levels, the ability to compare and combine property evolutions and the ability of history navigation between relations. Our meta-model differs from this study in three aspects: the meta-model is programming-language independent (supports object-oriented and procedural languages), it has the ability to represent software metrics and it provides a sufficient basis for detecting structural code changes between versions.

Tichelaar et al. [45] investigated similarities between refactorings for Smalltalk and Java programming languages. They derived a language independent meta-model for object-oriented source code and showed that it is feasible to build a language independent engine for refactorings on top of this meta-model. Our study is similar in the context of an independent meta-model and differs in the ability to provide sufficient data to analyze different structural source code changes between versions over the software evolution. However, some refactorings are composed by one or several structural source code changes.

Studies conducted in [21, 45] are based on a language independent and extensible model for modeling object oriented software systems, called FAMIX [44].

7.3. Programming language independence

This section focuses on various universal software tools that strive to achieve the independence of an input programming language.

The FAMIX meta-model [44] boasts one of the most similar general goals with our project. Its strength is mainly in language independence. It supports OO design (at the interface level of abstraction) for various input programming languages and is supported by separate tools for filling in the meta-model with sources in different programming languages. Our approach is more general - it is based on (enriched) syntax trees representing all aspects of source code, instead of just the design. It is thus equally appropriate for supporting a broader set of software metrics. However, it also fully supports procedural languages, including legacy ones (e.g., COBOL).

Arbuckle[11] presented an interesting approach for the measuring evolution of a multi-language software system. He avoids difficulties related to syntax, semantics and language paradigms by looking directly at relative shared information content. His approach measures a relative number of bits of shared binary information between artifacts of consecutive releases. However, our approach uses source code changes from software repositories to analyze software evolution.

The ATHENA tool for assessing the quality of software [15] was based on the parsers that generate abstract syntax trees as a representation of a source code. The generated trees were structured in such a way that the metric algorithms were easily applied. The final goal of the tool was to generate a report that describes the quality. However, it was only executable under the UNIX operating system and its official support is not available anymore. Our approach

is also based on usage of specific parser for generating of syntax trees, but our parers are not manually implemented but generated by a parser generator. This makes the process of adding support for a new programming language easier. Furthermore, the eCST has a richer representation than AST. At the end, the SMIILE tool is implemented in Java and can therefore be used on a broader range of platforms.

The development of the CodeSquale metrics tool was based on a similar idea. The early results were published on the project website [3], [4]. The authors developed a system, based on the representation of a source code through abstract syntax trees, and implemented one object-oriented metric for the Java source code. Furthermore, an idea for the additional implementation of other metrics and opportunities for extending the tree to other programming languages was described. However, their final goal was programming language independence. Unfortunately, later results were not published. However, a weakness of the project was the use of an AST to represent source code. By using the eCST we are able to implement algorithms that are independent of programming language.

The Wide Spectrum Language (WSL) [10] is used for the intermediate representation of software programs in translating legacy to maintainable code (eg. assembly code to C/COBOL code). The main characteristics of WSL is a formal background and the application of formal transformations of code internally represented by using abstract syntax trees. Even the WSL is (by definition) independent of programming languages. Nevertheless, it still does not support object-oriented languages. In the process of program transformation, a small set of software metrics is used to measure the effects of transformations. In comparison with WSL, our approach supported a broader scope of languages and metrics (including object-oriented).

Static analysis usually includes some metrics calculation and further analysis of the obtained values. Such a study was presented in [46] where the authors used a static analysis for student programs written in Java. The study is based on an abstract syntax tree (AST) to represent the code. The XML format was used in order to represent the data.

The AST representation of the source code also led to language independence in some related areas of software engineering. The tool described in [16] uses the abstract syntax tree for the representation of source code in a duplicated code analysis. The tree has specific mechanisms designed for the easier implementation of algorithms and comparisons. A similar approach was described in [13] but a more complex algorithm for comparison was implemented.

An approach for detecting similar classes in Java source code was presented in [42]. Furthermore, ASTs were also used to monitor software changes [35]. The specified tool was implemented for the analysis of code written in the C programming language. Its significance is in its ideas for change analysis based on AST.

8. Conclusion and future work

The programming language independent framework for analyzing software structure and metrics during software development and evolution was presented in the study. The framework consists of three modules. The first one defines the meta-model for providing sufficient data, which constitutes the basis for approaches dealing with software change detection processes. The second module uses a mechanism for evaluating software metrics. Both modules are built on the eCST for the unified representation of a source code. The last module contains an approach for collecting evolutionary software artifacts that then enables further analysis.

The integration of improved characteristics of eCST into a framework for metrics calculation and the framework for software evolution extended by software metrics and the changes repository lead to the following important benefits.

- Usage of the eCST leads to language independence.
- The storing of software metrics in the software metrics repository enables a better interpretation of acquired data.
- Integration with the repository additionally gives opportunities to extend it in such a way as to store data about structural software changes.
- Enables the further analysis of stored data (e.g. custom metrics) and provides the opportunity to give recommendations to users about the improvement of a product and the development process or even the automation of some of the suggested improvements (e.g. automatic refactoring). The need for this analysis is examined in the related work. An advanced calculation on metric values and visualization are enabled by the software metrics repository and the rest of an intelligent analysis are planned for future work.

These features distinguish the framework from existing techniques and approaches and provide it with significant prospects in the field of software development and evolution. Furthermore, a special engine for detecting structural source code changes is already being implemented, but it is out of the scope of this paper.

The framework was successfully presented in the case study. In the first part, the software structure was analyzed from source code written in two different programming languages (i.e. C# and Java). The data extracted from the eCST into the meta-model fulfilled all described requirements. However, programming language independency of the eCST has been also shown on other case studies [39] and [40]. Therefore, the extraction is independent of an input language. In the second part, several software metrics were evaluated based on the same source code. The algorithms defined upon the universal nodes correctly calculated the values for *lines of code* and *cyclomatic complexity* and also for some other design metrics (i.e. *NOC, NOA, NOM*). In the last part, the results were successfully imported into the software repository for collecting and storing meta-data.

The framework and its components are still on prototype level. In future work, the framework will be tested with more programming languages and adjustments will be made if necessary. Also, support for additional algorithms for calculating software metrics will be added. Furthermore, all data will be stored in a repository with the intention of analyzing the correlation between the changes and software quality and to provide more useful information to the user or even to develop an automated quality improvement.

Acknowledgments. Work of the second and third author is partially supported by the Serbian Ministry of Science and Technological Development through project no. OI174023 "Intelligent Techniques and Their Integration into Wide-Spectrum Decision Support". Bilateral project between Slovenian Research Agency and Serbian Ministry of Science and Technological Development (Grant BI-SR/10-11-027) enabled the exchange of visits and ideas between authors of this paper and their institutions.

References

1. Understand 2.0 user guide and reference manual. online (2008), http://www.scitools.com
2. Code counter pro. online (2010), http://www.geronesoft.com/
3. Codesquale. online (2010), http://code.google.com/p/codesquale/
4. Codesquale. online (2010), http://codesquale.googlepages.com/
5. Krakatau essential pm (kepm)- user guide 1.11.0.0. online (2010), http://www.powersoftware.com/
6. Krakatau suite management overview. online (2010), http://www.powersoftware.com/
7. Rsm. online (2010), http://msquaredtechnologies.com/
8. Sourcemonitor,. online (2010), http://www.campwoodsw.com/sourcemonitor.html
9. Antlr - another tool for language recognition (2011), http://www.antlr.org
10. Wsl - wide spectrum language (2012), http://www.smltd.com/wsl.htm
11. Arbuckle, T.: Measuring multi-language software evolution: a case study pp. 91–95 (2011)
12. Basili, V.R.: Data collection, validation and analysis, p. 143160. MIT Press (1981)
13. Baxter, I.D., Yahin, A., de Moura, L.M., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: ICSM. pp. 368–377 (1998)
14. Breivold, H.P., Crnkovic, I., Larsson, M.: A systematic review of software architecture evolution research. Information and Software Technology 54(1), 16 – 40 (2012)
15. Christodoulakis, D., Tsalidis, C., van Gogh, C., Stinesen, V.: Towards an automated tool for software certification. In: Tools for Artificial Intelligence, 1989. , IEEE International Workshop on Architectures, Languages and Algorithms. pp. 670–676 (1989)
16. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: ICSM. pp. 109–118 (1999)
17. Fenton, N.E., Neil, M.: Software metrics: successes, failures, and new directions. Journal of Systems and Software (1999)
18. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1 edn. (Jul 1999)
19. Črt Gerlec, Andrej Krajnc, M.H.J.B.: Mining source code changes from software repositories. Central and Eastern European Software Engineering Conference in Russia (2011)

20. Gilb, T.: Software Metrics. Chartwell-Bratt (1976)
21. Gîrba, T., Ducasse, S.: Modeling history to analyze software evolution: Research articles. J. Softw. Maint. Evol. 18, 207–236 (May 2006)
22. Goeminne, M., Mens, T.: A comparison of identity merge algorithms for software repositories. Science of Computer Programming (2011)
23. Grune, D., Bal, H.E., Jacobs, C.J.H., Langendoen, K.: Modern Compiler Design. John Wiley (2002)
24. Harrison, W.: A flexible method for maintaining software metrics data: a universal metrics repository. Journal of Systems and Software 72(2), 225–234 (2004)
25. Heričko, M., Živkovič, A., Porkolb, Z.: A method for calculating acknowledged project effort using a quality index. Informatica 31(4), 431–436 (2007)
26. Illes-Seifert, T., Paech, B.: Exploring the relationship of a files history and its fault-proneness: An empirical method and its application to open source programs. Information and Software Technology 52(5), 539 – 558 (2010)
27. Institute, C.M.U.S.E., Martin, R., Carey, S., Coticchia, M., Fowler, P., Maher, J.: Proceedings of the Workshop on Executive Software Issues, August 2-3 and November 18, 1988. Technical report, Carnegie Mellon University, Software Engineering Institute (1989)
28. Jedlitschka, A., Ciolkowski, M., Pfahl, D.: Reporting experiments in software engineering. Guide to advanced empirical software engineering (1), 201–228 (2008)
29. Kan, S.: Metrics and Models in Software Quality Engineering Second Edition Boston. Addison-Wesley (2003)
30. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer (2006)
31. Lehman, M., Ramil, J.F., Kahen, G.: Evolution as a noun and evolution as a verb. In: Proc. Workshop on Software and Organisation Co-evolution (SOCE) (July 2000)
32. Lincke, R., Lundberg, J., Löwe, W.: Comparing software metrics tools. In: ISSTA. pp. 131–142 (2008)
33. Madhavji, N.H., Fernandez-Ramil, J., Perry, D.: Software Evolution and Feedback: Theory and Practice. John Wiley & Sons (2006)
34. N. Fenton, S.L.P.: Software Metrics: A Rigorous and Practical Approach. Thomson Computer Press (1996)
35. Neamtiu, I., Foster, J.S., Hicks, M.W.: Understanding source code evolution using abstract syntax tree matching. In: MSR (2005)
36. Novak, J., Rakić, G.: Comparison of software metrics tools for :net. In: Proc. of 13th International Multiconference Information Society - IS, Vol A. pp. 231–234 (2010)
37. Pfleeger, S.: Lessons learned in building a corporate metrics program. Software, IEEE 10(3), 67 –74 (may 1993)
38. Rakić, G., Budimac, Z.: Introducing enriched concrete syntax trees. In: Proc. of 13th International Multiconference Information Society - IS, Vol A. pp. 211–214 (2011)
39. Rakić, G., Budimac, Z.: Problems in systematic application of software metrics and possible solution. In: Proc. of The 5th International Conference on Information Technology (ICIT) (2010)
40. Rakić, G., Budimac, Z.: Smiile prototype. AIP Conference Proceedings 1389(1), 853–856 (2011)
41. Rochkind, M.J.: The source code control system. IEEE Transactions on Software Engineering 1(4), 364–370 (1975)
42. Sager, T., Bernstein, A., Pinzger, M., Kiefer, C.: Detecting similar java classes using tree algorithms. In: MSR. pp. 65–71 (2006)

43. Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: A non-invasive approach to product metrics collection. Journal of Systems Architecture 52(11), 668–675 (2006)
44. Tichelaar, S., Ducasse, S., Demeyer, S.: Famix and xmi. In: Reverse Engineering, 2000. Proceedings. Seventh Working Conference on. pp. 296 –298 (2000)
45. Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, O.: A meta-model for language-independent refactoring. In: Principles of Software Evolution, 2000. Proceedings. International Symposium on. pp. 154 –164 (2000)
46. Truong, N., Roe, P., Bancroft, P.: Static analysis of students' java programs. In: ACE. pp. 317–325 (2004)
47. Wheeler, D.A.: Sloccount user's guide, version 2.26. online (2004), http://www.dwheeler.com/sloccount/sloccount.html

Appendix

Table 8. Catalog of universal nodes.

Universal node	coresponding element of language syntax
PACKAGE_DECL	package, workspace, etc
CONCRETE_UNIT_DECL	class, implementation module, etc
ABSTRACT_UNIT_DECL	abstrat class, etc
INTERFACE_UNIT_DECL	interface, definition module, etc
EXTENDED_BASE_UNITS	extended class
IMPLEMENTED_INTERFACE_UNITS	implemented interface, corresponding definition module, etc.
INSTANTIATED_UNIT	instantionation of a new object
IMPORT_DECL	unit or function import
ATTRIBUTE_DECL	attribute, field, etc.
PROPERTY_DECL	property
FUNCTION_DECL	method, procedure, function, etc
FUNCTION_CALL	call of a function
PARAMETERS_DECL	parameters of a function
ARGUMENT_LIST	parameters passed to a function
VAR_DECL	local variable defined in functions
MAIN_BLOCK	main block of program
STATEMENT	Any statement
BRANCH_STATEMENT	Any Branch Statement (each branch will be additionally marked)
BRANCH	Branch in Branch Statement
LOOP_STATEMENT	Any Loop Statement
JUMP_STATEMENT	Any Jump Statement
CONDITION	Condition (in loop, branch,... statements)
CONDITION_BRANCH	each branch of condition separated by logical operator
LOGICAL_OPERATOR	logical operator (in condition)
OPERATOR	any operator
OPERAND	any operand
NAME	name of any element (unit, function,etc.)
TYPE	type of any element (unit, function,etc.)

Črt Gerlec is a researcher and PhD student associated with the Faculty of Electrical Engineering and Computer Science, Institute of Informatics at the University of Maribor. His research interests are mining software repositories, software evolution, software quality, software metrics, information systems and

more. He is experienced software developer on Microsoft.NET platform and expert for software architecture, design patterns and best practices.

Gordana Rakić has received her MSc degree in 2010 from Faculty of Sciences, University of Novi Sad. Currently she is the PhD student and works as assistant at Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad. Her fields of interest are Software Engineering, Software Metrics, Software Maintenance, etc.

Zoran Budimac Since 2004 holds position of full professor at Faculty of Sciences, University of Novi Sad, Serbia. Currently, he is head of Computing laboratory. His fields of research interests involve: Educational Technologies, Agents and WFMS, Case-Based Reasoning, Programming Languages. He was principal investigator of more then 20 projects. He is author of 13 textbooks and more then 220 research papers most of which are published in international journals and international conferences. He is/was a member of Program Committees of more then 60 international Conferences and is member of Editorial Board of Computer Science and Information Systems Journal.

Marjan Heričko is a Full Professor at the University of Maribor, Faculty of EE&CS, Institute of Informatics. He received his M.Sc. (1993) and Ph.D. (1998) in computer science from the University of Maribor. His research interests include all aspects of IS development with emphasis on metrics, software patterns, process models and modeling.

Received: January 04, 2012; Accepted: May 31, 2012.

DOI: 10.2298/CSIS111231027S

High-level Multicore Programming with C++11

Zalán Szűgyi, Márk Török, Norbert Pataki, and Tamás Kozsik

Department of Programming Languages and Compilers,
Eötvös Loránd University
Pázmány Péter sétány 1/C, H-1117 Budapest, Hungary
{lupin,tmark,patakino,kto}@caesar.elte.hu

Abstract. Nowadays, one of the most important challenges in programming is the efficient usage of multicore processors. All modern programming languages support multicore programming at native or library level. C++11, the next standard of the C++ programming language, also supports multithreading at a low level. In this paper we argue for some extensions of the C++ Standard Template Library based on the features of C++11. These extensions enhance the standard library to be more powerful in the multicore realm. Our approach is based on functors and lambda expressions, which are major extensions in the language. We contribute three case studies: how to efficiently compose functors in pipelines, how to evaluate boolean operators in parallel, and how to efficiently accumulate over associative functors.

Keywords: multicore programming, C++.

1. Introduction

The new standard of the C++ programming language supports parallel computation. Strictly speaking, it supports only low level constructs [20]. Although several libraries are available that provide high level parallelization tools for C++, there are numerous occasions when the usage of these libraries is not beneficial [2, 5, 17]. These libraries are complex and robust, their structure can be different from that of other ones and they have their own coding styles. Thus, if the programmer wants to use one of these libraries, first she needs to spend a lot of time to get familiar with it to be able to use it properly [16]. Our goal was to extend the standard library of C++ to support high level parallelization techniques. In our solution we made an effort to extend it only slightly, and to allow simple usage of our library for a programmer familiar with the STL.

The *C++ Standard Template Library* (STL) was developed by *generic programming* approach [3]. In this way containers are defined as class templates, and many algorithms can be implemented as function templates. Furthermore, algorithms are implemented in a container-independent way, so one can use them with different containers [14]. C++ STL is widely-used inasmuch as it is a very handy, standard C++ library that contains useful containers (like list, vector, map etc.), a large number of algorithms (like sort, find, count etc.) among other utilities.

The STL was designed to be extensible [4]. We can add new containers that can work together with the existing algorithms. On the other hand, we can extend the set of algorithms with a new one which can work together with the existing containers. Iterators bridge the gap between containers and algorithms [11]. The expression problem [21] is solved with this approach. STL also includes adaptor types which transform standard elements of the library for a different functionality [12]. Adaptors can modify the interface of a container, transform streams into iterators, modify the behavior of functors etc.

Functor objects make STL more flexible as they enable the execution of user-defined code parts inside the library [13]. Basically, functors are usually simple classes with an `operator()`. Inside the library `operator()`s are called to execute user-defined code snippets. This can call a function via pointer to functions or an actual `operator()` in a class. Functors are widely used in the STL inasmuch as they can be inlined by the compilers and they cause no runtime overhead in contrast to function pointers. Moreover, in case of functors extra parameters can be passed to the code snippets via constructor calls.

Functors can be used in various roles: they can define predicates when searching or counting elements, they can define comparison for sorting elements and property searching, they can define operations to be executed on elements of collections.

C++11, the next standard of C++, includes a new feature called *lambda functions* or *lambda expressions* [9]. Lambda expressions are able to express the functionality of a function call operator without writing explicit functor types. The call of an algorithm and the inner logic is not separated with this technique. Lambda expressions can be considered as locally defined functors. The experimental compilers generate functor types from lambda expressions. Our solution also supports lambda expressions.

It is frequently advised that one should prefer standard library to other ones. C++ programmers are familiar with the STL. Unfortunately, whereas the STL is preeminent in a sequential realm, it is not aware of multicore environment [19].

In this paper we present our results to provide high level parallelization by extending the STL. In our research we made an effort to highly reuse the existing utilities of the STL. Different functor-related techniques are implemented to make STL a more advanced, multicore supporting library. Using our library those programmers who are familiar with STL can easily adopt our extensions, without the need to spend much time to learn it. The source code of the library can be downloaded from the `http://kp.elte.hu/STLpar` URL.

The rest of this paper is organized as follows. Section 2 shows how a pipeline can be effectively implemented with functors. Section 3 describes the evaluation of composite predicates in a multithreaded way. A solution which is able to select a faster evaluation technique if we use an associative computation on huge amount of data is presented in section 4. Finally, section 5 concludes the paper.

2. Pipeline

The algorithm for_each of STL applies a functor to each element in a given range. If the functor is defined in a special way, it is able to represent the stages of a pipeline, while the algorithm for_each itself feeds it with data.

We extended the STL with a new functor adaptor to help the programmer create this kind of special functor called parallel_compose. This functor adapts two unary functors to a functor composition, and processes them in the following way. Assume that parallel_compose adapts the functors f and g, and the input parameter is x. The result of the computation is $g(f(x))$. However, after $f(x)$ is processed, the result value is passed to the functor g in a new thread. Hereby while the functor g computes its result, the functor f can start to process the next input data.

One input of parallel_compose can be an another parallel_compose thus it is possible to create arbitrary long functor composition.

This way the algorithm for_each acts as a pipeline. The simple functors in a functor composition act as a stages of the pipeline, while the algorithm for_each feeds it with data defined by the input range.

2.1. Implementation details

The core of our implementation is the parallel_compose functor adaptor. Its constructor receives two unary functors that are playing role on a functor composition. Then it wraps the second argument by a thread wrapper class (described later) and starts it to run in a new thread. However, the algorithms of STL can freely copy functors, by definition. This behavior is ineligible for us because we do not want to copy a thread. To avoid this situation we allocate the thread dynamically, and store it in a type shared_ptr, which is a smart pointer, provided by the new standard of C++ [15]. This ensures that all the temporarily copied parallel_compose functors refer to the same thread, and the memory will be deallocated automatically.

The operator() of parallel_compose invokes its first unary functor to compute the first member of the composition, and sends the result to the thread wrapper.

The member function join blocks the execution until the last element is applied on all the stages. After that it destructs the pipeline. When there is more than two stages in the pipeline, – i.e. parallel_compose functors are composed in a chain –, join must be invoked for all parallel_compose functors recursively. The naive method that invokes the join member function of the second argument does not work because the second argument of the last parallel_compose is a different unary functor, see the example in subsection 2.2, and hence it would cause a compilation error.

We applied the SFINAE (Substitution Failure Is Not An Error) technique [22] to solve this problem. We defined a new type trait to check whether the given type is a parallel_compose. Because type traits define a compile-time template-based interface to query or modify the properties of types [10], we can cus-

tomize our code for a given type without any run-time overhead. Then we created two auxiliary join functions that can be seen below:

```
template<typename T>
void join_aux(T& t,
              typename std::enable_if<
                  is_parallel_compose<T>::value,
                  int>::type n = 0)
{
  t.join();
}

template<typename T>
void join_aux(const T&)
{
}
```

The `enable_if` is a template metafunction provided by the new standard. If its first template argument is `true`, it has a public member typedef `type`, equal to its second template argument; otherwise, there is no member typedef. This metafunction is used to conditionally remove the first function `join_aux` from overload resolution based on type traits.

We invoke this `join_aux` function in the following way:

```
void join()
{
  join_aux(second_argument);
}
```

If the `second_argument` itself is a `parallel_compose` functor, then our type trait returns with true, and the metafunction `enable_if` has typedef `type`, thus both definitions of `join_aux` are valid. Since the `second_argument` is not a constant, the first `join_aux` will be selected by the compiler, thus the `join` is invoked on the next `parallel_compose` functor in a chain. However, if the `second_argument` is a different functor, then there is no typedef `type` inside of `enable_if`. This means the definition of the first `join_aux` is invalid, thus it is removed from the overload resolution. In this case only the second `join_aux` is left – the one which does not invoke any `join` –, thus that one will be selected.

The thread wrapper class wraps a unary functor, and runs it in a new thread. It has two main member functions: the `receive` and the `operator()`. The `parallel_compose` sends the data to a thread via member function `receive`, and the `operator()` performs the computation in the new thread. The key parts of these two member functions can be seen below:

```
void receive(const argument_type& a)
{
```

```
  //...
  data_lock();
  data = a;
  data_ready=true;
  //...
}

void operator()()
{
  //...
  while(run)
  {
    wait_for(data_ready);
    if(run) stored_functor(data);
    data_ready = false;
    data_unlock();
  }
  //...
}
```

The `data_lock` is used to prevent the previous stage to overwrite the data before it is computed. It will be only unlocked when the computation of current data is finished. The main loop of `operator()` runs while the logical variable `run` is true. It waits for the data, then performs the computation, and finally unlocks the semaphore to be able to receive the next one.

This class has a `kill` method, which terminates the thread. First it sets `run` to false, and `data_ready` to true. The second one is necessary because the `operator()` may be waiting for `data_ready`. But, because `run` is false, there will be no false computation.

2.2. Example

The example below illustrates the way to apply our pipeline solution in a classical image processing task [6]. The image processing contains three steps: transformation, rasterization and pixel processing. These steps will be the stages of the pipeline, and the input data is a range of triangles. The hereinafter example demonstrates our approach, but it highly simplifies the problem. In real life image processing is a more complex process, and the stages can be split into more substages to improve performance [23].

```
struct transformation
{
  triangle operator()(const triangle& value)
  {
    // ...
  }
```

```
};

struct rasterization
{
  triangle operator()(const triangle& value)
  {
    // ...
  }
};

struct pixel_processing
{
  triangle operator()(const triangle& value)
  {
    // ...
  }
};

for_each(input_iterator_begin,
         input_iterator_end,
         pcompose(transformation(),
                  pcompose(rasterization(),
                           pixel_processing()))
        ).join();
```

The `input_iterator_begin` and `input_iterator_end` are two iterators defining the input range of triangles. The stages are functors, thus they have to overload the `operator()`, which performs the computation. The `pcompose` is a helper function that creates a `parallel_compose` functor object by its arguments and returns it. Helper functions simplify the creation of functors, thus they are very common in the STL, because the C++ compiler can deduce the template arguments by the type of actual parameters (e.g. `std::make_pair`). The algorithm `for_each` returns with the functor. Thus the last function invocation calls the `join` method of the functor. This synchronization step waits for the pipeline to finish the computation.

3. Speculative Functors

There are several algorithms in the STL that take a predicate functor as argument to decide whether an element must be processed. The predicate is a unary functor (its `operator()` has exactly one argument) that returns a boolean value. If the predicate returns true for a given element, the algorithm will deal with that element. The names of these algorithms have an `_if` postfix, such as: `find_if`, `count_if`, `remove_if`, `replace_if` etc.

In many cases the predicates are very complex. As the predicate is a logical condition, it is often constructed from functors composed by `logical_and` or `logical_or`.

If the subexpressions composed by `logical_and` are complex, it might worth to evaluate them in parallel. The more complex the subexpression is, the more speed-up we can achieve.

We introduce a new functor adaptor called `speculative_logical_and`, which can evaluate the subexpressions in separate threads. If one thread computes the result of its subexpression, we check whether it is `false`. If so, we got the result – thus, we kill the other thread, or drop its result if it is terminated already. Otherwise we wait for the result of the other thread and use both results.

3.1. Implementation details

Technically the `speculative_logical_and` is a unary functor that composes the predicates `f` and `g` in the following way: $f(x) \&\& g(x)$, where `x` is the input parameter.

The `speculative_logical_and` receives the predicates in its constructor, which wraps them with a thread wrapper class to ensure they run in separate threads.

The work is done by the `operator()`. The core of its implementation can be seen below:

```
return_type operator()(const argument_type& a)
{
  compute_in_new_thread(f,a);
  compute_in_new_thread(g,a);

  wait_for(impl->has_result_f || impl->has_result_g);

  if(impl->has_result_f)
  {
    if(impl->result_f == false)
    {
      kill(g);
      return false;
    }
    else
    {
      wait_for(impl->has_result_g);
      return impl->result_g;
    }
  }
  else
  {
```

```
    if(impl->result_g == false)
    {
      kill(f);
      return false;
    }
    else
    {
      wait_for(impl->has_result_f);
      return impl->result_f;
    }
  }
}
```

The members of `speculative_logical_and` functor are put into an implementation class and the variable `impl` – a shared pointer pointing to it. This solution ensures that the functor can be copied by the STL, and that all the copies refer to the same implementation. `impl` is added to the thread wrappers, thus the threads can store their results into that. After one thread computes the result it sets its `has_result` variable to true, indicating to the main thread that the data is ready.

The `speculative_logical_and` waits until one thread is ready and checks the result. If it is `false`, the whole result is `false`, thus the other thread can be killed, and `false` will be returned. Otherwise `speculative_logical_and` waits for the result of the other thread, that value will be returned. (That way the first argument of *logical and* is true, thus the result depends on the second argument.)

In practice, `speculative_logical_or` behaves similarly. The only difference is that it kills the slower thread if the result of the faster one is `true`.

3.2. Example

The example above shows the usage of our solution. There is a range of log entries which contains several fields, such as: timestamp, priority, user name, log message. We would like to find those entries which were created on 20.03.2011 and the message fits a given regular expression.

```
struct log_entry
{
  std::string username;
  std::string message;
  time_t      timestamp;
  int         priority;
  // ...
};

struct is_proper_date
```

```
{
  bool operator()(const log_entry& le)
  {
    /*compute if le is created on 20.03.2011*/
  }
};

struct has_proper_message
{
  bool operator()(const log_entry& le)
  {
    /*compute if message fits to a regex*/
  }
};

std::find_if(input_iterator_begin, input_iterator_end,
             speculative_and(
                 is_proper_date(), has_proper_message()));
```

The `input_iterator_begin` and `input_iterator_end` are two iterators defining the input range of log entries. The `speculative_and` is a helper function to create `speculative_logical_and` functor. This helper function behaves similarly to the helper function `pcompose` described in the previous section.

This solution is efficient to use when the subexpressions are complex.

4. Associative Functors

In this section we present an approach to compute an associative operation on a huge amount of data effectively. We improve the `accumulate` algorithm of STL to be as effective as possible [8].

By default the algorithm `accumulate` computes the sum of the elements of a given range. However, we can customize the algorithm defining an own operation instead of addition. The operation is defined by a binary functor (it has two arguments) and it is an argument of `accumulate` [7]. If the operation is associative, we can apply the optimized version of the accumulate algorithm.

A technique is presented to overload algorithms on the associativity of their functor in [19]. This technique includes a trait type called *functor traits*. This type is similar to the iterator traits of STL; functor traits consist of some typedefs. It is possible to overload algorithms on the associativity of the functor based on these typedefs [18].

Our main goal was to support the new standard proposal, where lambda expressions can replace functors. However, it is not possible to define functor traits in lambda expressions. We need to denote that an operation is associative in a different way.

4.1. Implementation details

In our solution an extra argument of a lambda expression, which has a special type, called `associative`, denotes that the operation is associative. Our implementation of algorithm `accumulate` is able to detect whether a given lambda expression has that extra argument or not. If the lambda expression is associative, the optimized algorithm [19] is chosen – otherwise we take the original one.

The example below shows the way we determine if the lambda expression has that extra argument. In this section we suppose that the range is defined by a pair of *random access iterators*. STL algorithms can be overloaded on iterator category easily [18].

```
template< typename RandomAccesIterator,
          typename T,
          typename BinFunctor>
T accumulate( RandomAccesIterator first,
              RandomAccesIterator last,
              T init,
              BinFunctor bf )
{
  typedef
    T (BinFunctor::*funtype) (T, T, associative) const;

  if( std::is_same< decltype(&BinFunctor::operator()),
                    funtype>::value )
  {
    return associative_accumulate(first, last, init,
      std::bind(bf, std::_1, std::_2, associative());
  }
  else
  {
    return std::accumulate(first, last, init, bf);
  }
}
```

The `BinFunctor` template type refers to the lambda expression, while `T` refers to the elements of the input range. The static field `value` of template type `is_same` is true if the its two template arguments are same. We instantiate it with a type of the member function pointer of the `operator()` which has that extra argument and the type of the member function pointer of `operator()` of the current functor. If the lambda expression has the extra argument, the two types are the same. The `value` is computed at compile time. As C++ template metaprograms run during compilation [1], the if statement in the example can be replaced by a template metaprogram to make our solution more efficient. That way the selection of the proper algorithm is done at compile-time. When

the `accumulate` is instantiated by an associative functor, that functor technically is a ternary functor, – its third argument refers to the associativity. That case we need to transform it into a binary functor. The `std::bind` does this work, binding a dummy value to the third arguments. This solution is backward compatible to the original functor usage.

The more efficient version of the algorithm uses the following functor for the computation in a distributed way:

```
template <typename Iterator, typename BinFunctor>
struct Accumulate
{
  void operator()(
    Iterator first,
    Iterator last,
    typename
      std::iterator_traits<Iterator>::pointer p )
  {
    typename std::iterator_traits<Iterator>
      ::difference_type diff =
         last - first;

    if ( 2 == diff )
    {
      *p = BinFunctor()( *p, *first );
      *p = BinFunctor()( *p, *(first + 1 ) );
    }
    else if ( 1 == diff )
    {
      *p = BinFunctor()( *p, *first );
    }
    else
    {
      typename
        std::iterator_traits<Iterator>::pointer p1 =
        new std::iterator_traits<Iterator>::value_type;

      typename
        std::iterator_traits<Iterator>::pointer p2 =
        new std::iterator_traits<Iterator>::value_type;

      std::thread t1(Accumulate(),first,first+diff/2,p1);
      std::thread t2(Accumulate(),first+diff/2,last,p2);
      t1.join();
      t2.join();
      *p = BinFunctor()( *p, *p1 );
      *p = BinFunctor()( *p, *p2 );
```

```
      delete p1;
      delete p2;
    }
  }
};
```

If the range has only 1 or 2 elements, the functor calculates the associative operation, otherwise it divides the input range into two smaller ranges and starts the calculation of smaller ranges in separate threads.

The `associative_accumulate` initializes the shared data and starts the parallel computation:

```
template< typename RandomAccessIterator,
          typename T,
          typename BinFunctor>
T associative_accumulate( RandomAccessIterator first,
                          RandomAccessIterator last,
                          T init,
                          BinFunctor bf )
{
  typename
    std::iterator_traits<RandomAccessIterator>::
      pointer p = new T( init );

  std::thread s( Accumulate<RandomAccessIterator,
                            BinFunctor>(),
                 first,
                 last,
                 p );
  s.join();
  T result = *p;
  delete p;
  return result;
}
```

4.2. Example

The example below shows the usage of our solution to calculate the product of the input range of integers.

```
accumulate(input_iterator_begin, input_iterator_end, 1,
           [](int a, int b){return a * b;});
accumulate(input_iterator_begin, input_iterator_end, 1,
           [](int a, int b, associative){return a * b;});
```

The first function call summarizes the input range in conventional way, while the second one applies the more effective algorithm which exploits the associativity.

4.3. Threshold

A more sophisticated and more effective implementation of the evaluation of associative operation uses a threshold parameter. This parameter defines how many elements in the range require the evaluation in a new separate thread. This parameter highly depends on the characteristic of the problem.

As the previous subsection presents, the user may not know that different implementation strategies are available according to the defined operation. However, the user just states that his own operation is associative. How the threshold parameter can be defined by user?

Two different approaches are discussed: if the operation is defined by a functor type or a lambda function.

A functor type can contain special member variables and member function that can be used by the `accumulate`. For convenience, an associative operation base type can be defined. This base type contains the `associative` typedef and the default threshold value. The user has to create a subtype, and he is able to override the threshold value. If the functor type is not subtype of the associative operation base type, but is an associative operation, the previous implementation works.

Lambda expressions cannot contain member variables and member functions. The compiler generates a simple functor class from the definition of lambda, but the generated functor is unavailable for extension. However, the possibility of lambda-defined threshold needs extensive inspection.

However, the threshold value does not belong to the definition of the associative operation from the view of modularity. It belongs to the `accumulate`. This means that `accumulate` has an extra parameter which defines the threshold value. This argument may be defined if the operation is associative. Since associativity is a compile-time information, compilation diagnostics can be emitted if the operation is not associative and threshold parameter is given by the user. If the operation is associative but no threshold is defined, the previous code does work. This scenario does not depend on if the operation is defined by functor or lambda function. Our future work includes the detailed implementation of this approach.

There are other approaches to solve this problem, but we reject them. One of them is that the `associative` type contains the threshold value. This can be a static value which is problematic from the view of parallelism. In the other one, the threshold value is a template argument or a member set by its constructor. Unfortunately, this makes the invocation of algorithm hard to maintain and the algorithm cannot obtain this value effectively.

5. Conclusion

Multicore programming is an interesting new way of programming. Although the current C++ programming language contains no constructs to write multi-threaded programs, extensions and libraries can still be used. The next stan-

dard of C++ includes constructs for parallel program execution. Unfortunately, these constructs are at a low level.

In this paper we argue for higher level constructs – ones at the level of the widely used C++ Standard Template Library. We implemented special functors and adaptors which support different kinds of evaluation in a multithreaded way.

- We can build up a pipeline of computations using a functor combinator.
- We can make use of speculative parallelism in the case of complex predicates.
- We can take advantage of associative operations; STL algorithms can be overloaded on their operation's associativity, even if the operation is defined as a lambda expression.

A programmer familiar with the STL can easily adopt our library.

Acknowledgments. The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TÁMOP 4.2.1./B-09/1/KMR-2010-0003).

References

1. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Reading, MA., USA (2004)
2. Aldinucci, M., Ruggieri, S., Torquati, M.: Porting decision tree algorithms to multicore using FastFlow. In: Proceedings of European Conference in Machine Learning and Knowledge Discovery in Databases (ECML PKDD), Lecture Notes in Computer Science, vol. 6321, pp. 7–23. Springer-Verlag, Berlin Heidelberg New York (2010)
3. Alexandrescu, A.: Modern C++ Design. Addison-Wesley, Reading, MA., USA (2001)
4. Austern, M.H.: Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison–Wesley, Reading, MA., USA (1998)
5. Dagum, L., Menon, R.: Openmp: An industry-standard API for shared-memory programming. IEEE Computational Science and Engineering 5, 46–55 (1998)
6. Foley, J.D., van Dam, A., Feiner, S.K., Hughes, J.F.: Computer Graphics, Principles and Practice. Addison-Wesley, Reading, MA., USA (1990)
7. Frigo, M., Halpern, P., Leiserson, C.E., Lewin-Berlin, S.: Reducers and other cilk++ hyperobjects. In: Proceedings of Symposium on Parallel Algorithms and Architectures (SPAA). pp. 79–90 (2009)
8. Gottschling, P., Lumsdaine, A.: Integrating semantics and compilation: using C++ concepts to develop robust and efficient reusable libraries. In: Proceedings of the 7th international conference on Generative programming and component engineering, GPCE 2008. pp. 67–76 (2008)
9. Järvi, J., Freeman, J.: C++ lambda expressions and closures. Science of Computer Programming 75(9), 762–772 (2010)
10. Kalev, D.: The type traits library, [Online]. Available: http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=276
11. Kozsik, T., Pataki, N., Szűgyi, Z.: C++ Standard Template Library by infinite iterators. Annales Mathematicae et Informaticae 38, 75–86 (2011)

12. Matsuda, M., Sato, M., Ishikawa, Y.: Parallel array class implementation using C++ STL adaptors. In: Proceedings of the Scientific Computing in Object-Oriented Parallel Environments, Lecture Notes in Computer Science, vol. 1343, pp. 113–120. Springer-Verlag, Berlin Heidelberg New York (1997)
13. Meyers, S.: Effective STL - 50 Specific Specific Ways to Improve Your Use of the Standard Template Library. Addison-Wesley, Reading, MA., USA (2001)
14. Stroustrup, B.: The C++ Programming Language (Special Edition). Addison-Wesley, Reading, MA., USA (2000)
15. Stroustrup, B.: The design of C++0x – Reinforcing C++'s proven strengths, while moving into the future. C/C++ Users Journal 23(5) (May 2005)
16. Szűgyi, Z., Pataki, N.: Generative version of the FastFlow multicore library. Electronic Notes in Theoretical Computer Science 279(3), 73–84 (2011)
17. Szűgyi, Z., Pataki, N.: A more efficient and type-safe version of FastFlow. In: Proceedings of Workshop on Generative Programming 2011. pp. 24–37 (2011)
18. Szűgyi, Z., Török, M., Pataki, N.: Multicore C++ Standard Template Library in a generative way. Electronic Notes in Theoretical Computer Science 279(3), 63–72 (2011)
19. Szűgyi, Z., Török, M., Pataki, N.: Towards a multicore C++ Standard Template Library. In: Proceedings of Workshop on Generative Programming 2011. pp. 38–48 (2011)
20. Szűgyi, Z., Török, M., Pataki, N., Kozsik, T.: Multicore C++ Standard Template Library with C++0x. In: NUMERICAL ANALYSIS AND APPLIED MATHEMATICS ICNAAM 2011: International Conference on Numerical Analysis and Applied Mathematics, AIP Conference Proceedings, vol. 1389, pp. 857–860. American Institute of Physics (2011)
21. Torgersen, M.: The expression problem revisited – four new solutions using generics. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP) 2004, Lecture Notes in Computer Science, vol. 3086, pp. 123–143. Springer-Verlag, Berlin Heidelberg New York (2004)
22. Vandevoorde, D., Josuttis, N.M.: C++ Templates – The Complete Guide. Addison–Wesley, Reading, MA., USA (2002)
23. Wei, H., Yu, J., Li, J.: The design and evaluation of hierarchical multi-level parallelisms for h.264 encoder on multi-core architecture. Computer Science and Information Systems 7(1), 189–200 (2010)

Zalán Szűgyi is assistant at Faculty of Informatics, Eötvös Loránd University (Budapest, Hungary) since 2010. He is teaching C++ programming language. His research area includes static analysis of programming languages, multicore programming, and generative programming.

Márk Török is a PhD student at Faculty of Informatics, Eötvös Loránd University (Budapest, Hungary) where he is assistant since 2010. Programming languages and multicore programming belong to the fields of his interest.

Norbert Pataki is assistant at Faculty of Informatics, Eötvös Loránd University (Budapest, Hungary) since 2009. His research area includes programming languages (especially the C++ programming language), multicore programming, software metrics, and generative programming.

Tamás Kozsik received his PhD (summa cum laude) in computer science in 2006 at Eötvös Loránd University (Budapest, Hungary), where he works as associate professor and vice-dean for scientific affairs and international relations of Faculty of Informatics. Since 1992 he has been teaching programming languages, as well as distributed and concurrent programming. His research fields are program analysis and verification, refactoring, type systems and distributed systems. His PhD thesis investigated the integration of logic-based and type system based verification of functional programs.

Received: December 31, 2011; Accepted: May 21, 2012.

DOI: 10.2298/CSIS120102025M

Supporting heterogeneous agent mobility with ALAS

Dejan Mitrović[1], Mirjana Ivanović[1], Zoran Budimac[1], and Milan Vidaković[2]

[1] Faculty of Sciences, Department of Mathematics and Informatics
Trg Dositeja Obradovica 4, 21000 Novi Sad, Serbia
{dejan, mira, zjb}@dmi.uns.ac.rs
[2] Faculty of Technical Sciences
Trg Dositeja Obradovica 6, 21000 Novi Sad, Serbia
minja@uns.ac.rs

Abstract. Networks of multi-agent systems are considered to be heterogeneous if they include systems with different sets of *API*s, running on different virtual machines. Developing an agent that can operate in this kind of a setting is a difficult task, because the process requires regeneration of the agent's executable code, as well as modifications in the way it communicates with the environment. With the main goal of providing an effective solution to the heterogeneous agent mobility problem, a novel agent-oriented programming language, named *ALAS*, is proposed. The new language also provides a set of programming constructs that effectively hide the complexity of the overall agent development process. The design of the *ALAS* platform and an experiment presented in this paper will show that an agent written in *ALAS* is able to work in truly heterogeneous networks of multi-agent systems.

Keywords: agent-oriented programming languages, mobile agents, heterogeneous agent mobility, multi-agent systems

1. Introduction

According to the *weak notion of agency* [39], *software agents* can be defined as executable software entities characterized by autonomous behavior, social interaction with other agents, reactivity to environmental changes, and the ability to take the initiative and express goal-directed behavior. The *strong notion of agency* [40] extends this definition by including human-like behavior and mental categories, such as beliefs, desires, and intentions (the so-called *BDI* agents).

Agents usually don't exist on their own, but are rather situated inside an environment. This runtime agent environment is often referred to as a *multi-agent system* (*MAS*). Main tasks of a *MAS* are to control the agent life-cycle, provide the messaging infrastructure, and offer a service subsystem that effectively supports agents, giving them the possibility of accessing resources, executing complex algorithms, etc.

An agent, however, does not have to be confined to a single *MAS* instance. A *mobile* agent is able to physically leave its current *MAS* and continue pursuing its goals in another machine in a network.

EXtebsible Java EE-based Agent Framework (*XJAF*) [34, 35] is a multi-agent system developed by the authors of this paper. The system is designed as a modular architecture, comprised of a set of *managers*. Each manager is a relatively independent module in charge of handling a distinct part of the overall agent-management process. There are several benefits of the modular design. For example, the system's functionality can easily be extended by the addition of new managers. In addition, each manager is accessible only though its interface, which means that even the behavior of standard managers can be easily changed.

Over the years, *XJAF* has been successfully used in several software systems, such as the virtual central catalogue and a metadata harvesting system for library records [34]. Recently [16], it has also been proposed as an underlying platform for agent-based harvesting of learning resources needed by e-learning and tutoring systems. But, despite its successful practical usage, *XJAF* had a disadvantage of being "locked" into a particular development platform. Because it was implemented in Java, only Java-based external clients were able to use the system and interact with its agents. In order to increase the interoperability of the system and enable its wide-spread use, *XJAF* has been redesigned as a service-oriented architecture (*SOA*). The new system, named *SOA-based MAS* (*SOM*) [22], retains the manager-based design, but with managers re-defined in terms of web services. In this way, even regular web browsers can be used as clients of *SOM*, since the interaction relies on the standardized communication protocol (i.e. *SOAP* [36]).

The *SOA*-based design of *SOM*, however, poses another problem. The system is (only) an abstract specification of web services, their functionalities and interactions, and any modern implementation platform can be used. But, developing an agent that can run on any of these implementations becomes almost an impossible task. In the literature (e.g. [24]), this issue has been recognized as an *agent-regeneration* problem: if a mobile agent migrates across a network consisting of *MAS*s that offer the same *API*, but are based on different virtual machines, its executable code needs to be regenerated for each *MAS* it visits.

Unfortunately, the lack of *MAS* interoperability is not specific to different implementations of *SOM*. Currently, there exists a large number of *MAS*s offered by different vendors. And although significant efforts have been put into the standardization of the *MAS* development process (e.g. the *Foundation for Intelligent Physical Agents*, *FIPA* [11]), agents are often incapable of operating in these truly heterogeneous environments. This problem arises as a consequence of standards incompliance, usage of different implementation technologies, different sets of *API*s offered to agents, etc. The lack of interoperability is a severely limiting factor in the agent development, and in the wide-spread use of the agent technology.

In order to solve the *MAS* interoperability problem, a new agent-oriented programming language named *Agent LAnguage for SOM* (*ALAS*) is proposed. Besides providing developers with programming constructs that hide the overall complexity of the agent-development process, one of the main goals of *ALAS* is

to serve as a tool for writing agents that can execute their tasks regardless of the underlying *MAS*. Originally, in [22], *ALAS* has targeted the agent-regeneration problem. The idea has since been broadened to support the execution of agents in heterogeneous environments. The focus of the research presented in this paper will thus be to demonstrate how *ALAS* can be used to develop mobile agents that can migrate across a network consisting of Java EE-based *SOM*, Python-based *SOM*, and *JADE* [2, 17] instances.

As noted, *ALAS* belongs to the category of agent-oriented programming languages (*AOPL*s) which represent crucial tools of the *agent-oriented programming* (*AOP*). *AOP* is a software development paradigm aimed at efficient development of software agents and multi-agent systems. Its main goals are to identify, analyze, and offer solutions for the most important theoretical and practical issues associated with the design and construction of software agents.

The rest of the paper is organized as follows. Section 2 provides an overview of existing research efforts related to the work presented in this paper. Section 3 describes the architectures of *XJAF* and *SOM*, multi-agent systems that form the basis for this research. Main features of *ALAS*, its syntax and programming constructs are given in Section 4. A practical example of an *ALAS*-based mobile agent operating in a heterogeneous environment is given in Section 5. Finally, the overall conclusion and future research directions are outlined in Section 6.

2. Related work

Related research efforts presented in this section are divided into three parts. The first part includes a general overview of existing multi-agent systems. The second part outlines the state-of-the-art of *AOPL*s. The final part deals with the work dedicated to interoperability multi-agent systems.

2.1. Multi-agent systems

Java Agent DEvelopment Framework (*JADE*) [2, 17] is a Java-based, *FIPA*-compliant *MAS*. At runtime, the framework consists of one of more *agent containers*, runtime environments with full support for agent execution. Individual containers can be distributed across a network, in which case they are linked to a designated *main* container. Each *JADE* agent has it own thread of control and exposes its functionalities in terms of *behaviors*. That is, for each functionality offered by an agent, developers need to define a separate class which extends the *Behaviour* class, or one of its more specialized subclasses. A background *scheduler* is then used to schedule execution of each behavior.

The main advantage of *XJAF* and *SOM* over *JADE* is in the use of Java EE, the *de facto* standard development platform for building large-scale, scalable, secure, and reliable software. Java EE includes a large set of standardized libraries and technical solutions which simplify the process of *MAS* development. More importantly, the use of modern enterprise application servers incorporates effective runtime agent load-balancing techniques into *XJAF* and *SOM*. Finally,

the *SOA*-based design of *SOM* results in the system with greater interoperability.

The *SOA* design philosophy has been employed in the development of *FUSION@* [31, 32], a modular, *FIPA*-compliant *MAS*. Functionalities of the system are exposed as *services* that can be accessed locally, or remotely through web interfaces. The set of services is not fixed, which means that the system can easily be extended with new functionalities. Any programming language can be used for implementing new services, as long as it supports *SOAP*. *FUSION@* also employs several types of specialized, system-level *BDI* agents. Their main task is to maintain high quality of service (*QoS*), by performing runtime load distribution, monitoring all incoming and outgoing messages, etc.

Many properties of *FUSION@*, such as the extensible modular architecture, and the use of low-level services, have been used in *XJAF*, although several years earlier. Additionally, *XJAF* delegates some functionalities of *FUSION@*'s system-level agents, such as runtime load-balancing, to an enterprise application server, which simplifies the overall development process. Finally, it is not clear how and if agent mobility is supported in *FUSION@*, or whether there is a mechanism for organizing distributed instances of the environment. These techniques have been built into *XJAF* and *SOM* from the start.

NOMADS [4, 30] is one of the few *MAS*s that support strong agent mobility (e.g. all other systems mentioned in this paper support weak mobility only). In order to achieve this feature, *NOMADS* runs on top of a customized, Java-compatible virtual machine named *Aroma*. *Aroma* can transparently capture the execution state of a single or all running threads, at the fine granularity level, and in a cross-platform manner. Additionally, it can limit the agent's access to resources and enforce similar security-related restrictions.

The use of a custom virtual machine in *NOMADS*, however, has several major disadvantages. These include interoperability issues, as well as the large amount of work that needs to be conducted in order to maintain and update *Aroma* for different operating systems and in accordance to new Java virtual machine specifications. Nonetheless, the system does offer an interesting technical insight into requirements of strong and safe agent mobility.

Java is by far the most widely used platform for *MAS* development. However, there exist notable examples of systems implemented using different technologies. One such example is *Smart Python multi-Agent Development Environment* (*SPADE*) [1, 29] implemented in Python. *SPADE* is characterized by the usage of *XMPP/Jabber* [41] instant messaging protocol for agent communication. Benefits of *XMPP/Jabber* include using [1] "an existing communication channel, the concepts of users (agents) and servers (platforms) and an extensible communication protocol based on *XML*". The *presence* feature included in the protocol also enables the real-time detection of the agent's state.

SPADE platform is accompanied by a Python-based *agent library* of classes, functions, and data structures that simplify the agent development process. The *SPADE* agent development process is heavily inspired by that used in *JADE*: functionalities of an agent are expressed in terms of behaviors, there is a sup-

port for automatic, pattern-based matching of incoming messages, and so on. The successful usage of Python as the implementation platform for *SPADE* has inspired the development of Python-based *SOM* (discussed in more details in Section 5).

2.2. Agent-oriented programming languages

AGENT0 [28] was the first agent-oriented programming language and a direct implementation of the *AOP* paradigm. In *AGENT0*, agents are defined in terms of capabilities, beliefs, and commitment rules, which consist of message and mental preconditions and resulting actions. Agents communicate by exchanging *request*, *unrequest*, and *inform* messages. New features were added to the language over time, with the two most notable direct extensions being *PLACA* [33], which introduced support for agent planning, and *Agent-K* [7], which replaced the custom communication messages with the standardized *KQML*, improving the overall interoperability. These three languages, however, mostly served as prototypes and were not designed for practical use. Their importance lays in the influence they had on the development of many later *AOPL*s.

A large family of *AOPL*s includes languages that use first-order logical formulae for describing agent's mental state and behavior. Thus, they are specifically suited for *BDI* agent architectures. Influential representatives of this family are *3APL* [6], and *AgentSpeak(L)* [26]. *3APL* supports descriptions of goals and basic and composite plans, as well as "embedding" actions inside pre-condition and post-condition rules. These rules, respectively, describe agent's belief before and after the action is executed. Another important concept of *3APL* are *goal*, *interaction*, and *plan rules*, which are used generate new and update or drop existing goals and plans. *3APL* has inspired the development of many other programming languages, most notably *2APL* [5] and *GOAL* [15]. *2APL* increases the expressiveness of *3APL* and aims at developing a more practical language. It makes a clear distinction between declarative concepts for describing agent's beliefs and goals, and imperative concepts for developing plans (unlike *3APL*, which mixes both declarative and imperative concepts in defining goals). *GOAL*, among other things, introduces the *blind commitment strategy*, a built-in goal update mechanism that automatically drops goals that have been fully achieved, and *perception rules* which enable agents to respond to external environmental changes.

AgentSpeak(L) is gaining more and more popularity due to the development of *Jason* [3, 18], an interpreter for an extended version of the language. Along with programming constructs for describing "common" features of *BDI* agents, such as beliefs, goals, rules, and plans, the extended version of *AgentSpeak(L)* supports *belief annotations*. Annotations are programming constructs used to attach additional details to agent's beliefs. They do not increase the expressiveness of the language, but improve its readability and allow for (semi-)automatic management of the agent's belief base.

Although powerful and expressive, these languages suffer from several major drawbacks. First of all, their descriptive nature and logical foundation might

make them computationally expensive and complex. Secondly, the languages were specifically designed for developing *BDI*-style agents and so are inadequate for implementing other types of architectures (e.g. purely reactive agents). But, most importantly, the logical foundation and highly-abstracted programming constructs used by this family of languages might prove to be their greatest weakness. To avoid the faith of logic and functional programming languages that were never widely adopted by the software development industry, *AOPL*s should first and foremost be as simple to use as possible, i.e. without too many high-level abstractions and without requiring a degree in computational logic to understand their concepts. *ALAS* was designed to appear as a member of the OO family of languages – the most widely used programming paradigm of today – but with clear distinctions between objects and agents. This simplified approach might turn out to be its main advantage, allowing for a broader acceptance of the agent technology.

JACK Intelligent Agents [38] is a robust, light-weight framework for rapid development of multi-agent systems. It extends the Java programming language by introducing new keywords and language constructs. The accompanying compiler produces pure Java code, which allows for each *JACK* agent to be used simply as another Java object. Although powerful, this system suffers from the same drawbacks as the original *XJAF*: it is locked into a particular development platform (i.e. Java). The *ALAS* platform is designed to allow transformation of the original agent source code into a pure source code written in an arbitrary language (such as Java and Python). With the *SOA*-based infrastructure supporting their execution, these agents can work in truly heterogenous environments and cooperate with any external *SOA*-enabled entity. In addition, *JACK* is a commercial product, while *SOM* can be freely downloaded and used.

2.3. *MAS* interoperability

Two *MAS*s are said to be *interoperable* "if a mobile agent of one system can migrate to the second system, the agent can interact and communicate with other agents (local or even remote agents), the agent can leave this system, and it can resume its execution on the next interoperable system" [25]. Therefore, interoperability of *MAS*s can seriously affect the agent's performance, by limiting its ability to move across the network or to interact with other agents.

Based on the actual types of *MAS*s that appear in a network, several types of agent mobility can be distinguished [24]:

- *Homogeneous*: all *MAS*s in the network offer the same *API* and are based on the same virtual machine (*VM*). This is the easiest type of mobility to implement, since no modifications of the agent's code are needed.
- *Cross-platform*: *MAS*s in the network offer different sets of *API*s, but are based on the same *VM*. In this scenario, the agent's executable code remains the same, but the *API* calls it makes need to be adapted.
- *Agent-regeneration*: the agent moves across *MAS* instances that offer the same *API*, but are based on different *VM*s. Therefore, the agent's executable code needs to be regenerated for each instance it visits, although

its *API* calls remain the same. This is the problem that affects different *SOM* implementations.

– *Heterogeneous*: *MAS*s in the network offer different sets of *API*s and are based on different *VM*s. This is the most difficult type of mobility to achieve, as it includes both regeneration of the executable code, and the modifications in *API* calls.

Cross-platform agent mobility is usually achieved via one or more software *layers*, where each layer is responsible for transforming *API* calls from one form to another. For example, *Grid Mobile-Agent System* (*GMAS*) [13] includes a *Foreign2GMAS* layer, which transforms agent's native *API* calls into an intermediary *GMAS API*, and a *GMAS2Native*, which transforms *API* calls made to *GMAS API* into calls to the native platform. The first layer is required for a *MAS* that needs to be able to send its agents to other architectures. Similarly, the second layer is required for a *MAS* that needs to be able to accept agents from other architectures.

Java-based Interoperable Mobile Agent Framework (*JIMAF*) [12] operates on the principle of splitting the agent implementation into a platform-independent (called the *head*) and a platform-dependent part (called the *body*). The body is executed within *Platform-dependent Mobile Agent Layer* which is implemented for each supported platform. When compared to *GMAS*, *JIMAF* is reported [12] to introduce significantly less overhead to the agent migration process.

In *ALAS*, the task of adapting *API* calls is handled transparently by the compiler's *MAS selector* component (see Section 4 for more details). The component transforms, on-the-fly, the calls made to the *ALAS* standard library of functions into native *API* calls. The main advantage of this approach is that, once the agent's executable code is regenerated, all native *API* calls are made directly. That is, there is no additional overhead introduced by layering *API* calls.

Much more work, however, is needed for heterogeneous agent mobility, since both the executable code and *API* calls need to be regenerated. *Generative migration* [24] is one proposed solution for this problem. Rather than on software layering, it relies on a pool of agent *building blocks*, platform-independent descriptions of reusable software components. Each building block is characterized solely by a description of its interface, without any details regarding the implementation. An agent is defined (or, rather, designed) by assembling and interconnecting these building blocks into an *agent blueprint*. During the migration process, the agent's blueprint is transferred, along with its runtime state. Using the blueprint, an *agent factory* tool can rebuild the agent's executable code for a specific *MAS*.

ALAS solves the same problem as generative migration does. However, *ALAS* is a programming language, syntactically (and, to a certain degree, conceptually) similar to many popular OO programming languages. Generative migration, on the other hand, might be formalized as a *Model-Driven Architecture* [27]. For a common software developer, this means that *ALAS* has a flatter learning curve than generative migration. In addition, *ALAS* has a wider goal of

simplifying the whole agent-development process, and it's not focused just on enabling heterogeneous agent migration.

3. *XJAF* and *SOM*: design and functionalities

XJAF was originally designed as a modular architecture. Each module, called a *manager*, is responsible for handling a distinct part of the overall agent-management process. The architecture defines a set of standard managers, described in the following paragraphs. This set is not fixed, and new managers (that is, new functionalities) can be added as needed. Additionally, managers are defined and used solely by their interfaces, so even the standard behavior can be changed.

AgentManager maintains the directory of agents and controls the agent life-cycle. Its functionality matches the one defined for the *FIPA*'s *Agent Directory Service* [9]. The directory of agents consists of two lists: *local* and *remote*. The first list keeps a record of agents located in the manager's host *XJAF* instance. The second, remote list is used to support agent mobility. Once an agent leaves its current host and migrates to another machine in the network, it is removed from the local, and placed in the remote list (along with the address of its new host *XJAF*). So when a message needs to be delivered to the agent, *Agent-Manager* will:

- Check the local list and, if the agent is available there, deliver the message directly.
- Otherwise, check the remote list and, if the agent is available there, forward the message to *AgentManager* of the agent's new host.

These steps are repeated in each *XJAF* instance in the agent's migration path, until the agent is finally located (i.e. until it appears in the *AgentManager*'s local list). This simple, yet effective technique of agent location tracking is known as the *forwarding pointers* technique [23].

ConnectionManager is the manager in charge of maintaining a network of distributed *XJAF* instances and, in combination with the previously described agent tracking technique, serves as the support for agent migration. In the earliest implementation [34], each *XJAF* instance in a network had a single other *XJAF* instance to register with, forming a tree-like structure. This organization, however, was characterized by a single point of failure – if one instance breaks, the whole tree is divided into two sets of mutually unreachable instances. Recently [20, 21], this organizational structure has been replaced with a fully-connected graph. A new type of a mobile agent, named *ConnectionAgent* was added to the system, with the job of building and maintaining the graph in an efficient fashion. This new approach of organizing *XJAF* instances is shown [21] to be fault-tolerant to unexpected failures, enabling each remaining *XJAF* to have the correct overview of the network state, regardless of the number of failures.

MessageManager provides the messaging infrastructure. It supports inter-agent communication via the exchange of *KQML* messages [8]. A *KQML* message sent from one agent to another is embedded into a *JMS* message [14]. *MessageManager* then broadcasts the message to all *XJAF* instances that have previously subscribed to this service, but only the instance containing the target agent will process the message and extract *KQML* content from it. In the ongoing work of increasing the interoperability of *XJAF*, the *KQML*-based messaging system will be replaced by the *de facto* standard *FIPA ACL* [10].

An important aspect of each *MAS* is security. In terms of the agent technology, security features are used to protect both agents and the *MAS* itself from malicious attacks, to keep the confidentiality of exchanged messages, etc. In *XJAF*, these features are offered by *SecurityManager*. And since often there is a significant computational overhead associated with security (e.g. encryption/decryption of messages), the security features are not applied automatically, but can rather be included on-demand, through an *API* exposed by the manager.

XJAF includes a *service* sub-system, where the service is a reusable software component managed by *ServiceManager*. The basic idea is to expose common tasks, such as file management, in form of services that can be directly accessed and used by agents. This approach simplifies the agent development process and supports the development of lighter agents (in terms of size). The list of services is not fixed and can be expanded as needed.

XJAF agents expose their capabilities in form of *tasks*. A task includes a detailed description of a single functionality offered by the agent. It incorporates types and names of input parameters as well as of the returned value. The list of tasks offered by the agents is maintained by *TaskManager*. External clients of the system can ask for a task execution. In response, *TaskManager* will find the most suitable agent for the given task, and then send it an appropriate message. For interoperability reasons, the format of task descriptions is based on the standardized and widely-used *W3C XML Schema language* [37].

The main advantage of *XJAF* over other existing *MAS* implementation is in its use of the Java EE technology. Java EE has been endorsed by large business enterprises as the main tool for building large-scale, scalable, secure, and reliable software. As such, it represents an excellent platform for *MAS* development. Immediate direct benefits of this approach are shorter development time, standards compliance, and harnessing of advanced programming features. For example, each *XJAF* agent is a regular Java object (i.e. a *POJO*), but wrapped inside an *Enterprise JavaBean* (*EJB*) component. At runtime, the component is passed to an enterprise application server in order to employ runtime load-balancing and object pooling features.

The original *XJAF* had one serious disadvantage – it was "locked" into a particular development platform. A consequence of this problems is the lack of interoperability, in the sense that only Java-based external clients could access the system and interact with its agents. In order to overcome this issue, a new system, named *SOA-based MAS* (*SOM*) has been developed. *SOM* follows the

manager-based design approach of *XJAF*, but with managers implemented as web services. The most important improvement introduced by the "switch" to the *SOA*-based design is increased interoperability: external clients and third-party tools can interact with *SOM* and its agents through web service interfaces, i.e. in a familiar fashion, and using the standardized communication protocol.

Unfortunately, the *SOA*-based design of *SOM* introduced a new, major issue. Because the system is a specification of web services, it can be implemented using many modern programming languages. However, developing an agent that can run on any of these implementations becomes almost an impossible task. In order to solve this problem, a new agent-oriented programming language, named *Agent LAnguage for SOM* (*ALAS*) has been developed. Its main features and functionalities are described in the following section.

4. Main features of *ALAS*

Originally, in [22], *ALAS* and its accompanying set of tools (in further text, the *ALAS platform*) were primarily aimed at the development of agents for different *SOM* implementations. One of the main characteristics of the *ALAS* platform is *hot compilation*: when an agent arrives to an instance of *SOM* implemented in a certain programming language *X*, its *ALAS* source code is transformed on-the-fly into the source code written in *X*. The generated source code is then forwarded to the native compiler, if any, to produce the executable code for the target platform.

Since the original proposal, the functionality of *ALAS* platform has been broadened to include support for other *MAS* implementations, such as *JADE*. According to the classification of agent mobility presented in [24] and described earlier, this means that the *ALAS* platform has been upgraded from delivering agent-regeneration to supporting true heterogeneous agent mobility. The main goal of *ALAS* is, therefore, to create an agent-oriented programming that hides the complexity of the overall agent-development process from developers, and, at the same time, operates regardless of the underlying *MAS*.

The *ALAS* platform has been designed ground-up with the idea of heterogeneous agent mobility in mind. The entire process of transforming the *ALAS* source code into the executable code for the target platform is shown in Fig. 1. In the first step, the agent source code written in *ALAS* is parsed to produce an abstract syntax tree. The tree is then fed into the *VM selector* which associates it with the proper *ALAS standard library*. The standard library includes utility functions for common operations, such as string processing, file management, and network connections. To support the idea of heterogeneous agent mobility, the library was re-implemented for each of the supported target languages (currently, Java and Python). The output of this step is fed into the *MAS selector* which replaces *MAS*-specific *ALAS* instructions with native *API* calls. *MAS* selector produces a fully-functional source code of the agent for the target *MAS* which is, finally, sent to the native language compiler (if any) to produce the executable code.

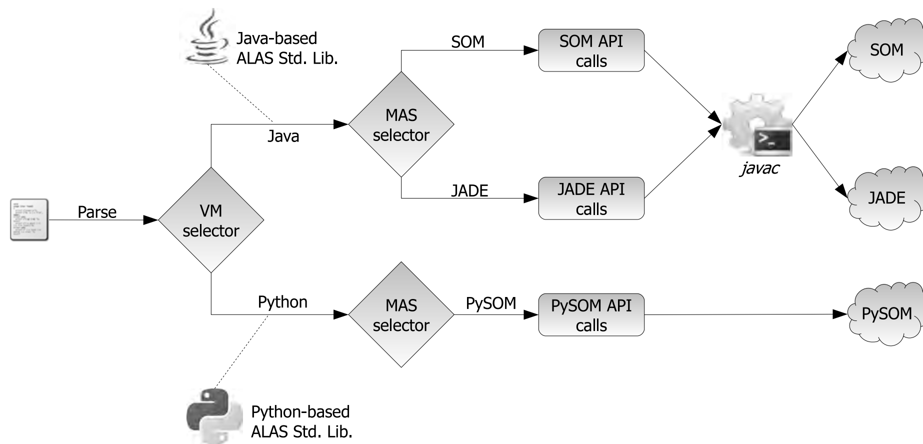


Fig. 1. The process of compiling *ALAS*-based source code of an agent into the executable code for the target platform

Protection from malicious attacks is an important issue, especially in systems that employ agent mobility. To protect the agent code during the migration process, *code certificates* can be used [35]. A certificate holds the *hash* of the *ALAS* source code, as well as the *digital signature* of the agent's internal state. The security check is performed before the parsing step, and if it fails, the agent is discarded for unauthorized modifications.

Common, non agent-specific programming constructs, such as *if-then-else* and *switch* control statements, *while*, *do-while*, and *for* loops, are also supported. Their syntax is, like in many modern programming languages, based on the syntax of the C programming language.

In the current stage, *ALAS* can be used for developing purely reactive agents. These assume agents that execute actions in response to some external events, such as messages received from other agents. The support for *BDI*-style architecture is planned for a latter stage.

ALAS compiler handles a single *compilation unit* at a time, which includes definition of one agent, and an optional package declaration:

```
CompilationUnit = [ Package ] AgentDefinition <EOF> ;
Package         = "package" Name ";" ;
Name            = Identifier { "." Identifier } ;
```

ALAS packages serve the same purpose as packages (or namespaces) in traditional procedural and *OO* programming languages. They provide the means for distinguishing between agents that have the same name, for logical grouping of related agents, etc.

4.1. Services

The main language construct for exposing agent's behavior is a *service*. In *ALAS*, a service is a functionality that the agent offers to others, and can be seen as a counterpart of a public method in *OOP*. External entities can ask for a service execution by sending an appropriate message to the agent. A *function*, on the other hand, is local and its primary use is to break large service implementations into smaller logical units. It cannot be accessed by external entities and is, therefore, a counterpart of a private method in *OOP*.

Agent definition in *ALAS* consists of the agent name and the agent body, which, in turn, is defined as a set of states, services, and functions:

```
AgentDefinition  = "agent" Identifier
                   "{" { AgentBodyDef } "}" ;
AgentBodyDef     = ( LookAhead(3) AgentState |
                     "service" Function |
                     "services" "{" { Function } "}" |
                     LookAhead(3) Function ) ;
Function         = ResultType Identifier ParamList Block ;
ParamList        = "(" Param { "," Param } ")" ;
Param            = Type Name ;
```

Agent service definition begins with the keyword *service*, followed by the return type, unique name of the service, formal parameter list, and a body. As a shortcut (i.e. to avoid typing the *service* keyword for each new service), several services can grouped under a single *services* block. When *SOM* is used as the target platform, a separate *XML*-based task description is produced for each defined service.

An important thing to note about agent services is that method overloading from *OOP* languages cannot be applied. That is, an agent cannot expose two or more services under the same name, even if formal parameters differ. Although the syntax of a service definition resembles the syntax of a method definition, services are actually message handlers. In standardized agent communication, the order of values passed as the message content does not (or, should not) matter. For example, the following code represents a *KQML* message that *AgentA* sends to *AgentB* asking for the execution of its service *PrintSum*. The message includes values 5 and 6 for the service's two integer parameters, *a* and *b*, respectively:

```
(achieve
  :sender AgentA
  :receiver AgentB
  :language XML
  :content "<service>
      <name>PrintSum</name>
      <args>
        <arg name="a" type="int"><![CDATA[5]]></arg>
        <arg name="b" type="int"><![CDATA[6]]></arg>
```

```
    </args>
  </service>" )
```

So even if the order of *arg* tags changes, the agent is still asked to execute the same service.

Unlike services, functions can be overloaded.

4.2. Agent runtime state

ALAS is a strongly and statically typed language. In addition to *void*, the language supports *byte*, *short*, *int*, *long*, *float*, *double*, *boolean*, and *char* primitive data types. These types match the appropriate primitive data types of the Java programming language. Currently, the only supported complex data type is *String*, while the support other complex data types (e.g. for artifacts modeling) will be included later.

The *runtime state* of an *ALAS* agent is represented by a set of *persistent* and *temporary* properties. During the migration process, values of persistent properties are automatically saved before the agent leaves its host *MAS*, and later automatically restored once it reaches the target *MAS*. Temporary properties, on the other hand, should be used only to store values that are not supposed to be transferred along with the agent.

The syntax for defining runtime state of an *ALAS* agent is as follows:

```
AgentState = ( "state" "{" { LocalVar ";" } "}" |
               LocalVar ";" ) ;
LocalVar   = Type Var { "," Var } ;
Var        = Identifier [ "=" Expression ] ;
```

Any property defined within the *state* block will be considered persistent. Properties defined outside of this block will be considered temporary.

As show, expressions can be used to set initial values of properties along with declarations. If more complex initialization steps are required, a function with the following signature can be defined: *void initialize()*. This special-purpose, parameterless function is automatically invoked during the agent's startup, and before any other function or service. Therefore, it corresponds to a constructor in traditional OO programming languages (or the $__init__$ method in Python).

4.3. Support for agent mobility

In order to support agent mobility, *ALAS* includes two programming instructions: *copy* and *move*. The first instruction makes a clone of the original agent in the target *MAS*, which means that the agent continues to operate in the source *MAS*. This instruction can appear at any point in a service or function implementation. The *move* command, on the other hand, physically moves the agent to the target *MAS*, which means that the original agent is disposed. This is why the *move* instruction can be only the very last instruction in a service implementation. The syntax of *copy* and *move* instructions is as follows:

```
MoveStatement = ( "copy" | "move" )
                  "(" Expression [ "," Expression
                  { "," MoveArg } ] ")" ";" ;
MoveArg       = StringLiteral "=" Expression ;
```

Each instruction accepts at least one parameter, an expression that represents the target *MAS*. The value can be the concrete name of the *MAS*, or the physical network address of the machine that hosts it, in the format $host : port$. The second expression, if specified, is the name of the agent's service that should be automatically invoked once the target *MAS* is reached. Finally, a set of values that should be passed to the service can be specified, in the form of $ParameterName = Value$.

Once the migration process is initialized, the agent is serialized into an *XML* stream. The transferred data includes:

- The original (*ALAS*) source code of the agent.
- Identifier of the agent's originating *MAS* (included for convenience reasons).
- Agent's persistent properties.
- Name of the service to be automatically invoked.
- The set of arguments for the service.

The presented set of *ALAS* features already offers the possibility of writing powerful agents. With the backing of the *ALAS* platform, these agents are able to operate in networks consisting of Java EE-based *SOM*, Python-based *SOM*, and *JADE* instances, as shown in the next section.

5. A mobile *TimeSync ALAS* agent

The example of using the *ALAS* platform presented in this section is intended to serve as a proof of concept. The experiment will demonstrate how an *ALAS*-based agent operates and migrates in a truly heterogeneous network environment.

For the purpose of this experiment, a *TimeSync* agent was developed. The agent, upon receiving an appropriate message, visits all *MAS*s in the network and synchronizes their timers. The message sent to the agent includes a comma-separated list of network addresses to be visited, and the value of time (of type *double*) to be set in each *MAS*. Once it synchronizes the timers of all systems, the agent returns to its originating *MAS*.

The network includes 3 different multi-agent systems:

- A Java EE-based implementation of *SOM*
- A Python-based implementation of *SOM*, named *PySOM*
- *JADE* version 4.1

Each system features a module that accepts a serialized form of the *ALAS* agent, de-serializes it, invokes the *ALAS* compiler, restores the agent's runtime state, and then sends it the message for service execution.

Python was chosen as the second implementation platform for *SOM* in this experiment because the language itself is very different from Java in many aspects, such as the syntax, dynamic vs. static typing, multi-paradigm vs. OO paradigm, etc. In this way, the experiment will show that the *ALAS* platform is not tied to the Java platform.

JADE was included in order to demonstrate the ability of *ALAS* agents to operate in heterogeneous environments. The system was extended with an *ALAS* platform plug-in that performs the aforementioned steps from accepting the agent, to requesting service execution.

The full source code of the *TimeSync* agent written in *ALAS* is shown in Listing 5.1.

Listing 5.1. Full source code of the mobile *TimeSync ALAS* agent, capable of operating in a network consisting of *SOM*, *PySOM*, and *JADE* instances

```
1   package example.agents;
2
3   agent TimeSync {
4     state { String startingHome; }
5     String next, remaining;
6
7     service void SyncTimers(String hosts, double time) {
8       if (startingHome == null)
9         startingHome = host(); // remember the starting point
10      else if (startingHome.equals(host())) { // am I back home?
11        log("I'm back!");
12        startingHome = null;
13        return; }
14      // apply the time
15      log("Setting the system time to ", time);
16      applySystemTime(time);
17      // go to the next host
18      if (hosts.length() == 0) // no more hosts, go back home
19        next = startingHome;
20      else
21        parseHosts(hosts);
22      move(next, "SyncTimers", "hosts"=remaining, "time"=time); }
23
24    void parseHosts(String hosts) {
25      int n = hosts.indexof(",");
26      if (n == -1) {
27        next = hosts;
28        remaining = "";
29      } else {
30        next = hosts.substring(0, n);
31        remaining = hosts.substring(n + 1); } } }
```

Line 1 sets the agent's package to *example.agents*, and the agent definition starts at line 3. Lines 4 and 5 define the agent's runtime state as a set of three

String properties: *startingHome*, *next*, and *remaining*. The first property is persistent. It holds the identifier of the agent's starting *MAS*, i.e. the *MAS* that was hosting the agent when it received the request to run the time synchronization process. This value is used by the agent to return home once it finishes the process. The latter two properties are temporary, and are used by the *parseHosts()* function described later.

TimeSync agent exposes a single service (lines 7–22) called *SyncTimers*. The service has no return value, and accepts two parameters: a comma-separated list of *MAS* instances the agent needs to visit, and the time value to be set in each *MAS*. The starting *MAS* is stored persistently in lines 8 and 9. As noted earlier, when a mobile agent arrives to a new *MAS*, its runtime state is restored after the initialization, but before any service execution is requested. This means that the expression *startingHome == null* will resolve to true only at the agent's starting *MAS*. Lines 10–13 are used to detect if the agent has returned back home. For this evaluation, the *host()* library function is invoked, returning the address of the agent's current host.

Line 16 invokes a function called *applySystemTime()*, which is used to set the system time to the given value. For security purposes, however, this is a dummy functions. *ALAS* agents are not actually able to change the system time.

Lines 18–21 are used to extract the next *MAS* that needs to visited. Line 18 in particular demonstrates the usage of the *ALAS* standard library. The *hosts* parameter is of *ALAS* String complex type, which offers a set of functions for string manipulation, including (among others):

- *int length()* – returns the length of the string.
- *int indexof(String sub)* – returns the index of the first occurrence of *sub* within the string, or -1 if the parameter does not occur. The first character in a string has the index of 0.
- *String substring(int start [, int end])* – returns the substring of the string, starting with index *start* (inclusive), and until the index *end* (exclusive). If *end* is not specified, the call corresponds to *str.substring(start, str.length())*.

Because the String type is included both in Java and Python, the *VM* selector (Fig. 1) replaces these calls with calls to appropriate native string manipulation methods.

The *SyncTimers* service utilizes a helper function, named *parseHosts()* (lines 24–31) which extracts the next *MAS* from the comma-separated list. The next *MAS* is stored into the temporary property *next*, while the updated list (e.g. the list without the extracted *MAS*) is stored into the second temporary property, *remaining*.

The agent's final step is to move to the next *MAS*, and it does so by invoking the *move* instruction (line 22). The instruction's arguments indicate that, once it reaches the target, the agent should be asked to again execute the *SyncTimers* service with parameter values *remaining* and *time*.

5.1. Running the *TimeSync* agent

The network used in this experiment consists of three *MAS*s, i.e. a single instance of each of the *SOM*, *PySOM*, and *JADE*. The network addresses of the systems are, respectively, $192.168.0.1:8081$, $192.168.0.2:8081$, and $192.168.0.3:8081$. Initially, the agent is located in the *SOM* instance.

A client can ask for the *SyncTimer* service execution by sending the following *KQML* message to the agent:

```
(achieve
  :sender ALAS_IDE
  :receiver TimeSync
  :language XML
  :content "<service>
      <name>SyncTimers</name>
      <args>
        <arg name="hosts" type="String">
          <![CDATA[192.168.0.2:8081,192.168.0.3:8081]]>
        </arg>
        <arg name="time" type="double">
          <![CDATA[40893,639141169]]>
        </arg>
      </args>
    </service>")
```

The sender of this message is *ALAS IDE*, which, among common features such as syntax highlighting, provides the means for specifying the target *MAS*, in form of $type@address$. In this scenarion, the compilation process will automatically load the agent in the specified *MAS*.

Once the agent executes the service, it will set the time of the current *MAS*, extract the network address of the next *MAS* to visit (*PySOM* instance at $192.168.0.2:8081$), and initiate the migration process. The migration process will serialize the agent, resulting in the following *XML* stream:

```
<?xml version="1.0" encoding="UTF-8"?>
<alas xmlns="http://alasagents.org/SerializedAgent"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://alasagents.org/SerializedAgent
      SerializedAgent.xsd ">
  <source>
    <![CDATA[package example.agents; agent ...]]>
  </source>
  <home>192.168.0.1:8081</home>
  <state>
    <property name="startingHome" type="String">
      <![CDATA[192.168.0.1:8081]]>
    </property>
  </state>
  <service>SyncTimers</service>
```

```
  <args>
    <arg name="hosts" type="String">
      <![CDATA[192.168.0.3:8081]]>
    </arg>
    <arg name="time" type="double">
      <![CDATA[40893,639141169]]>
    </arg>
  </args>
</alas>
```

The type information for persistent properties and service arguments may be redundant, but is included in the stream for convenience reasons.

Running the agent in *PySOM* The *PySOM* instance accepts the serialized stream, extracts the agent source code, and passes it to the *ALAS* compiler. The agent's persistent property is then restored to $192.168.0.1 : 8081$. Finally, the system sends a message to the agent, asking for the execution of the *SyncTimers* service. The generated Python source code of the *TimeSync* agent is shown in Listing 5.2 [3].

Listing 5.2. Source code of the mobile *TimeSync ALAS* agent, regenerated for *PySOM*

```
1  class TimeSync:
2    def __init__(self):
3      # original, ALAS source code of the agent
4      self.__AGENT_SOURCE__ = "package example.agents; ..."
5      # comma-separated list of persistent properties
6      self.__PERSISTENT_VARS__ = "startingHome"
7      # agent state
8      self.startingHome = None
9      self.next = None
10     self.remaining = None
11
12   # handler of incoming messages
13   def onMessage(self, service):
14     if service.getName() == "SyncTimers":
15       # SyncTimers service implementation
16       def SyncTimers(hosts, time):
17         # remember the starting point
18         if self.startingHome == None:
19           self.startingHome = host()
20         else: # am I back home?
21           if self.startingHome == host():
22             alas.stdlib.pysom.Log.write("I'm back!")
23             startingHome = None
24             return
```

[3] This and subsequent listings showing transformed source code of the agent have been manually reformatted to make them more human-readable.

```
25          # apply the time
26          alas.stdlib.pysom.Log.write("Setting the ",
27            "system time to ", time)
28          alas.stdlib.pysom.Dummy.applySystemTime(time)
29          # go to the next host
30          if len(hosts) == 0: # no more hosts, go back home
31            self.next = self.startingHome
32          else:
33            self.parseHosts(hosts)
34          # prepare service parameters
35          __mv_obj_2 = time
36          __mv_obj_1 = self.remaining
37          # start the migration process
38          alas.stdlib.pysom.PySOMFacilitator.instance().move(
39            self, self.next, "SyncTimers", "hosts", __mv_obj_1,
40            "time", __mv_obj_2)
41        # execute the service
42        SyncTimers(service.get("hosts"), service.get("time"))
43        return
44
45    def parseHosts(self, hosts):
46      n = hosts.find(",")
47      if n == -1:
48        self.next = hosts
49        self.remaining = ""
50      else:
51        self.next = hosts[0:n]
52        self.remaining = hosts[n+1:]
```

The *TimeSync* agent is represented by a class which includes a constructor (lines 2–10), a function for handling incoming messages (lines 13–43), and the helper *parseHosts()* function (lines 45–52). The actual code of the *SyncTimers* services is inserted as an inner function of *onMessage()* (lines 16–43), right under the conditional statement that determines the name of the requested service. The actual call to this inner function is made in line 42.

Agents that wish to leave a *PySOM* instance can do so by calling the *move* method of *PySOMFacilitator*. The method relies on Python reflection features for extracting the necessary runtime information about the agent, such as its original, *ALAS* source code stored in line 4, values of persistent properties listed in line 6, etc.

As it can been seen from the given source code, the *ALAS* standard library for *PySOM* is available under the *alas.stdlib.pysom* package.

After executing the service here, the agent will move to the *JADE* instance.

Running the agent in *JADE* Once the *TimeSync* agent reaches *JADE*, its *ALAS* source code will be processed to produce *JADE behavior* class, shown in Listing 5.3. *MyBehavior* is defined as an inner class of the *TimeSync* class, which represents the actual agent. State properties, original *ALAS* source code, and the list of persistent properties are all defined in this agent class.

Listing 5.3. Source code of the mobile *TimeSync ALAS* agent, regenerated for *JADE*

```
1  private class MyBehavior
2      extends jade.core.behaviours.CyclicBehaviour {
3    private TimeSync agent;
4
5    public MyBehavior(TimeSync agent) { this.agent = ag; }
6
7    @Override public void action() {
8      // wait for a message
9      jade.lang.acl.ACLMessage msg = agent.receive();
10     if (msg == null) { block(); return; }
11
12     // extract service description from the msg content
13     alas.stdlib.java.common.migration.ServiceDesc content = null;
14     try {
15       content = (alas.stdlib.java.common.migration.ServiceDesc)
16         msg.getContentObject();
17     } catch (jade.lang.acl.UnreadableException ex) { return; }
18
19     if (content.isService("SyncTimers")) {
20       // SyncTimers service implementation
21       class Service_SyncTimers {
22         void SyncTimers(String hosts, double time) {
23           // remember the starting point
24           if (startingHome == null)
25             startingHome = alas.stdlib.java.common.Facilitator.
26               instance().getHome();
27           // am I back home?
28           else if (startingHome.equals(alas.stdlib.java.
29               common.Facilitator.instance().getHome())) {
30             alas.stdlib.java.common.Log.write("I'm back!");
31             startingHome = null;
32             return; }
33           // apply the time
34           alas.stdlib.java.common.Log.write("Setting the ",
35             "system time to ", time);
36           alas.stdlib.java.common.Dummy.applySystemTime(time);
37           // go to the next host
38           if (hosts.length() == 0) // no more, go back home
39             next = startingHome;
40           else
41             parseHosts(hosts);
42           // prepare service parameters
43           Object mv_obj2 = time;
44           Object mv_obj1 = remaining;
45           // start the migration process
46           alas.stdlib.java.jade.JADEFacilitator.instance().
47             move(this, next, "SyncTimers",
```

```
48              "hosts", mv_obj1, "time", mv_obj2); } }
49        // execute the service
50        String hosts = content.get("hosts", String.class);
51        double time = content.get("time", double.class);
52        new Service_SyncTimers().SyncTimers(hosts, time);
53        return; } }
54
55    private void parseHosts (String hosts){
56      int n = hosts.indexOf(",");
57      if (n == -1) {
58        next = hosts;
59        remaining = "";
60      } else {
61        next = hosts.substring(0, n);
62        remaining = hosts.substring(n + 1); } } }
```

Inside *JADE*, the agent will wait for an incoming message (lines 9 and 10), and then extract the message content (lines 13–17). The content is defined as a *ServiceDesc* class, which stores all the information about the service the agent is asked to execute. The actual source code of the *SyncTimers* service is inserted as the inner *Service_SyncTimers* class (lines 21–48), and it's invoked using lines 50–52.

Two root packages, *alas.stdlib.java.common* and *alas.stdlib.java.jade*, make up the *ALAS* standard library for *JADE*. All classes under the first package are shared between all Java-based *MAS*s (in this case, *SOM* and *JADE*) and include functionalities that are architecture-independent. The second package incorporates classes that implement *JADE*-specific behavior (e.g. *JADEFacilitator*).

Similarly as with *PySOM*, the *move* instruction (lines 46–48) relies on Java reflection *API* to extract and serialize agent's runtime properties.

After executing the service in *JADE*, the agent determines that there are no more *MAS*s to visit, and returns to its home – the *SOM* instance.

Running the agent in *SOM* The *TimeSync* source code produced for *SOM* is shown in Listing 5.4. Again, the agent is defined as a regular Java class implementing the *SOM*-specific *Agent* interface that represents all agents.

Listing 5.4. Source code of the mobile *TimeSync ALAS* agent, regenerated for *SOM*

```
1  public class TimeSync
2        implements xjafs.agentmanager.ejb.interfaces.Agent {
3    // original, ALAS source code of the agent
4    private final String __AGENT_SOURCE_ =
5      "package example.agents; ...";
6    // comma-separated list of persistent properties
7    private final String __PERSISTENT_VARS_ = "startingHome";
8    // agent state
9    private String startingHome, next, remaining;
```

```
10
11   // handler of incoming messages
12   @Override
13   public void onKQMLMessage(String kqml, String agentID) {
14     // unmarshall the KQML message
15     final xjafs.agentmanager.ejb.utils.xml.kqml.KqmlMessage msg =
16     xjafs.agentmanager.ejb.utils.XMLMapper.unmarshallKQML(kqml);
17
18     if (message.getCommand().equals("SyncTimers")) {
19       // SyncTimers service implementation
20       class Service_SyncTimers {
21         void SyncTimers(String hosts, double time) {
22           // remember the starting point
23           if (startingHome == null)
24             startingHome = alas.stdlib.java.common.Facilitator.
25               instance().getHome();
26           // am I back home?
27           else if (startingHome.equals(alas.stdlib.java.common.
28               Facilitator.instance().getHome())) {
29             alas.stdlib.java.common.Log.write("I'm back!");
30             startingHome = null;
31             return; }
32           // apply the time
33           alas.stdlib.java.common.Log.write("Setting the ",
34             "system time to ", time);
35           alas.stdlib.java.common.Dummy.applySystemTime(time);
36           // go to the next host
37           if (hosts.length() == 0) // no more, go back home
38             next = startingHome;
39           else
40             parseHosts(hosts);
41           // prepare service parameters
42           Object __mv_obj_$2 = (time);
43           Object __mv_obj_$1 = (remaining);
44           // start the migration process
45           alas.stdlib.java.som.SOMFacilitator.instance().move(
46             this, next, "SyncTimers",
47             "hosts", __mv_obj_$1, "time", __mv_obj_$2); } }
48       // execute the service
49       SyncTimers task = XMLMapperSyncTimers.
50         unmarshallSyncTimers(msg.getContent());
51       new Service_SyncTimers().SyncTimers(task.getHosts(),
52         task.getTime());
53       return; } }
54
55   private void parseHosts (String hosts) { ... ommitted ... }
56 }
```

SOM agents communicate by exchanging *KQML* messages, and the system relies on *JAXB marshalling/unmarshalling* [19] for message serialization/dese-

rialization (lines 15 and 16). Similarly as with *JADE*, in this *SOM*-specific definition of *TimeSync*, the *SyncTimers* service implementation is provided in form of an inner class (lines 20–47), and is invoked using lines 49–52.

This experiment demonstrates how *ALAS* can be used to develop agents in the *write once, run anywhere* manner. The *ALAS*-based *TimeSync* agent is implemented once to solve a specific problem, and is then able to work in *SOM*, *PySOM*, and *JADE* instances without any interventions on the developer's part. Therefore, the main goal behind the development of *ALAS* has been achieved.

6. Conclusions and future work

XJAF is *FIPA*-compliant *MAS* developed by the authors of this paper. It is designed as a pluggable, manager-based architecture, which allows for easy additions of new functionalities. The system is implemented in Java EE, today's leading development platform for building large-scale, scalable, secure, and reliable software. In the course of improving its interoperability, *XJAF* has recently been redesigned as a service-oriented architecture. The resulting system, named *SOM*, follows the manager-based approach of *XJAF*, but with managers being implemented as web services. The main advantage of this approach is that external clients and third-party tools can use *SOM* and interact with its agents through *SOAP*, the standardized communication protocol.

SOM is a conceptual specification of web services, and it can be implemented using many modern programming languages. But, this poses a major problem: an agent written for, e.g., Java-based implementation of *SOM* cannot move to a Python-based implementation. In order to overcome this issue, a new agent-oriented programming language named *ALAS*, has been proposed. The two main goals of *ALAS*, as originally described in [22], are:

1. To provide developers with programming constructs that simplify the overall complexity of agent development.
2. To include tools for agent code regeneration, and enable migration across *SOM* instances implemented using different programming languages.

Since this original proposal, however, the design goal of *ALAS* has been extended. The language itself and its accompanying set of tools have been upgraded to support true *heterogeneous* agent mobility. Unlike the agent regeneration, heterogeneous mobility assumes that agents are able to migrate across the network consisting of *MAS* instances that offer different sets of *API*s *and* are implemented using different programming languages. This, obviously, is more difficult problem to solve, as it requires both the regeneration of agent's executable code and modifications of the code in order to adapt to the *API* of the underlying *MAS*.

As shown in this paper, the desired goal has been achieved. *ALAS* agents are able to seamlessly, without any interventions on the developer's part, operate inside Java-based *SOM*, Python-based *SOM*, and *JADE* instances. The

presented experiment demonstrates how the agent's executable code is regenerated and modified transparently, on-the-fly, to suit the requirements of the underlying *MAS*. This enables the developer to focus on solving the concrete problem, and, in a truly platform-independent manner, disregard information about the target *MAS*. To the best of our knowledge, there currently exists no other agent-oriented programming language that offers this significant benefit.

Future research directions will be concentrated onto improving the expressive power of *ALAS* and extending its standard library of functions. This will simplify the agent development process even further.

The support of other *MAS*s is planned as well.

In the long run, the language will be enriched with programming constructs for defining agent's beliefs, desires, intentions, and goals, in order to support the development of *BDI*-style agents.

Acknowledgments. This work is partially supported by Ministry of Education and Science of the Republic of Serbia, through project no. OI174023: "Intelligent techniques and their integration into wide-spectrum decision support".

References

1. Aranda, G., Palanca, J., Criado, N.: SPADE user's manual. http://spade.gti-ia.dsic.upv.es/manuals/html-chunk/index.html (October 2007), retrieved on December 7, 2011
2. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing multi-agent systems with JADE. John Wiley and Sons (2007)
3. Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology). John Wiley & Sons (2007)
4. Bradshaw, J., Breedy, M., Groth, P., Hill, G., Jeffers, R., Mitrovich, T., Suri, N.: An overview of the NOMADS mobile agent system. In: 2nd International Symposium on Agent Systems and Applications, ASA/MA2000 (September 2000)
5. Dastani, M.: 2APL: a practical agent programming language. Autonomous Agents and Multi-Agent Systems 16(3), 214–248 (2008)
6. Dastani, M., van Riemsdijk, B., Dignum, F., Meyer, J.J.C.: A programming language for cognitive agents - goal directed 3APL. In: Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.) PROMAS. Lecture Notes in Computer Science, vol. 3067, pp. 111–130. Springer (2003)
7. Davies, W.H.E., Edwards, P.: Agent-K: An integration of AOP and KQML. In: Proceedings of the Third International Conference on Information and Knowledge Management (1994)
8. Finin, T., Fritzson, R., McKay, D., McEntire, R.: KQML as an agent communication language. In: Proceedings of the third international conference on Information and knowledge management. pp. 456–463. CIKM '94, ACM, New York, NY, USA (1994), http://doi.acm.org/10.1145/191246.191322
9. FIPA abstract architecture specification. http://www.fipa.org/specs/fipa00001/SC00001L.pdf (2002), retrieved on December 7, 2011
10. FIPA ACL message structure specification. http://www.fipa.org/specs/fipa00061/SC00061G.pdf (2002), retrieved on December 7, 2011

11. FIPA homepage. http://www.fipa.org/, retrieved on December 7, 2011
12. Fortino, G., Garro, A., Russo, W.: Achieving mobile agent systems interoperability through software layering. Information and software technology 50(4), 322–341 (2008)
13. Grimstrup, A., Gray, R.S., Kotz, D., Carvalho, M.M., Cowin, T.B., Chacón, D.A., Barton, J., Garrett, C., Hofmann, M.: Toward interoperability of mobile-agent systems. In: International symposium on mobile agents. pp. 106–120 (2002)
14. Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K.: Java Message Service (JMS) specification. http://www.oracle.com/technetwork/java/jms/index.html (April 2002), retrieved on December 7, 2011
15. Hindriks, K.V.: Programming rational agents in GOAL. In: El Fallah Seghrouchni, A., Dix, J., Dastani, M., Bordini, R.H. (eds.) Multi-Agent Programming:, pp. 119–157. Springer US (2009)
16. Ivanović, M., Mitrović, D., Budimac, Z., Vidaković, M.: Metadata harvesting learning resources – an agent-oriented approach. In: Proceedings of the 15th International Conference on System Theory, Control and Computing (ICSTCC 2011). pp. 306–311 (October 2011)
17. JADE homepage. http://jade.tilab.com/, retrieved on December 7, 2011
18. Jason homepage. http://jason.sf.net/, retrieved on December 7, 2011
19. Java Architecture for XML Binding (JAXB) homepage. http://www.oracle.com/technetwork/articles/javase/index-140168.html, retrieved on December 7, 2011
20. Mitrović, D., Budimac., Z., Ivanović, M., Vidaković, M.: Improving fault-tolerance of distributed multi-agent systems with mobile network-management agents. In: Proceedings of the International Multiconference on Computer Science and Information Technology. vol. 5, pp. 217–222 (October 2010)
21. Mitrović, D., Budimac, Z., Ivanović, M., Vidaković, M.: Agent-based approaches to managing fault-tolerant networks of distributed multi-agent systems. Multiagent and Grid Systems 7(6), 203–218 (December 2011)
22. Mitrović, D., Ivanović, M., Vidaković, M.: Introducing ALAS: a novel agent-oriented programming language. In: Simos, T.E. (ed.) Proceedings of Symposium on Computer Languages, Implementations, and Tools (SCLIT 2011) held within International Conference on Numerical Analysis and Applied Mathematics (ICNAAM 2011). pp. 861–864. AIP Conf. Proc. 1389 (September 2011), iSBN 978-0-7354-0956-9
23. Moreau, L.: Distributed directory service and message router for mobile agents. Science of Computer Programming 39(2–3), 249–272 (2001)
24. Overeinder, B.J., Groot, D.R.A.D., Wijngaards, N.J.E., Brazier, F.M.T.: Generative mobile agent migration in heterogeneous environments. Scalable computing: practice and experience 7(4), 89–99 (2006)
25. Pinsdorf, U., Roth, V.: Mobile agent interoperability patterns and practice. In: Proceedings of the 9th IEEE international conference on engineering of computer-based systems. pp. 238–244 (2002)
26. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: de Velde, W.V., Perram, J.W. (eds.) MAAMAW. Lecture Notes in Computer Science, vol. 1038, pp. 42–55. Springer (1996)
27. Schmidt, D.C.: Model-driven engineering. Published by IEEE Computer Society (February 2006)
28. Shoham, Y.: Agent-oriented programming. Artificial Intelligence 60(1), 51–92 (1993)
29. SPADE homepage. http://code.google.com/p/spade2/, retrieved on December 7, 2011

30. Suri, N., Bradshaw, J., Breedy, M.R., Groth, P.T., Hill, G.A., Jeffers, R.: Strong mobility and fine-grained resource control in NOMADS. In: Kotz, D., Mattern, F. (eds.) Proceedings of the Second international Symposium on Agent Systems and Applications and Fourth international Symposium on Mobile Agents. Lecture Notes In Computer Science, vol. 1882, pp. 2–15 (September 2000)
31. Tapia, D.I., Bajo, J., Corchado, J.M.: Distributing functionalities in a SOA-based multi-agent architecture. In: Demazeau, Y., Pavón, J., Corchado, J.M., Bajo, J. (eds.) 7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009), Advances in Intelligent and Soft Computing, vol. 55, pp. 20–29. Springer Berlin / Heidelberg (2009)
32. Tapia, D.I., Rodríguez, S., Bajo, J., Corchado, J.M.: FUSION@, a SOA-based multi-agent architecture. In: Corchado, J.M., Rodríguez, S., Llinas, J., Molina, J. (eds.) International Symposium on Distributed Computing and Artificial Intelligence 2008 (DCAI 2008), Advances in Soft Computing, vol. 50, pp. 99–107. Springer Berlin / Heidelberg (2009)
33. Thomas, S.R.: The PLACA agent programming language. In: Wooldridge, M., Jennings, N.R. (eds.) ECAI Workshop on Agent Theories, Architectures, and Languages. Lecture Notes in Computer Science, vol. 890, pp. 355–370. Springer (1994)
34. Vidaković, M.: Extensible Java based agent framework. Ph.D. thesis, Faculty of Technical Sciences, University of Novi Sad, Serbia (2003)
35. Vidaković, M., Sladić, G., Konjović, Z.: Security management in J2EE based intelligent agent framework. In: Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (SEA 2003). pp. 128–133 (November 2003)
36. World Wide Web Consortium (W3C) SOAP version 1.2. http://www.w3.org/TR/soap/, retrieved on December 7, 2011
37. World Wide Web Consortium (W3C) XML Schema. http://www.w3.org/XML/Schema, retrieved on December 7, 2011
38. Winikoff, M.: JACK Intelligent Agents: an industrial strength platform. In: Bordini, R.H., Dastani, M., Dix, J., Fallah-Seghrouchni, A.E. (eds.) Multi-Agent Programming, Multiagent Systems, Artificial Societies, and Simulated Organizations, vol. 15, pp. 175–193. Springer (2005)
39. Wooldridge, M., Jennings, N.: Agent theories, architectures, and languages: A survey. In: Wooldridge, M., Jennings, N. (eds.) Intelligent Agents, Lecture Notes in Computer Science, vol. 890, pp. 1–39. Springer Berlin / Heidelberg (1995)
40. Wooldridge, M., Jennings, N.: Intelligent agents: Theory and practice. Knowledge Engineering Review 10, 115–152 (1995)
41. XMPP standards foundation homepage. http://xmpp.org/, retrieved on December 7, 2011

Dejan Mitrović is a teaching and research assistant at Faculty of Sciences, University of Novi Sad, Serbia. He graduated in 2006 (Informatics), and received master's degree (Computer Science) in 2008, enrolling the PhD studies afterwards. He published 11 research papers on software agents, multi-agent systems, and distributed computing. He is a member of several science research projects.

Mirjana Ivanović holds position of full professor since 2002 at Faculty of Sciences, University of Novi Sad, Serbia. She is head of Chair of Computer Sci-

ence. She is author or co-author of 13 textbooks and of more then 230 research papers on multi-agent systems, e-learning and web-based learning, software engineering education, intelligent techniques (CBR, data and web mining), most of which are published in international journals and international conferences. She is/was a member of Program Committees of more then 80 international Conferences and is Editor-in-Chief of Computer Science and Information Systems Journal.

Zoran Budimac holds position of full professor since 2004 at Faculty of Sciences, University of Novi Sad, Serbia. Currently, he is head of Computing laboratory. His fields of research interests involve: Educational Technologies, Agents and WFMS, Case-Based Reasoning, Programming Languages. He was principal investigator of more then 20 projects and is author of 13 textbooks and more then 220 research papers most of which are published in international journals and international conferences. He is/was a member of Program Committees of more then 60 international Conferences and is member of Editorial Board of Computer Science and Information Systems Journal.

Milan Vidaković received the BSc, MSc and PhD degrees in electrical engineering from the Faculty of Technical Sciences, University of Novi Sad, in 1995, 1998 and 2003 respectively. He is a professor at Computing and Control Department, University of Novi Sad. He participated in several science projects and published more than 60 scientific and professional papers. His research interest covers web and internet programming, distributed computing, software agents, embedded systems, and language internationalization and localization.

Received: January 2, 2012; Accepted: Jun 12, 2012.

DOI:10.2298/CSIS120130032C

Language Engineering for Syntactic Knowledge Transfer

Mihaela Colhon

Departament of Computer Science, A.I.Cuza street 13,
200585 Craiova, Romania1231
mcolhon@inf.ucv.ro

Abstract. In this paper we present a method for an English-Romanian treebank construction, together with the obtained evaluation results. The treebank is built upon a parallel English-Romanian corpus word-aligned and annotated at the morphological and syntactic level. The syntactic trees of the Romanian texts are generated by considering the syntactic phrases of the English parallel texts automatically resulted from syntactic parsing. The method reuses and adjusts existing tools and algorithms for cross-lingual transfer of syntactic constituents and syntactic trees alignment.

Keywords: parallel treebank, syntactic phrase alignment, bilingual corpus, word-alignments.

1. Introduction

Probably the most important trend in linguistics in the last decade is the massive use of large natural language corpora [7]. In any Natural Language Processing system (NLP system), corpora is often used to provide empirical and statistical data [8]. Typically, NLP applications that use corpora as basic linguistic resource are Word Sense Disambiguation (WSD) programs [19] and all types of parsers. Machine Translation (MT) represents the usage of computers as tools for translating texts from a source language to a target language [35]. The vast majority of current approaches to MT systems are also corpus-based. Among these, Phrase-Based Statistical MT (PBSMT) are by far the most dominant paradigm [24]. In this case, the linguistic resource is in the form of pairs of aligned parallel texts in the Source Language (SL) and Target Language (TL).

Current practice in phrase-based translation extracts regular phrases and translation rules from word-aligned parallel texts [13] as it is well-known that more and more researchers have devoted themselves to syntax-based MT systems [12], [18], [37]. Parallel treebanks are useful not only for syntax-based MT or example-based MT but also can be exploited in statistical approaches of translation. More precisely, by providing alignments between the syntactic tree of two corresponding sentences on a sub-sentential level

(word, phrase and/or clause level) automatic derivation of syntactic transfer rules, very important in any translation study, can be obtained.

The most common type of linguistic annotation is Part-Of-Speech (POS) tagging or, more accurately, morphosyntactic tagging, that is the procedure of assigning to each word token appearing in a text its morphosyntactic description [10]. Many studies consider that POS tags contain enough syntactical information to support word abstraction in any NLP system training. For example, the search space of a translation rules database can be greatly reduced by focusing only on POS tags instead of real words [35]. A treebank is a corpus that has been grammatically annotated in order to identify and label different syntactic components [15].

The treebank generation mechanism presented in this article automatically constructs a syntactic annotated parallel corpus from a bilingual word-aligned corpus with morphosyntactic annotations. The corpus was manually word aligned, tokenized, POS-tagged and lemmatized. The English texts were processed with one of the existing English syntactic parsers[1] while, for Romanian texts, a tree generation algorithm guided by the word-alignments of the corpus was implemented. As a consequence, the algorithm for Romanian syntactic tree generation depends greatly on the word-alignments of the bilingual corpus as will be shown in the following sections.

Parallel treebanks, like the treebank described in the present paper, are successfully used in various NLP applications but their main scope is to enhance syntax-based translation performance of the corpora language pairs. Also, based on the alignment mechanism encoded in the treebank annotations, rich and robust set of translation rules for the corpus languages can be identified.

As noted by competent linguists, Romanian language is morphologically rich and relatively flexible word order language [5]. The term Morphologically Rich Languages refers to languages in which substantial grammatical information, i.e., information concerning the arrangement of words into syntactic units or cues to syntactic relations, are expressed at word level. Because of its rich morphology, the morphological markers themselves could serve as strong cues for identifying the syntactic relations between the words in the sentence. But in languages with free or flexible word-order, such as Romanian, constituency-based representations are overly constrained, this fact causing word-order choice to influence the complexity of the syntactic analysis.

The method we present here is not restricted to the pair of languages chosen for the current implementation, which are English and Romanian languages. As it will be shown, the involved methodology for the treebank

---

[1] Some of the well known English syntactic parsers are: Stanford Parser (web page: http://nlp.stanford.edu/software/lex-parser.shtml), Link Parser (web page: http://www.link.cs.cmu.edu/link/) or Minipar (web page: http://webdocs.cs.ualberta.ca/~lindek/minipar.htm).

generation works on the abstract level of syntactic components and thus, all the particular information about the two languages lexicon is discarded. Also, the presented method does not make use of any particular information regarding the grammatical rules of any of the two involved languages.

The aim of the performed experiment is to test whether it is possible to reuse the syntactic constituents of one language texts in order to annotate their translations into another language. A study of this type was successfully performed for the same pair of languages but with the intention of testing the import of syntactic relations contracted by verbs [24].

2. Parallel Treebank from Bilingual Corpus

While monolingual treebanks are widely available thanks to large-scale annotation projects (NEGRA Treebank [28], Penn Treebank [30], Prague Dependency Treebank [31], Swedish Treebank [32]), bilingual parallel corpora with syntactic tree-based annotation on both sides, so-called parallel treebanks, are quite rare.

Despite of their enormous importance, the manually generation of such linguistic resources usually implies huge efforts. Manual construction is an expensive, time-consuming and error-prone process which requires linguistic expertise in both languages in question. For this reason, there has been a lot of research on automatic generation, basically using tree-to-string MT models, (e.g. [39]), while the development of tree-to-tree based MT models, despite their potential, has suffered.

The treebank generation algorithm presented in this paper is guided by the word alignments existing between the parallel sentences of a bilingual corpus. For this reason, the generation process is strongly dependent on the quality and quantity of the word-alignments, as accordingly to the Blinker annotation guidelines [23]: "*if a word is left unaligned on the source side of a sentence pair, this implies that the meaning it carries was not realized anywhere in the target side*". From the MT usage point of view, this implies that the meaning together with all morphosyntactic information of the source word to be lost. Therefore, the more accurate the word alignments are, the better the quality of the induced syntax tree for the target part of the resulted treebank will be.

2.1. Treebank Linguistic Resources

Treebanks, as large collections of syntactically parsed sentences, are considered valuable resources not only for computational tasks such as grammar induction and automatic parsing, but also for traditional linguistic and philological pursuits as well [17].

Syntactic annotation is the practice of adding syntactic information to a text by incorporating into it markers indicating syntactic dependencies relations. In order to obtain a parallel treebank from the bilingual corpus each sentence

has to be annotated with POS data. This kind of annotation is usually resulted with a POS Tagger tool. Another type of syntactic annotations consists of syntactic phrase labels for the both parts of a bilingual corpus which are aligned usually by following the word-alignments of the corpus.

English language is the best supported language, at this moment there are many large corpora, syntactic trees resources and testing language processing tools.

Although not to the extent of the languages with greater electronic visibility, efforts have been invested by researchers in different places (Romanian, Republic of Moldova, Unites States, United Kingdom, Germany, Italy, etc.) to develop Romanian linguistic resources such as corpora, dictionaries, wordnets and collections of linguistic data in both symbolic and statistical form [6].

From the available parallel corpora, the *Acquis Communautaire* linguistic resource represents the biggest parallel corpus existent at this moment, taking into account both its size and the number of covered languages [10]. The corpus includes the total body of European Union (EU) law applicable in the EU Member States. It is available in 22 official languages (including Romanian) of the European Union. A significant part of these parallel texts have been compiled by the Language Technology Group of the European Commission into an aligned parallel corpus, called JRC-Acquis Multilingual Parallel Corpus [18]. In most bilingual corpora derived from JRC-Acquis corpus, we find English paired with a European language.

In order to make a bilingual corpus with POS annotations an appropriate linguistic resource for the presented treebank generation method, word-alignments have to be provided (manually or with automatic tools such GIZA++ [27]).

A great progress has been done in the MT development from manually crafted linguistic models to empirically learned statistical models, from word-based models to phrase-based models and from string-based to tree-based models [21].

Two segments of texts from a bitext which represent reciprocal translations make a *translation unit* [39]. A translation unit may contain, in one or both the paired languages, one or more textual units (paragraph, sentence, phrase, word). Traditionally, phrases are taken to be syntactic components of a sentence. These units can be used to generate more complex constructions in that language and based on them a new phrase-based strategy was employed in MT: instead of generating translation of individual words from the source language, generate translations of the phrases and assemble the final translation by a permutation of these [39].

In literature, there are several translation theories formalized on parallel corpora with word-level alignments. In [11] is defined a generative process by means of which a symbol tree over a target language is derived from a string of source symbols. In order to distinguish between good and bad derivations, the notion of alignment is implemented. The triples

(*source_string*, *target_tree*, *word_alignment*)

are depicted in special structures named *alignments graphs* from which the set of derivation rules are inferred. Many translation models pay little attention to the context and to the syntactic structures of the translated phrases. The theory of alignments, spans and crossings is discussed in [11] where the phrasal coherence across two languages is studied. It is proved that incorporating syntactic information into translation models presents several advantages by the fact that syntactic phrases in one language tend to say together (i.e. cohere) during translation. Also, several studies have reported alignment or translation performance for syntactically augmented translation models.

The mechanism described in this article was designed in order to test the feasibility of the automatic cross-lingual transfer of syntactic phrases being built upon an English-Romanian parallel corpus developed at *Alexandru Ioan Cuza University of Iaşi* by the Natural Language Processing Group from Faculty of Computer Science. The corpus is XML encoded obeying a simplified form of the XCES standard [16]. For the bilingual corpus construction, the English and Romanian parts of the *Acquis-Communitaire* corpus[2] were used.

All the words of this English-Romanian corpus are annotated with lemmas, morphosyntactic information (gender, number, person and case) and Part of Speech markers. The tagsets used to annotate the words of the English-Romanian corpus comes from MULTEXT-East morphosyntactic specifications, version 3 (these specifications can be found at [25]). The latest version of these specifications, version 4 called “MondiLex”, is available at [26].

The MULTEXT-East project, developed for a large number of mainly Central and Eastern European languages (including Romanian) defines tagsets not only for Part of Speech data (POS data), but also includes the EAGLES-based morphosyntactic specifications, defining the features that describe word-level syntactic annotations [9].

The proposed algorithm works only on parallel sentences that are in 1:1 correspondence, meaning that every English sentence is translated into a single Romanian sentence. Best results are obtained for parallel sentences that are as closed as possible with respect to the syntactic realization of their content.

3. The Treebank Generation Algorithm

In this section we describe the Treebank Generation algorithm used to construct the parallel treebank with syntactic constituents from an English-Romanian corpus word-aligned and annotated at the morphological and

[2] Acquis Communitaire corpus contains about 12,000 Romanian documents and 6,256 parallel English-Romanian documents [6].

syntactic level. The resulted treebank is intended to set up a MT rule-based transfer system. Thus, instead of manually designing the rules, we could derive them from the generated treebank structures.
Because of the intended purpose, the algorithm works in the following scenario:

- one language of the bilingual corpus, the source language for the MT system must have a well-known syntactic parser by means of which the parse trees corresponding to this language texts could be obtained
- the part of the bilingual corpus corresponding to the target language of the MT system must have POS annotations or there must be available a POS tagger for the target language
- the bilingual corpus upon which the treebank is constructed must be word-aligned.

Following these requirements, the English sentences of the corpus were processed with Stanford Parser [20] in order to generate the English part of the treebank.

Stanford Parser is a natural language parser developed by Dan Klein and Christopher D. Manning from the Stanford Natural Language Processing Group. By parsing the English sentences with this tool, PENN Treebank parse trees were generated. As a direct consequence, the English texts are annotated with PENN Phrasal tags as this is the tagging standard used by Stanford Parser.

The parse trees labeled with PENN tagsets [30] consist of words in leaves, POS tags for the preterminal nodes and phrase tags for the next levels. The inner nodes denote grammatical constituents (for example NP for *noun phrase*, VP for *verb phrase*, etc.). Abstraction of words in syntactic trees represents almost no informational loss from syntactic point of view.

The implemented method can be summarized as follows: given a parse tree T_s for a source language sentence noted with *s*, and its target sentence, noted with *t* (that is, the translation of the source sentence in the target language) together with the word-level alignments, the parse tree of the sentence *t*, noted with T_t has to be constructed.

The algorithm generates the target tree T_t in a bottom-up fashion by mapping constituents of T_s onto contiguous substrings of *t* via lexical alignments.

For a lexical alignment, the most frequent alignment category is 1:1 such that one word in the source text is translated exactly to one word in the target text. However, there are other alignment categories, such as *omissions* (0:1 or 1:0), *expansions* (n:m, with $n<m$, $n,m\geq 1$), *contractions* (n:m, with $n>m$, $n,m\geq 1$) or unions (n:n, with $n>1$) [3].

A very popular way for visualization the parse tree of a source language sentence ending in leaf nodes – the sentence words connected by alignment links to the target sentence's words, is the *alignment graph*. An alignment link is a function $A(n) \rightarrow \{1,\ldots,m\}^*$ that maps a source leaf node *n* of the parse tree T_s to a set of zero or more of target leaf nodes.

Based on the lexical alignments and of the source tree structure decoded in the alignment graph, the corresponding target tree structure is generated. As it is described in Algorithm 1, the tree structure of T_t created from the source parse tree T_s following the word-alignments, noted with *WA*, by means of a bottom-up mechanism.

Indeed, the algorithm starts by constructing the set of the leaf nodes, $leaf(T_t)$ together with the set corresponding to the first level of non-terminal nodes, $nonT(T_t, 1)$, where the POS tags corresponding to the leaf node are included.

For the next levels, each non-terminal node $n_t \in nonT(T_t, level)$, $level \geq 2$, is considered to be the root of a subtree $tree \in T_t$ and labeled with the phrase tag of a non-terminal node n_s from T_s if the span of n_s is the frontier of the target subtree *tree*. The Treebank Generation algorithm is given in the next section.

3.1. The Algorithm

The alignment of syntactic trees is the process of finding the correspondences between internal and leaf nodes of two parsing trees representing parallel sentences in different languages. For example, Prime Factorization and Alignment (PFA) algorithm assigns prime numbers to terminal nodes and spreads them to the rest of the tree from the leaf nodes towards to the roots by assigning the product of child values to their fathers [1].

For two parallel syntactic trees: T_s corresponding to a source language sentence and T_t for the target language translation, a non-terminal node $n_s \in nonT(T_s, level_s)$ is aligned with a non-terminal node $n_t \in nonT(T_t, level_t)$, $level_s$, $level_t > 1$, if:

$$\text{span}(t_s) = \text{leaf}(tree)$$

where *tree* is a subtree of T_t, $tree \in T_t$ and $n_t = root(tree)$.

Because the target parse tree T_t is constructed taking into consideration the structure and the nodes of T_s the Treebank Generation algorithm also includes the alignments or correspondences between internal and leaf nodes of the two parallel trees, so it could be also considered as an alignment algorithm.

Algorithm 1. The Treebank Generation algorithm

```
Input:
a bilingual source language-target language corpus3,
the word alignments WA, a source language parser4 and POS
annotations for words in the target language
```

[3] An English-Romanian morphosyntactic annotated corpus was used.
[4] Stanford Natural Language Processing Group, Stanford Parser, http://nlp.stanford.edu:8080/parser/

Output: syntactic trees for target sentences

```
1. APPLY WA on the corpus C to obtain the word alignments
2. FOR each pair of parallel SL-TL sentences (s, t) of C
3.   FOR each word si of s =(s1, ..., sm)
4.     FOR each word tj of t = (t1, ..., tn)
5.       IF si IS ALIGNED WITH tj
6.         aligns(i,j) ← 1
7.       ENDIF
8.   APPLY STANFORD PARSER for sentence s
9.   LET Ts the parse tree of s
10.   LET leaf(Ts) the leaf nodes set of Ts
11.       leaf(Ts) ← {si| 1≤i≤m}
12.       LET nonT(Ts,lvl)←non-terminal nodes set in Ts,lvl≥2
13.           nonT(Ts, 1) ←{POS(s)| s ∈ leaf(Ts)}
14.   nonT(Ts, lvl) ←{parent(n)|n∈ nonT(Ts,lvl-1), lvl≥2}
15.   LET Tt the parse tree of t
16.   LET leaf(Tt) the leaf nodes set of Tt
17.       leaf(Tt) ← {tj| 1≤j≤n}
18.   LET nonT(Tt, lvl) ← non-terminal nodes set in Tt,lvl≥2
19.   nonT(Tt, 1) ← {POS(n) | n ∈ leaf(Tt)}
20.   nonT(Tt, lvl) ← ∅, lvl≥2
21.   FOR each node si IN leaf(Ts)
22.       span(si) ← {tj| aligns(i,j)=1}
23.   FOR each node N in nonT(Ts, lvl), lvl≥2
24.       span(N)←{span(si)|si∈leaf(Ts) ∧ si∈ descendant(N)}
25.   FOR each node tj IN leaf(Tt)
26.       parent(tj)←POS(tj)
27. lvl ← 2
28. WHILE (lvl ≤ max_level(Ts))
29.   nonT(Tt,lvl)← ∅
30.   FOR each node tj IN leaf(Tt)
31.     IF ∃ N∈ nonT(Ts, lvl): tj∈ span(N)
32.         parent(last_parent(tj))← N
33.         nonT(Tt,lvl)← nonT(Tt,lvl)∪{N}
34.     ENDIF
35.  lvl← lvl+1
36. ENDWHILE
```

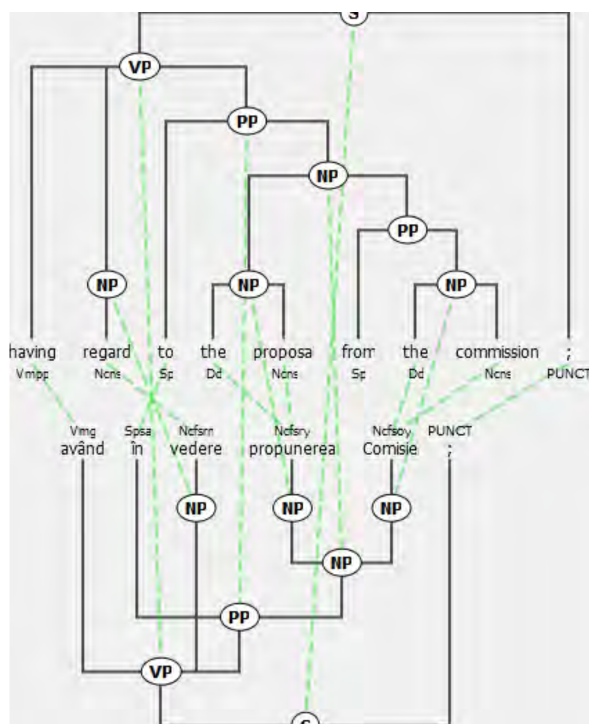


Figure 1. Cross word-alignments generate overlapping phrase-structures[5]

In any translation process, the one-to-zero lexical alignment is undesirable. One-to-zero means the lack of equivalent lexical translation in the target language, this phenomena being called “lexical hole”. The unaligned Romanian words can be resolved using specific grammatical information relative to the Romanian language but such a study does not make the subject of this article.

The time complexity of our algorithm relative to each pair of parallel Source Language-Target Language sentences (*s,t*) is $O(n^{max_level(Ts)})$ because of the *while* loop that includes a linear browsing of the leaf nodes from *Tt*, where *n* is the length of the sentence in the Target Language (this means that sentence *t* has *n* words, property that is given by $t = (t_1, \ldots, t_n)$ in the algorithm notations).

Algorithm 1 assumes that all the spans of the non-terminal nodes of *Ts* are continuous lists of nodes and does not resolve the crossing alignments between each pair of English-Romanian parallel sentences. These issues will be discussed in the next section.

---

[5] The treebank alignments are loaded in Stockholm TreeAligner program [36].

3.2. The Alignments of the Treebank Parallel Components

The most important feature of the developed algorithm consists in finding the translation equivalence between two syntactic phrases of each bitext. Basically, translation equivalence rely only on the lexical tokens (words, phrases) paired by an alignment link. Even if not all the words between the two phrases are aligned, the phrases can still align very well.

The word alignments were drawn manually between the parallel sentences of the English-Romanian corpus. Although the syntactic structures in the two languages are not similar, some alignments can still be identified in order to support the syntactic equivalences. For all that, aligning two words with the same meaning but with different part of speech is not desirable from this kind of study point of view because, in this case, even if the alignment is semantically correct it can't help the phrases equivalence.

Crossing alignments

In any translation process, lexical mapping is inevitable. Crossing lexical alignments between a source sentence and a target sentence happen when the order between the source words and their translations is not preserved.

In Figure 1 it is shown one crossing among the word-alignments links, indicating one instance of reversing syntactic constituents during translation process. This particular crossing involves reversal of the prepositional word with the noun word. Depending on how often this reversal is encountered, in a translation process we could consider to invert all the TO constituents that appear before NP constituents.

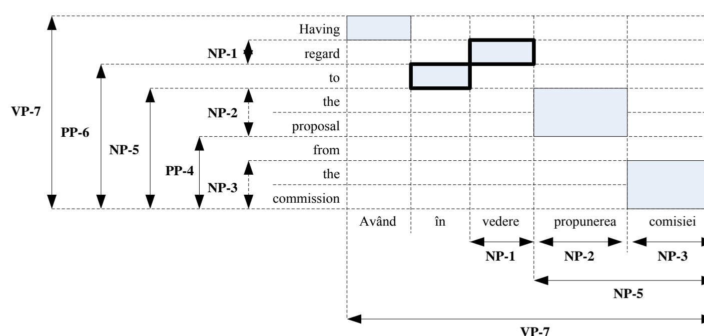


Figure 2. The alignment matrix[6]

[6] The crossing alignments are marked with a thick border.

The crossing alignments can be easily identified using the `aligns` matrix constructed in Algorithm 1.

The alignment matrix that corresponds to a pair of English-Romanian parallel sentences (*s*, *t*) from the *JRC-Acquis* corpus, for:

s = (*Having*, *regard*, *to*, *the*, *proposal*, *from*, *the*, *commission*)

t = (*Având*, *în*, *vedere*, *propunerea*, *comisiei*)

is illustrated in Figure 2.

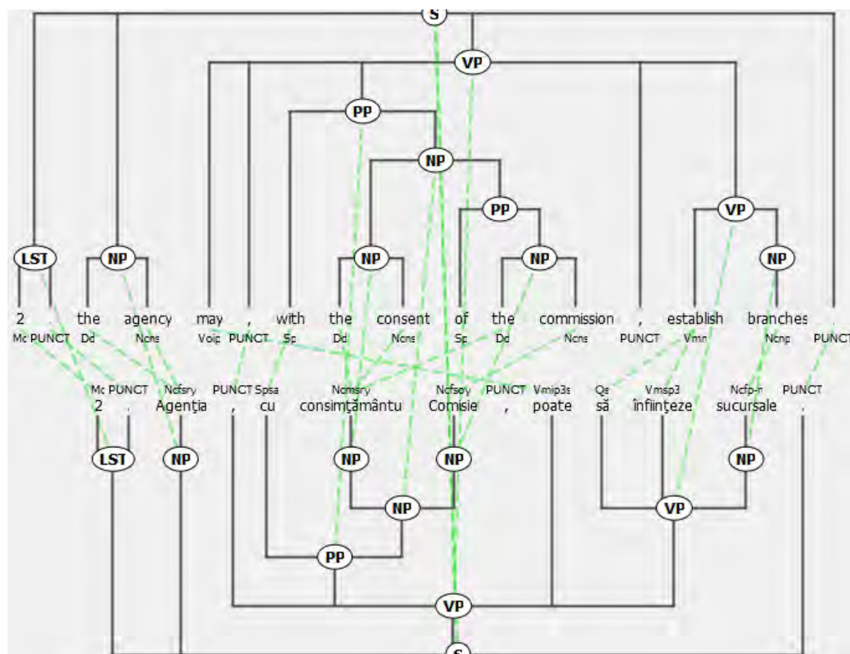


Figure 3. Cross word-alignments do not necessarily generate overlapping phrase-structures[7]

Usually, the crossing alignment problem implies reordering of the source tree such that the lexical order of the leaf nodes matches the order of the target sentence. But resolving this issue implies particular studies that address the particularities of the target language with respect to the particularities of the source language. Such studies do not make the subject of the present article, being left for a future work.

Still, not all crossing word-alignments determine overlapping between the target tree syntactic components as it is exemplified in Figure 3.

[7] The treebank alignments are loaded in Stockholm TreeAligner program [27].

3.3. Parallel Treebank Annotations

After the parse trees were generated for both parts of the corpus, the hierarchical language representations for the parse trees have to be flattened into linear string representations, which can be easily input to many feature-like probabilistic models. Thus, during model-training, these string representations together with the alignment information can generate statistics needed to build translation grammars. Our goal is to extract rich and robust set of English-Romanian translation rules.

In line with the PENN parse tree format used by the Stanford Parser we propose a format in which the aligned phrase tags for the inner nodes of the trees are indexed by the same number. The common notation for the phrase nodes accompanied by the lexical alignments for the leaf nodes make easier to find the alignments between the parallel parse trees.

The annotations of the treebank preserve, from the used English-Romanian corpus, the MULTEXT-EAST words specifications as these data include all the morpho-syntactic details needed for any syntactic study, while for the phrasal constituents the PENN Treebank Phrasal Tags are used.

In order to evaluate the Phrasal tags for Romanian sentences resulted from the Treebank Generation algorithm, the corpus annotations with syntactic chunks for the Romanian words are compared with the Phrasal tags sequences “inherited” from the parallel English sentences parse trees.

The chunks annotations of the corpus were generated by means of a simple regular expression chunker in order to mark the syntactic constituents that form a given sentence. More precisely, two separate English and Romanian grammars were implemented for generating PERL regular expressions over sequences of POS tags for English and Romanian types of phrases founded in the corpus sentences [24]. Using the languages regular expressions defined over the tagsets, the chunker accurately recognizes the (non-recursive) syntactic phrases both for Romanian and English.

The chunk parser detects the chunks of a text, like *noun phrases* (*NPs*), *prepositional phrases* (*PPs*) or *verb phrases* (*VPs*). Chunks are non-overlapping spans of text, usually consisting of a head word (such as noun) and the adjacent modifiers and function words [6]. The chunk annotations are the references in the evaluation process based on which the performance of the Treebank Generation algorithm is measured.

4. Experimental Results and Evaluation

The Treebank Generation algorithm for the Romanian sentences was tested by taking into account the chunker annotations for the Romanian part of the previously mentioned bilingual corpus. More precisely, for every word of a Romanian sentence, each syntactic phrase determined by the Treebank Generation mechanism that correctly matches within the syntactic chunks annotations of that word adds to the mechanism precision.

Resuming, we have that the PENN phrase tags identified for the Romanian words by the Treebank Generation algorithm are compared with the sequences of syntactic chunks specified for the Romanian words of the English-Romanian corpus.

Table 1. The corpus annotations and the corresponding PENN Phrasal Tags

Corpus annotations	Ap adjective phrase/ adverb phrase	Np noun phrase	Pp prepositional phrase	Vp verb phrase
Corresponding PENN Phrasal Tags	ADJP ADVP PP WHADVP	NP WHNP	PP WHPP	VP

As it is given in Table 1, the *NP* and *WHNP* PENN Phrasal tags are considered the equivalent PENN notations for the *Np* chunk annotations, a *VP* tag matches only with a *Vp* chunk while the *PP* and *WHPP* tags match only with *Pp* chunks. In the case of the *Ap* chunk a discrimination algorithm had to be implemented in order to correctly evaluate this notation according with its corresponding meaning.

Table 2. Example of parallel sequences of treebank tags and chunker annotations together with their matching degrees

Token(word)	Treebank tags/ chunker annotations	Number of matches
vot	Ncms-n VP VP NP VP VP S Np Pp	no match[8]
de_asemenea	Rgp **ADVP** VP S ROOT **Ap**	one match
economic	Afpms-n **ADJP** **NP** NP VP … **Ap** **Np** Pp	two matches
dividende	Ncfp-n **NP** **PP** VP S ROOT **Np** **Pp**	two matches
în	Spsa **PP** **VP** **PP** S **Ap** **Vp** **Pp**	three matches

Indeed, because a single *Ap* notation is used by the chunker for both the *adjectival phrase* and the *adverbial phrase*, the evaluation mechanism has to

[8] The Romanian noun “*vot*” inherits different PENN tags from the alignment mechanism because it was aligned with a word with different Part of Speech, more precisely it was aligned with a verb.

discriminate among the cases when the *Ap* means *ADJP*, that is adjectival phrasal tag or *ADVP*, the adverbial phrasal tag or *WHADVP*, wh-adevarb phrase or even *PP* prepositional phrase tag. This discrimination is done upon the part of speech of the annotated token/word.

More precisely, if a Romanian word, annotated with the *Ap* chunk, is:

- an adjective, then *Ap* is considered the correspondent of the *ADJP* tag in the PENN Phrasal format
- an adverb then its *Ap* annotation will match only with *ADVP* or *WHADVP* PENN tags
- a preposition then the *Ap* annotation is considered equivalent with the *PP* PENN tag.

The number of matches between the tags of a PENN phrasal sequence of a Romanian word and the chunks of the corpus annotations for that word is counted for the transfer precision. In Table 2 we exemplify the manner in which the transfer precision is determined. Because we evaluate the knowledge transfer degree, it is obviously that only the sequences of PENN phrasal tags that correspond to Romanian words with non-null alignments in the English parallel part of the corpus will be considered.

The performance of the Treebank Generation algorithm is measured in terms of *Precision* and *Recall*, such that:

- *Precision* is the fraction of correctly identified Phrasal tags with respect to the total number of generated Phrasal tags
- *Recall* is the fraction of correctly identified Phrasal tags with respect to the total number of Phrasal tags specified in the chunker annotation sequences for the words of the corpus

The resulted scores of the evaluation process are given in Table 3. Analyzing the numbers of Table 3, one can observe that the scores for Precision and Recall do not critically depend on the size of the data sets (200 sentences of the first data set vs. 1420 sentences for the second data set).

Table 3. Data sets and the resulted precision and recall numbers

Corpus size	Number of tokens (words)	Precision	Recall
200 sentences	3433	0.8691	0.8411
1420 sentences	22345	0.8542	0.8225

5. Conclusions

The proposed mechanism provides a way to generate syntactic representations for a language without many parsing tools (like Romanian) by reusing tools of an intense studied language (English) to which word-alignments could be provided. It is well-known that the lexical alignments influence greatly the alignment of internal nodes in two parallel syntactic trees.

From our description it can be easily deduced that wrong or incomplete alignments can affect greatly the knowledge transfer. Also, the quality of the source representations has a great impact on the target induced representations.

Nevertheless, the method has to be improved in order to deal with specific constructions for the target language, which do not have any correspondence in the source language. As the authors say in [24], in order to achieve better results, the target language specific syntactic structures require a pre- and post- processing of the data.

It is important to say that although the treebank generation mechanism presented in this paper was carried out on a specific language pair, that is English and Romanian, it is so far language independent.

Acknowledgements. The author M. Colhon has been funded for this research by the strategic grant POSDRU/89/1.5/S/61968, Project ID 61986 (2009), co-financed by the European Social Fund within the Sectorial Operational Program Human Resources Development 2007-2013.

Also, the author would like to thank the Natural Language Group of Faculty of Computer Science, *Al. I. Cuza University of Iaşi*, Romania, for providing the English-Romanian corpus upon which the presented treebank generation mechanism was developed and also evaluated.

References

1. J.G. Araújo and H.M. Caseli, “Alignment of Portuguese-English syntactic trees using part-of-speech filters”. In Workshop on Natural Language Processing and web-based technologies (IBERAMIA-2010), Bahía Blanca,1-10. (2010)
2. D. Bamman, M. Passarotti, G. Crane, and S. Raynaud, “Guidelines for the Syntactic Annotation of Latin Treebanks” (v. 1.3), Technical report, Tufts Digital Library, Medford, 2007, http://nlp.perseus.tufts.edu/syntax/ treebank/1.3/ docs/guidelines.pdf
3. H. de Medeiros Caseli, A. M. de Paz Silva and M. das Graças Volpe Nunes, “Evaluation of Methods for Sentence and Lexical Alignment of Brazilian, Portuguese and English Parallel Texts”. In Brazilian Symposium on Artificial Intelligence - SBIA, 184-193. (2004)
4. A. Ceauşu, “Rich morpho-syntactic description for factored machine translation with highly inflected languages as target”. In Workshop on Machine Translation and Morphologically-rich langauges, University of Haifa. (2011)
5. A. Ceauşu, D. Tufiş, “Addressing SMT Data Sparseness when Translating into Morphologically-Rich Languages”, NLPSC 2011, Special Issue Human-Machine Interaction in Translation, August 2011, Copenhagen, Denmark. (2011)
6. D. Cristea, C. Forăscu, “Linguistic Resources and Technologies for Romanian Language”, Computer Science Journal of Moldova, vol. 14, no. 1(40). (2006)
7. J. Cuřín, M. Čmejrek, J. Havelka, V. Kuboň, “Building a Parallel Bilingual Syntactically Annotated Corpus”, in K.-Y. Su et al. (Eds.): IJCNLP 2004, LNAI 3248, 168–176. (2005)
8. R. Edqvist, “Developing a Core Lexicon for a Corpus-based Machine Translation System”, Master’s thesis in Computational Linguistics, Uppsala University, Department of Linguistics and Philology. (2005)

9. T. Erjavec, “MULTEXT-East Version 4: Multilingual Morphosyntactic Specifications, Lexicons and Corpora”, Lexicons and Corpora. (2010)
10. T. Erjavec, B. Sárossy, “Morphosyntactic Tagging of Slovene Legal Language”, Informatica(30), 483-488. (2006)
11. H. J. Fox. “Phrasal cohesion and statistical machine translation”. In Proceedings of. EMNLP-02, 304–311. (2002)
12. M. Galley, M. Hopkins, K. Knight and D. Marcu, “What’s in a translation rule?,” In Proceedings of HLT-NAACL 2004, Publisher: Association for Computational Linguistics, Boston, USA, 273-280. (2004)
13. M. Galley, J. Graehl, K. Knight, D. Marcu, S. DeNeefe, W. Wang, I. Thayer, “Scalable Inference and Training of Context-Rich Syntactic Translation Models”. In: ACL, 961–968. (2006)
14. A. de Gispert, J. Pino, W. Byrne, “Hierarchical Phrase-based Translation Grammars Extracted from Alignment Posterior Probabilities”. In Proceedings of EMNLP'2010, 545-554. (2010)
15. A. Göhring, “Spanish Expansion of a Parallel Treebank”, Ph.D Thesis, University of Zürich, Switzerland. (2009)
16. N. Ide, P. Bonhomme and L. Romary, “XCES: An xml-based encoding standard for linguistic corpora”. In Proceeding of the Second International Language Resources and Evaluation Conference, Paris: European Language Resources Association. (2000)
17. E. Irimia, “EBMT Experiments for the English-Romanian Language Pair”. In Recent Advances in Intelligent Information Systems, ISBN 978-83-60434-59-8, 91-102
18. JRC-Acquis, Available: http://langtech.jrc.it/JRC-Acquis.html
19. E. F. Kelly, Philip J. Stone, “Computer Recognition of English Word Senses”, North-Holland, Amsterdam. (1975)
20. D. Klein, C. D. Manning, “Accurate Unlexicalized Parsing”, In: Proceedings of the 41st Meeting of the Association for Computational Linguistics, pp. 423-430. (2003)
21. Y. Liu, Y. Huang, Q. Liu, S. Lin, “Forest-to-string statistical translation rules”, in: ACL, 704–711. (2007)
22. M. P. Marcus, B. Santorini and M. A. Marcinkiewicz, “Building a Large Annotated Corpus of English: The Penn Treebank”. In COMPUTATIONAL LINGUISTICS, vol. 19(2), 313-330. (1993)
23. Melamed, I.D.: Manual Annotation of Translational Equivalence: The Blinker Project. In IRCS Technical Reports Series, University of Pennsylvania. (1998)
24. V.B. Mititelu and R. Ion, “Automatic Import of Verbal Syntactic Relations Using Parallel Corpora”. In Proceedings of Recent Advances in Natural Language Processing, Borovets, Bulgaria. (2005)
25. MULTEXT-East version 3 specifications, http://nl.ijs.si/ME/V3/msd
26. MULTEXT-East version 4 specifications, http://nl.ijs.si/ME/V4/
27. S. De Neefe and K. Knight, “Synchronous Tree Adjoining Machine Translation”. In Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing EMNLP 2009 vol. 2. (2009)
28. NEGRA Treebank, http://www.coli.uni-sb.de/sfb378/negra-corpus/
29. F.J. Och, and H. Ney, A Systematic Comparison of Various Statistical Alignment Models. Computational Linguistics 29, 19-51. (2003)
30. Penn Treebank, http://www.cis.upenn.edu/~treebank/
31. Prague Dependency Treebank, http://ufal.mff.cuni.cz/pdt2.0/
32. Swedish Treebank: http://stp.ling.uu.se/~nivre/swedish_treebank
33. J. Tinsley, M. Hearne and A. Way, “Exploiting Parallel Treebanks to Improve Phrase-Based Statistical Machine Translation”. In K. De Smedt, J. Hajič and S.

Kübler (Eds.), Proceedings of the Sixth International Workshop on Treebanks and Linguistic Theories. (2007)
34. D. Tufiş, R. Ion, “Parallel Corpora, Alignment Technologies and Further Prospects in Multilingual Resources and Technology Infrastructure”, in C. Burileanu and H-N Teodorescu (Eds.), Proceedings of the 4th Conference on Speech Technology and Human-Computer Dialogue, SpeD 2007, Iaşi, Romania. (2007)
35. D. Tufiş, S. Koeva, T. Erjavec, M. Gavrilidou, and C. Krstev, “Building Language Resources and Translation Models for Machine Translation focused on South Slavic and Balkan Languages”, in M. Tadić, M. Dimitrova-Vulchanova and S. Koeva (eds.) Proceedings of the Sixth International Conference Formal Approaches to South Slavic and Balkan Languages (FASSBL 2008), 145-152, Dubrovnik, Croatia, ISBN 978-953-55375-0-2. (2008)
36. M. Volk, J. Lundborg, and M. Mettler, “Alignment tools for parallel treebanks”, in Proc. of The Linguistic Annotation Workshop at the Association for Computational Linguistics (LAW-ACL). (2007)
37. J. Vičič, and A. Broddnik, “Parse Tree Based Machine Translation for Less-used Languages”, Available: http://mrvar.fdv.uni-lj.si/pub/mz/m25.1/abst/vicic.htm
38. G. Maillette de Buy Wenniger, M. Khalilov and K. Sima’an, “A Toolkit for Visualizing the Coherence of Tree-based Reordering with Word-Alignments”, in The Prague Bulletin of Mathematical Linguistics no. 94, 97–106. doi: 10.2478/v10108-010-0024-4. (2010)
39. K. Yamada, K. Knight, “A syntax-based statistical translation model”. In Proceedings of the 39th Meeting of the Association for Computational Linguistics ACL 2001, 523-530, Toulouse, France. (2001)
40. D. Zhang, M. Li, C.-h. Li, M. Zhou, “Phrase reordering model integrating syntactic knowledge for SMT”, in: EMNLP/CoNLL, 533–540. (2007)

Mihaela Colhon (born Mihaela Ghindeanu): Since 2005: Assistant professor at Departament of Computer Science, University of Craiova, Romania; Since 2009: PhD in Computer Science, Department of Computer Science, Faculty of Mathematics and Computer Science, University of Pitesti, Romania; Competence domains: Logic programming (Prolog and Lisp programming languages), Knowledge Representation, Knowledge Bases, Expert Systems, Natural Language Processing (syntactic analysis); Teaches: Algorithms and Data structures at Departament of Computer Science, University of Craiova and Artificial Intelligence at Departament of Computer Science, University of Craiova, Romania and Departament of Computer Science, University of Bucharest, Romania; Representative articles: 1) Florentina Hristea, Mihaela Colhon, Feeding Syntactic Versus Semantic Knowledge to a Knowledge-lean Unsupervised Word Sense Disambiguation Algorithm with an Underlying Naive Bayes Model, FUND INFORM (2012), 2) Ion Iancu, Nicolae Constantinescu, Mihaela Colhon: Fingerprints Identification using a Fuzzy Logic System, INT J COMPUT COMMUN (2010), 3) Nicolae Tandareanu, Mihaela Ghindeanu, Sergiu Nicolescu: Hierarchical Distributed Reasoning System for Geometric Image Generation , INT J COMPUT COMMUN (2009).

Received: January 30, 2012; Accepted: May 28, 2012.

DOI: 10.2298/CSIS120108028P

Implementing an eXAT-based distributed monitoring system prototype

Gleb Peregud[1], Julian Zubek[1], Maria Ganzha[2,3], and Marcin Paprzycki[3,4]

[1] Warsaw University of Technology, Warsaw, Poland
[2] University of Gdansk, Gdańsk, Poland
[3] Systems Research Institute Polish Academy of Sciences
Warsaw, Poland
<firstname>.<lastname>@ibspan.waw.pl
[4] Warsaw Management Academy, Warsaw, Poland

Abstract. Monitoring resource utilization in distributed systems remains of importance. This is especially the case in LAN-based distributed systems (and, in particular, in global Grid systems), where individual nodes can be (may need to be) added to and/or removed from the system at "random" moments. The aim of this paper is to report initial results of the project that aims at using Erlang-based software agents as a robust and flexible resource monitoring infrastructure. The implemented prototype is capable not only of collecting performance data, but can also detect certain network problems. Furthermore, an assessment of the eXAT agent platform, based on experiences gathered during prototype implementation, is included.

Keywords: Grid computing, resource monitoring, Erlang, eXAT, intelligent agents

1. Introduction

In computing, *Grid* is a term that typically refers to a group of loosely coupled computers, working in a dynamically created arrangement to reach a common goal [37]. Grid technology is used both to solve computationally intensive scientific problems, and/or to deliver needed resources (e.g. computing cycles, software services, or data) in commercial applications. Such systems, by the definition, are heterogeneous and geographically dispersed. Here, two types of Grid systems can be distinguished. First, a *Global Grid*, somewhat similar to volunteer computing systems (e.g. the BOINC infrastructure [6]). Second, a *Local/Desktop Grid*, which can be characterized, among others, by existence of designated administrators (for the whole Grid installation, or for each of its parts). Unfortunately, Grid systems (as well as other LAN-based distributed systems) are prone to problems originating, for instance, from the network infrastructure, configuration, etc. Furthermore, one of the common problems concerning use of Grid infrastructures is load balancing. Hence, the need for software tools, which can help system administrators to monitor the state of the Grid (be it local or global) and efficiently manage its resources.

One of the interesting ideas, put forward by leading specialists in the fields of *Grid* and *agent* computing was to combine the strength of both approaches to deliver the computing fabric of the future (see [36], for more details). In other words, the idea was to use intelligence of software agents to provide the "brain" for the computational "muscle" of the Grid infrastructure. While there exists projects like the *Agents in Grid* [46,26,45,27,44], which attempt at directly realizing this vision, here, we focus our attention on application of software agents as "intelligent monitors" within the Grid (as well as in other LAN-based distributed systems, including Cloud infrastructures). In this context, note that a number of cases of agent-based monitoring systems have been described in the literature for other application areas. For instance, agents where used to monitor network traffic [53], an experimental environment in a laboratory [52], as well as power systems [51].

The aim of our project was two-fold. First, to develop foundations for a robust, fault tolerant, extensible, agent-based Grid / LAN / Cloud monitoring system, capable of working without need for manual configuration. Furthermore, the proposed system was to be capable of inferring knowledge from gathered data and acting upon it. Note that, while we focus on the Grid as the main use case, *all* results presented here are immediately applicable to the infrastructures within the Cloud environments, as well as to standard LAN infrastructures. Therefore, in what follows, the term Cloud (or LAN) could have been used in place of Grid (with proper caution applied, and with reflection on consequences of such interchange). Second, to assess robustness and flexibility of the eXAT agent framework [63,61,58] applied to the task at hand. Here, the potential advantages of an Erlang-based, FIPA compliant, agent framework are to be judged against their actual realization in the eXAT framework.

2. Related work

The proposed system is designed to support Grid / Cloud / LAN administrators in their routine activities. The two main use cases considered in our work are: (1) detection of "connectivity problems" (e.g. disappearance of a node or a link), and (2) monitoring (and reporting) performance metrics of individual nodes (or their groups). The latter use case can provide foundation for autonomous load re-balancing.

Task of resource monitoring has been solved by the monitoring software like Nagios [39] or Ganglia [49]. Both projects are quite mature, ready to use in complex, real-world situations. They were written without employing agent model, using traditional programming paradigms.

Nagios is an all-in-one system, which is able to monitor every key part of an IT infrastructure: system metrics, network protocols, applications, services, servers, etc. It is a general tool, which can be applied to monitoring Grid infrastructures as well. Within the Nagios there is a lot of space for customization through custom plugins. However, use of the Nagios system requires extensive manual configuration, which may be inconvenient in a geographically distributed

large-scale Grid infrastructures. Furthermore, the fact that ownership of various fragments of the (global) Grid belongs to different entities, makes any "global configuration" task much more difficult.

Ganglia is a more specialized software for clusters and Grids (possibly consisting of smaller clusters). It is designed to achieve low per-node overhead, focuses on gathering performance metric of each machine, and on generating statistics for the whole cluster. Ganglia requires little configuration to start working and, like Nagios, supports custom plugins. However, it is not easy to extend its functionality beyond the assumed one (e.g. add inferencing knowledge based on collected data, and acting on it).

Furthermore, both these systems depend on the existence of a central server (or a group of servers), gathering information from remote processes working on every node. Note that, while in the monitoring system's nomenclature, those remote processes are often called agents, they are just clients for a central server, and they lack typical properties of agents (for a classical definition of software agents and agent systems, see [40]). Design with a centralized server leads to potential problems. When the application server (or the machine on which it is running) fails, data will no longer be gathered. Similar situation occurs when, due to a network failure, some connections are broken. Another drawback of these approaches, is a need to reconfigure servers, in the case of adding new nodes to the Grid (or node removal).

Finally, let us recall that we would like to develop a system, which is able not only to provide information, which can be inferred from metrics gathered from the LAN, but also to act on it. While it would be possible to develop such system on top of either Nagios or Ganglia, it would immediately involve problems describe above. Furthermore, it would add another layer of software into already crowded Grid system software stack. Therefore, we have decided to build a system that will use capabilities of software agents, avoid the above mentioned problems of Nagios and Ganglia, and be capable of providing the additional needed functionality.

In the multi agent systems world, most of the projects connected with grid computing focus on goals similar to the *Agents in Grid* project. Monitoring of the physical network infrastructure is usually out of their scope. Nevertheless, some of them use approach similar to the proposed one.

AgentScape [23] is a distributed middleware that supports large-scale agent systems. It has features of an agent platform, as well as those of a distributed agent operating system. Among its features it provides decentralized resource discovery. However, since the focus of the AgentScape system is to develop agent middleware for large scale distributed systems, this project is much broader in scope. Furthermore, the last update of the AgentScape software is from April 2011 and it is unclear what is the progress of development of the AgentScape 2.

Similar, but less advanced, project was MAGDA: Mobile Agent Based Grid Architechture [20]. It was build on top of JADE agent platform and facilitated creating Grid applications based on mobile agents. Monitoring of agents and sys-

tem resources as well as service discovery and load balancing were planned. It provided support for collective communication and use spanning trees for effective broadcast over the Grid. The same approach was later used in our project. Note that, the last published reference to the MAGDA system is from 2006 and thus we have to assume that it is no longer pursued.

An interesting approach to service discovery is represented by the ARMS [25] project. It provides an agent-based resource management system for Grid computing. The system consists of homogeneous agents managing local resources and advertising them through the Grid. A special agent has a global view of the system and simulates other agents' performance during runtime. It uses the PACE performance prediction tool-kit and, based on computed metrics, optimizes the behaviour of other agents. However, according to J. Cao, the development of both the PACE tool-kit and the ARMS project stopped in 2002.

3. Proposed approach—overview

Taking into account the above considerations, let us outline the main tenets of our proposed approach. First, we use software agents to develop a framework for intelligent monitoring of the state of a distributed system. We envision that a single agent will be placed at each node the system. Such autonomous agent will be capable of acting both as a "client" and as a "server." As a result, it will be capable not only of monitoring the state of the node, but also of inferring knowledge about the state of the system (or its fragments) and act on this knowledge. For instance, in the case of load imbalance, it will be capable of initiating procedures leading to the load re-balancing (see, also [28]). To achieve these goals, the proposed system will be designed in such a way that information about the state of the Grid (nodes of a distributed system) will be spread among the agents. Therefore, the information will remain available (at least to some extent) even after network link (or Grid node) failure. Furthermore, adding new nodes will not require any reconfiguration, because agents will be able to discover themselves, communicate and share the load of monitoring of whole system (Grid) evenly between them. Finally, this design of the system will alleviate the potential problem of a single point of failure. Let us present now two simple use cases that we have implemented in the initial system prototype, to illustrate its features and properties.

The simplest use case is: monitoring basic metrics in a basic LAN environment, with a star topology. Due to the zero-configuration feature, deployment of the system in a standard LAN should be very easy to complete. Right after the system is deployed, its administrator should be able to access his local agent (via a web interface) and start receiving information about the state of the nodes in the LAN (e.g. in form of plots).

A more complex use case is: continuous monitoring of a LAN with topology different than the simple star. System should be able to detect problems with network links and distinguish between network link failures, node failures, and failures of monitoring agents. This functionality requires that the system

contains redundant network links, or alternative paths in the network. In most cases, no manual configuration should be necessary to make the monitoring system work.

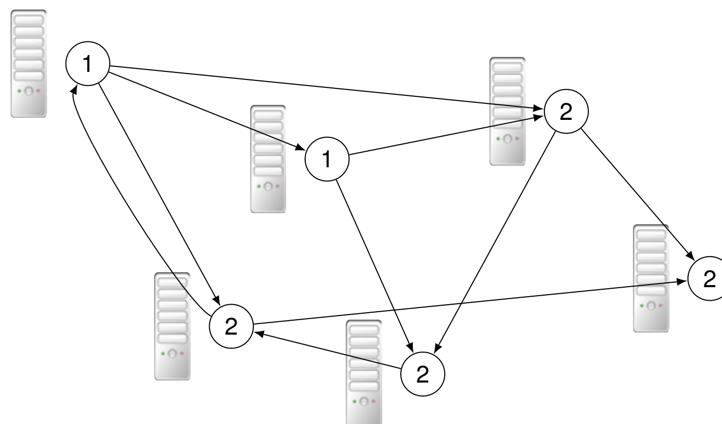


Fig. 1. Sample grid with monitoring agents. An edge from node A to B means that agent A monitors agent B. Number in circle is the number of agents monitoring agent running on the specific node.

To illustrate our approach, in Figure 1 we depict a sample network topology with agents running on every node. As we can see, there is no central node monitoring the remaining agents – the monitoring obligation is distributed, and every agent is monitored by at least one other agent. This illustrates how the robustness of monitoring process is achieved (this system will work correctly after the failure of a single node), and how the monitoring tasks are distributed evenly between agents, avoiding saturation of resources.

Recall that the proposed system is aimed primarily at supporting Grid managers (local administrators in the case of a local area network or a local Grid, as well as local and global administrators in the case of the global Grid). Therefore, as a starting point (to test the two use case scenarios), we have decided to implement the following features:

– access to the resource utilization metrics,
– automatic, zero-configuration discovery of agents in the local area network (or any other network where the multicast UDP is enabled), and
– inferring diagnostic information for basic problems, using a rule-based approach.

By the *resource utilization metrics* we understand various, easy to establish, metrics such as: CPU load, memory usage, running processes and their CPU usage, etc. Such data is going to be gathered and made available to the user. In

the case of a human user, it will be presented to her as plots showing how the value of selected parameter changes over time. In the case of an agent user, data will be presented in an appropriate, machine-understandable format. For instance, in the case of the *AiG* project, such format is going to be derived from the ontology of Grid used there (see, for instance, [29,30]. This feature remains to be implemented.

For the two use cases, we decided to initially diagnose the following problems: (a) broken connections, (b) inactive nodes, and (c) inactive agents. Obviously, these are very basic problems, but they have been selected to illustrate capabilities of our approach to monitoring (e.g. they will show how a rule based expert system can be used within an agent to diagnose the nature of the network problem). Obviously, this list can be easily extended (e.g. by adding new rules to the expert system).

In our case, the *zero-configuration* means that no special information needs to be provided (by the user) for the agent to join the system. Therefore, when system administrator plugs her laptop into the network, her agent will find all other monitoring agents within it autonomously. As soon as this is done, the administrator agent can immediately start gathering information about the status of the Grid. Note that, the administrator agent, by default, runs the same code as all other agents with one exception—it is also running a GUI interface to display the obtained information in a human-readable format (plots via a web interface; see, section 4 for more details).

3.1. Agent technologies in the system

Agents and actors today Let us start from a brief methodological reflection. Observe that, recently the *actor model* of computing has (re)gained popularity. This due to the increasing complexity of building distributed software systems using more conventional models. Multiple “older” systems modelled after the actor model became popular in the industry, while new ones have been developed recently. Among them we can mention, Scala [50,24], SALSA [67], JavAct [17] and Kilim [60] in the Java world [42]; E language [56]; Asynchronous Agents Library [1] and Axum [5] from Microsoft; Act++ [41], Thal [43], libactor [8] and Theron [11] for the C/C++ languages; Stackless Python [65] and Stage [21] for Python; and Revactor [18] for Ruby. Some industry leaders have also employed the actor model, like usage of Scala by Twitter [34], and usage of Erlang by Facebook [47].

As we can see, the actor model is being adopted by a broad rage of companies, with at least some degree of success. Therefore, let us briefly discuss the relationship between actors and agents. According to Gul Agha, actors have the following properties [14]:

- concurrent computational entities
- independent from other actors (autonomous)
- can communicate with other actors via messages
- reacts to received messages

– can create new actors

According to Stan Franklin and Art Graesser, intelligent agents have the following properties [38]:

– reactive (or sensing; or acting)
– autonomous
– goal–oriented (or pro-active purposefully)
– temporary continuous (or continuous running processes)

Additionally they may display the following properties:

– communicative (or socially–able)
– learning (or adaptive)
– mobile
– flexible (actions not limited to simple script)
– character (referring to “personality” and emotional state)

As we can see, basic properties of agents match well with properties of actors. Hence we can argue that the *agents model* is a superset of the *actor model*, with addition of intelligence, goal-orientation, adaptiveness, mobility, proactiveness, etc.

Since Erlang, as a specific implementation of the actor model, has all above properties of actors build-in, we find it a good and natural foundation to develop an agent system. This is precisely what underlined the eXAT project [63] that we will now focus our attention on.

3.2. Erlang, eXAT and ERESYE

As stated above, we have decided to develop our system using the eXAT agent framework [63]. There were multiple reasons for this choice. First, according to its author (see [58]), the eXAT provides a FIPA-compliant implementation of an agent platform that includes:

– FIPA-ACL (Agent Communication Language),
– AMS (Agent Management System),
– support for ontologies,
– messaging using the MTP protocol,
– integration with the ERESYE (ERlang Expert SYstem Engine).

Since this looked quite interesting, we have decided to assess the quality of eXAT (which is an experimental tool) in practice of agent system development. Note also that, according to our vision of agent system design and implementation, Erlang, as an actor-based concurrent language with a distributed virtual machine, is really promising for implementing agent-based systems. Furthermore, being based on Erlang, could provide eXAT with certain advantages over other agent frameworks. They include: (i) natural, functional language syntax with declarative elements, suitable for representing knowledge; (ii) efficient

agent communication and concurrency; (iii) easy access to Erlang libraries; and (iv) availability of a rule-based expert system engine build-in into eXAT.

One of often invoked characteristics of software agents is their intelligence. In our system this can be facilitated through the use of a rule-based expert system. We are referring to the ERESYE [62], which is a rule production system, written in Erlang and created by the same team that has developed the eXAT environment [62]. It is similar to other expert systems, like CLIPS [57] or Jess [13]. However, due to Erlang's declarative nature, it is possible to represent rules for the ERESYE using syntax very similar to the Erlang code itself. This simplifies the learning curve and allows use of the same structures for both communication between agents, and the reasoning subsystem. As stated in section 3, the ERESYE is to be used to infer information and provide it to the users of the system (e.g. system administrators).

Finally, note that, since the eXAT is claimed to be FIPA-compliant, and uses the FIPA-ACL message format, it should be possible to establish communication between agents written in eXAT and agents written in other FIPA-compliant systems, e.g. JADE [22]. This, in turn, should allow one to write systems, which utilize both agent platforms. For instance, it should be possible to add an eXAT-based monitoring subsystems to agent teams formed in the above-mentioned *Agents in Grid* project. Therefore, one of the auxiliary aims of our work was to establish that passing messages (bidirectionally) between eXAT and JADE is possible, without extra development efforts. Note also, that while implementing the system prototype (and the eXAT-JADE communication), as an extra result, we managed to made some observations comparing the Erlang/eXAT and the Java/JADE agent platforms, which we report in section 7.

4. Implementation details

Thus far we have implemented, a somewhat limited in scope, prototype of the above outlined system. This proof-of-concept implementation works in a LAN, which can be considered a basic variation of a Grid environment. In the near future, the system will be extended to support more complex environments. Let us now look into some details of the implemented prototype; starting from individual monitoring agents.

There are two basic functions of each agent in the system. First, monitoring other agent(s), and second, collecting local performance metrics. The third, extra function, is running a GUI, but it is executed only by those agents that are used to display data to the system administrators (see, below).

In our solution, to implement the *monitoring other agents* function, we use a method similar to that found in [45], and apply periodic pinging with FIPA-ACL *QUERY-REF* messages. Let us consider what information is needed for the agent operation. In our case this is:

- agent's own name (to provide correct reply address in sent messages),
- system metrics collected for the node,

– list of known nodes in the LAN,
– list of other agents which are monitored by given agent.

For an agent to be able to perform its monitoring tasks, it needs to gather the required information. Since the system assumes the zero-configuration approach, this has to be done automatically, using mechanisms, which are available at hand. Since agents are started independently on respective nodes, there is no pre-existing information about other agents in the system. Therefore, when choosing the name of the agent, which will be used to identify it in the system, we need to ensure its uniqueness. To achieve this goal, the name is generated automatically from the host name of the LAN (Grid) node. Here, we modified the eXAT code to use the "<nodename>.<hostname>" pattern for defining it's platform name. The *nodename* is the name of the Erlang node specified with the "-name" or the "-sname" parameter of the Erlang VM, while the *hostname* is the FQDN (Fully Qualified Domain Name) hostname of the system, as detected by the Erlang VM. The Erlang VM (and it's helper process *epmd*) ensure that there are no conflicting node names running locally. This ensures uniqueness of the "nodename" component of the agent name. The assumption that the host name of the node is correctly configured to be unique in the LAN, is sufficient guarantee of uniqueness of the agent name. Hence, since every monitoring node runs a single monitoring agent, we construct agent's name as "monitor_agent@<platform-name>". Additionally, a start script of a monitoring node retrieves list of locally registered Erlang nodes, and automatically selects a locally non-conflicting nodename.

System metrics can be acquired by using a suitable system library. Depending on the operating system used by the individual nodes, and the environment of choice, libraries like *parfait* [2] for Java, *glibtop* [3] for Linux systems coded in C/C++, or *Performance Counters API* [1] for Windows systems can be used. For the system prototype, we decided to use the *os_mon* ([33]) library, which comes with the standard Erlang distribution, and thus is a natural choice. Obviously, any of the above-mentioned libraries could be used, and interfaced with the monitoring agent. Here, the natural meta-encapsulation of the local information, which is the core of agent system development, is the guiding principle of our design. Additionally the os_mon abstracts all cross-platform details and exposes a consistent API for all platforms supported by Erlang. Therefore, let us make it explicit that the proposed monitoring system is operating system agnostic and will run also in a heterogeneous environment (consisting of computers running different OS'es) as long as all of them can run the Erlang VM.

List of all LAN nodes is discovered by using a DNSSD-based mechanism (as described in section 4.1). The remaining two lists can be built using a diffusion-based algorithm (described in section 4.3). In the near future, for more complex setups (e.g. in a distributed Grid), the agent discovery mechanism will be provided. This has to be done since the current algorithm assumes that multicast UDP is enabled, which is rarely the case in a non-LAN environment. If the monitoring system is going to be integrated into the project like the *Agents in Grid*,

it can reuse mechanisms of agents discovery used in it (e.g. the Grid middleware).

Users of the monitoring system are provided with the monitoring data, using a web-base interface. Specifically, the user GUI adds to an agent a built-in web server. In our implementation, we have selected the Misultin framework [9]. It consists of two parts—a static HTML page, and a JavaScript code, which governs all logic of the agent's web UI. The implemented GUI uses Websockets to receive information from the agent (in real-time). The Misultin provides a ready implementation of the server-side WebSocket protocol, which we take advantage of. The web server is integrated into an agent in a form of one (or more) Erlang process(es), which communicate with agent's process(es) using Erlang messaging. This allows for a clean separation between the agent's logic and the code, which is responsible for sending this information to the browser. Here, the JavaScript library Smoothie Charts [10] is used for plotting the near real-time system utilization metric data in the browser.

Two activities, based on communication with other agents, are used in monitoring a group of neighbours are: (i) periodically pinging, and (ii) broadcasting information about the state of each agent (e.g. performance metrics, node status, link status, etc.) throughout the agent-Grid. Here, by broadcasting we understand sending an information to all other agents in the LAN. To accomplish this, without over-saturating the network, we propagate messages over the edges of a spanning tree. Creation and use of the spanning tree are described in section 4.2.

Last of core activities of each monitoring agent is diagnosing problems with nodes, links and other monitoring agents (see, also, section 3). This is achieved by application of the ERESYE expert system, and described in detail in section 4.4.

4.1. Agent discovery

Let us now consider how agents can find the list of other agents existing at any given moment in the system. Perhaps the most obvious solution would be a central registry (e.g. the AMS provided service), as it is used by default by many agent platforms including the eXAT. However, the central registry would be a single point of failure (SPOF) of the system. As a result, failure of the AMS node would, for all practical purposes, lead to disintegration of the monitoring system itself. Besides, it would not go well with our policy of zero-configuration (joining agent would be forced to communicate with the AMS to start working). Thus we decided to proceed the way that P2P systems collect information about nodes in the system, and avoid the SPOF [15].

In our system every node runs its own eXAT instance acting as an autonomous agent platform and registering only local agents. Information about other, external agents is collected and stored explicitly by every agent. To pass a message to another agent, we need to know the Internet address and the port number of the destination platform, and the destination agent name.

In this way, an agent registers another agent, when it stores the following information: address, port number and agent name. Obviously, this means that the amount of locally stored information is of order of the number of nodes in the Grid, but this is a “fair price” for the zero-configuration and avoiding the SPOF. Note that, agents entering or leaving the Grid should be registered (or deregistered) by all other agents running in that Grid. Recall, that since we have adopted the naming schema described in section 4, full agent name already encapsulates the platform name and the hostname.

Since, as mentioned earlier, the initial system is designed to work in the LAN environments (e.g. private Clouds/Grids) we have chosen a well-tested approach known as the Zeroconf [12], which is based on the UDP multicast and provides a DNS-like discovery system (multicast DNS—mDNS). The two most popular implementations of the Zeroconf techniques are Bonjour [16] and Avahi [4].

Bonjour is an Apple Inc. implementation of zero-configuration networks, including address assignment, service discovery and name resolution. It implements the DNS Service Discovery (DNS-SD), among others.

Avahi is an open source free implementation of the Zeroconf, including the mDNS and the DNS Service Discovery. Avahi is currently a de facto standard implementation of the Zeroconf for Linux and *BSD operating systems. Avahi also implements a source code API compatibility layer for Bonjour. Therefore, we have decided to use the Bonjour API, since it is available on all operating systems, where either Bonjour or Avahi are available.

In our implementation, we use the *dnssd_erlang* library, which provides an Erlang interface for the Bonjour API [66]. Each agent, during its start procedure, registers itself in the local DNS-SD registry (as a provider of the monitoring service) and receives a list of other monitoring agents. At the same time, it spawns a process, which listens for newly registered agents and adds them to the list. Note that the Avahi/Bonjour DNS-SD service, running in the operating system, automatically broadcasts information about all registered services to all computers in the LAN. It also automatically removes from this registry processes, which have been terminated. In this way, agents that leave the system are automatically deregistered.

This approach provides a much more flexible way of handling discovery of agents in the LAN than the eXAT AMS. In fact the eXAT AMS can be, with relative ease, extended to support the DNS-SD based agents discovery in the LAN. However, in our case, agent discovery is implemented directly in agents, without use of the AMS.

Knowing how agents can find information about other agents that are available in the system at any given moment, let us now describe algorithms that use this information and provide the monitoring infrastructure. In particular, we will provide details of broadcasting through the spanning tree, building the spanning tree, and the neighbour selection.

4.2. Broadcasting through the spanning tree

We had to make sure that agents can effectively broadcast information in the Grid, while the monitoring system is as non-intrusive as possible. In the case of a system consisting of hundreds of nodes, if a naive broadcast (exchanging information between every pair of agents) is used, network will become unnecessarily loaded. On the other hand, we had to ensure that the system will, with high probability, survive the failure of a single agent, node or a connection. In such case, the monitoring system should be able to send messages between the remaining agents and report accurately where the problem occurred. Furthermore, it should be able to successfully deal with leaving nodes (agents). Therefore, since we decided to use a spanning tree as the network topology of the monitoring system, this case should be handled in exactly the same way as if that node (agent) had crashed. When developing the logical monitoring network topology, we have taken into account the following issues:

- network saturation with messages exchanged between agents,
- need to detect and handle failures of:
  - limited number of nodes (in limited time),
  - limited number of links between nodes (in limited time).

The proposed solution consists of two stages. First to build a spanning tree of agents/nodes and to use it to broadcast messages. This guarantees that every agent will receive information just once (since the tree is by definition acyclic). Note that the information will be propagated regardless of broken links between specific nodes, as long as the network graph is connected (since in every connected graph the spanning tree exists). Second, to reasonably increase the number of agents monitoring each-other. In other words, the agent monitoring process should go beyond the basic spanning tree structure.

To build the spanning tree we use the well-known token-based distributed depth-first search [48] algorithm. Its core is based on passing a token along the edges of the graph; where the token contains the following information:

- state (FORWARD or RETURN),
- sender,
- list of visited nodes.

Upon reception of a token an agent undertakes the following actions:

1. If the token is in FORWARD state:
   (a) remember sender as parent
   (b) add current node to visited list
   (c) start children registration
2. If there are no unvisited nodes among the neighbours:
   (a) return the token in RETURN state to parent
3. Otherwise:
   (a) send token in FORWARD state to first unvisited neighbour
   (b) register that neighbour as child

One of the visible problems of this approach, when applied to monitoring of a dynamic system, is the need to rebuild the spanning tree in the case of a node, or a link, failure, as well as in the case of a node entering[5] or leaving the Grid. We allow the rebuild process to be initiated by any agent in the system. Such agent will initiate the spanning tree re-build as soon as it discovers that it no longer can communicate with one of its child nodes in the spanning tree. To discover it swiftly, an agent always monitors its children in the tree (through periodic pinging). Since many agents can initiate the tree rebuilding process at almost the same time (note that processes resulting in the need to rebuild the spanning tree can happen in multiple locations; e.g. a node failure and a node leaving the Grid can occur concurrently in various locations within the Grid), mechanism of breaking ties is needed (so that only a single spanning tree will result).

To implement such mechanism, we have modified the basic algorithm. Here, observe that each agent has its own unique identifier and thus it is possible to compare these identifiers using lexicographical ordering of their names. Thus, along with the token, we send the identifier of an agent that initiated the spanning tree rebuild process. Now, if an agent receives another FORWARD token before the RETURN token (that it has forwarded earlier), it checks whether the ID of the agent initiating another tree-rebuild process precedes the ID of the agent that initiated the previous one. If this is so, the previous FORWARD token is forgotten, which will lead to the previous rebuild process to time out, and yield no result, while the new rebuild process will continue, eventually reaching the previous search initiator. Otherwise the received token is ignored. After the spanning tree is rebuild, a special FINISHED message is propagated through it. Only after the reception of that message agents are ready to accept any further spanning tree search request (FORWARD tokens), regardless of the initiator ID. Such modification guarantees that in the case where multiple nodes initiate search concurrently, only one will succeed.

Spanning tree (re)built using this algorithm will be used for broadcasting data through agent's network. Current approach does not take into consideration the physical structure of the network over which agent's are communicating. This means that pings sent through agents' spanning tree may be actually traversing the longest possible paths in physical network and imposing unnecessary stress over network devices and network links.

We believe that automatic adjustments of a spanning tree based on the ping times between agents in a cluster (which is a case of an online minimum weighted spanning tree problem [54,31]) can be implemented and we speculate that it can be moderately effective for detecting the actual physical structure of the network for small networks, if proper statistics of ping measurements are used to determine the weights of edges in the graph. While this is a research topic in its own right, and is out of scope of the current system prototype, we plan to experiment with this approach in the future.

[5] Currently new agents "appearing" in the Grid are initiating a full tree rebuild, but this can be optimized in the future by attaching such agent as a leaf of the tree

4.3. Neighbour selection

Obviously, the spanning tree guarantees only that any missing node (or agent) will be detected immediately, since every agent pings periodically his parent and all his children, as a part of a standard monitoring procedure. However, it would be impossible to discover all broken links while monitoring only the edges of the spanning tree. If there is no direct monitoring between given two nodes, a broken connection between them would remain undetected; see figure 2. There, broken link between A and C will be discovered instantly whereas lack of connection between A and B is left undetected with the depicted spanning tree. Therefore, we have decided to force agents to monitor multiple nodes.

Since we have no "external" / a'priori knowledge about the physical network topology, we do not know if given two nodes are connected directly or if they communicate through another node (which may or may not have a monitoring agent running on it, e.g. a network router). Obviously, to monitor existence of connections, it is necessary to monitor direct connections between nodes. However, without information about the actual network topology, we cannot decide, which other nodes should be monitored by an agent. Therefore, we have decided to employ a probabilistic approach: each agent monitors a set of random neighbours. Here, we are satisfied that, with certain probability, all physical connections are covered. Obviously, the question remains, how many nodes, not belonging to the spanning tree, should an agent monitor?

To be absolutely certain that any broken network link will be detected (in the worse case, when each pair of nodes has distinct physical connection) each agent should monitor all other agents in the Grid. In some cases this would be acceptable, however, for larger Grids and sparse network topologies, it would lead to unnecessary network load (similarly to the naive broadcast considered in section 4.2). As a solution, we propose a parametrized value k, which defines a trade-off between the level of robustness we want to achieve, and the level of network saturation, which is acceptable in the current environment. Here, $k = 0$ means that no additional monitoring besides that through the spanning tree edges takes place, while $k = n$, where n is the number of nodes in the Grid, corresponds to the situation when every agent is monitoring all other agents.

To build the actual monitoring dependencies, we proceed as follows. Based on the list of all Grid nodes (which is available at each node, see section 4), each agent starts monitoring k random nodes. Choosing the exact value of k depends on network topology, throughput of it's links and the desired robustness of the monitoring (in a sense of ability of the monitoring infrastructure to detect complex network failures, and time needed to detect them). Unfortunately, to the best of our knowledge, the value of k should be selected experimentally. Higher k ensures better robustness, but increases bandwidth utilization, resulting from the monitoring process, and vice versa. At first, some of this monitoring is bi-directional, which leads to an unbalanced number of connections at each node. To make it closer to the requested value k, we employ a simple diffusion-based approach. Whenever two agents ping each other, they exchange information about the number of "neighbours" each has and compare them. If it turns out

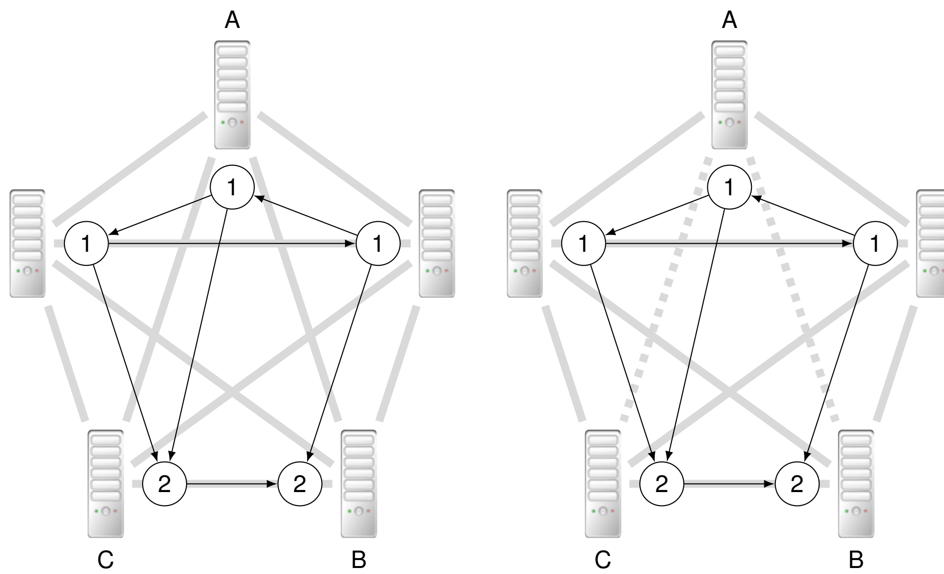


Fig. 2. Sample Grid with physical network connections. Thick gray edges denotes working network connections, dotted gray edges denote broken connections. Thin black edges denotes agents monitoring activity.

that one number is higher than the other (with difference larger than one), one agent will "pass" monitoring a specific node to the other. This will decrease the number of nodes it is monitoring and increase the number of nodes monitored by the other. To ensure data consistency, each exchange is realized as an elementary transaction, which has to be confirmed from both sides. After the diffusion is completed, in most cases monitoring ceases to be bi-directional and number of monitoring neighbours should be close(r) to the desired k. Our implementation of monitoring balancing is simplistic, and it may be slow to converge. Examples of better load-balancing algorithm of this type are well described in the literature (see, for instance [59]), and we plan to experiment with them in the future.

Furthermore, in the future versions of the system, this model will be slightly extended. When the number of neighbours monitored by each agent is stabilized, they will occasionally (after a predefined time) exchange monitoring obligations (pass one monitoring obligation and receive another). This will introduce neighbours rotation across the network. As a result, broken links that could not be discovered immediately, will be eventually discovered (after some time).

4.4. Knowledge inferencing

During work of the system, agents exchange information they have obtained (e.g. detected dead agents, detected dead nodes, performance metrics, etc.), using the broadcast through the spanning tree. Based on gathered cumulative information, agents are capable of interfering additional knowledge. Here, information gathered from other agents is added to agent's own knowledge base, which is used by it's expert system. These facts are also annotated with the name of the source agent. This allows to distinguish between information obtained first hand, and information obtained from other agents.

Let's consider three simple scenarios, which can be easily detected using a rule-based approach:

1. How to distinguish between a dead agent and a dead node?
2. How to distinguish between a dead node and a dead link?
3. How to detect problems of a specific feature of the system, which is running in the Grid?

For these scenarios, let us describe an appropriate set of rules, and their representation in the ERESYE inference engine.

– Case 1: We have an agent which monitors some remote agent *R* running on node *node(R)*, we can use the following rule to detect if the agent is malfunctioning, or it's the node that is dead.
  (agent *R* does not respond to pings) $\wedge$ (*node(R)* does respond to ICMP pings) $\Rightarrow$ (agent *R* is dead)

```
agent_failure(Engine, {agent_state, BNode_id, active},
                      {link_state, ANode_id, BNode_id, working},
                      {pinging_state, ANode_id, BNode_id, not_responding}) ->
   eresye:retract(Engine, {agent_state, BNode_id, active}),
   eresye:assert(Engine, {agent_state, BNode_id, inactive}).
```

(agent *R* does not respond to pings) $\wedge$ (*node(R)* does **not** respond to ICMP pings) $\Rightarrow$ (*node(R)* is unreachable)

```
node_unreachable(Engine,{link_state, ANode_id, BNode_id, broken},
                        {pinging_state, ANode_id, BNode_id, not_responding})->
   eresye:assert({node_unreachable, ANode_id, BNode_id}).
```

– Case 2: We have an agent, which monitors node *A* that stops responding to pings.
  (node *A* is unreachable)$\wedge$(node *A* is unreachable for other agents too) $\Rightarrow$ (node *A* is dead)

```
node_inactive(Engine, {node_state, ANode_id, active})
   when not["{link_state,ANode_id,_,working}"];true ->
   eresye:retract(Engine, {node_state, ANode_id, active}),
   eresye:assert(Engine, {node_state, ANode_id, inactive}).
```

  (node *A* is unreachable)$\wedge$(node *A* is reachable for some other agents) $\Rightarrow$ (link between *A* and myself is broken)

```
link_failure(Engine,{link_state, ANode_id, BNode_id, broken},
                    {link_state, ANode_id, _, active}) ->
   eresye:assert(Engine, {link_failure, ANode_id, BNode_id}).
```

– Case 3:
  For Case 3 let us assume that the system is installed in a heterogeneous Grid, where each monitored feature is handled by a subset of nodes. For example, let us assume that the Grid is handling back-end operations of some medium size web service, which has a search feature, which is being handled by 3 servers *A*, *B* and *C*. The fact that those specific hosts are handling some specific feature (in this case it is a search feature), can be fed to the knowledge base of agents in the monitoring infrastructure. Such information, would allow to create rules, which would detect that something is wrong with this specific feature. For example the following rule could be instantiated:
  (all nodes with this feature have high CPU utilization) $\wedge$ (error rates for related category is elevated) $\Rightarrow$ (the feature is broken)
  Coupled with an alerting system, this rule could generate an alert, informing administrators of the system that the search feature does not work correctly. As seen from the rule above, this example also needs an additional error rate metric originating from a monitoring system, which can be easily obtained by parsing error logs of said search servers. This can be implemented, for instance, by adding an additional agent, which will parse these logs and feed results to the local monitoring agent. Let us stress that to extend our existing prototype to handle this case would be relatively easy and would require only (a) adding appropriate rules to the expert system, and (b) adding (and integrating with other agents) a log parsing agent.

While these three cases are relatively simple, they were implemented to illustrate the eXAT $\leftrightarrow$ ERESYE integration. Furthermore, these are the functionalities that not only can be helpful in actual day-to-day work of an administrator of services running on multiple servers in LAN, but also show that the

implemented system works as assumed. Obviously, the set of possible rules is naturally extensible.

5. Monitoring system at work

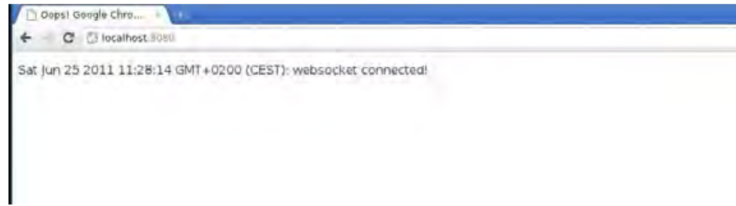


Fig. 3. Browser window after connecting to an agent.

Working with our system is very straightforward. We have prepared a wrapper script for running the agents:

```
./start_agent.sh -http_port 7778 -ws_port 8080 \
  -name platform_name
```

The http_port option denotes a port used by the eXAT for communication and the ws_port is the websocket port number used by the web interface. Finally, the $name$ parameter is mandatory for the platform name. However, it can be chosen freely, since it has no consequences for the operation of the system.

After the agent was started, the administrator can connect to it with a web browser and preview the collected information. Currently, only more recent versions of Internet Explorer, Firefox and Chrome support the WebSocket protocol, but since this support is likely to remain in these browsers in the future, we do not see it as a serious drawback. Figure 3 presents the browser window just after the connection with the administrator agent has been established.

Since the agent immediately starts gathering data, in a relatively short time user should be presented with animated plots presenting the CPU load of every node in the Grid (currently this is the metric, depiction of which has been implemented). This is illustrated in figure 4.

In the case of discovery of an inactive agent (as described in section 4.4), it is signaled by a red message appearing under its plot, as can be seen in figure 5.

Similarly, any link problems are signaled by a red text in the links state column. Since links are symmetrical, failure of a link from A to B would be always accompanied by failure of a link from B to A. This situation is depicted in figure 6.

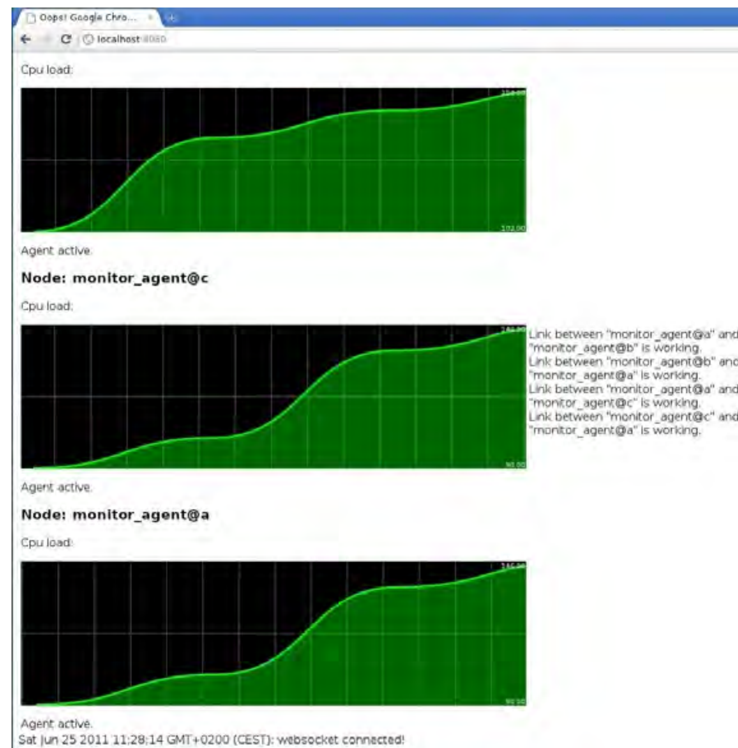


Fig. 4. Resource load plots for different nodes.

6. Experimenting with the monitoring infrastructure

In addition to the simplistic use cases described in section 5, the monitoring system has been deployed in a network of 4 Ubuntu Linux workstations at a small company in Warsaw. After the Erlang has been installed (using apt−get install erlang), the monitoring system was unpacked and started with a provided startup shell script start.sh. Each node has been detected and was able to connect into the monitoring system in time below 5 seconds. The initial spanning tree is depicted in figure 7. To observe the monitoring system at work, we have implemented an additional monitoring mechanism, which recorded ping times along the edges of the spanning tree. These times were consistent with the physical layout of the network, which consisted of multiple 100 Mbps switches connected in a star topology.

After the system has been started it reported the CPU load of each computer in the network. Another metric, which has been presented to the user, was the recorded ping time between agents, which were pinging each other. Due to the mechanisms of broadcasting most locally harvested information, a user could look up the state of all nodes in the network at any node. Full knowledge base

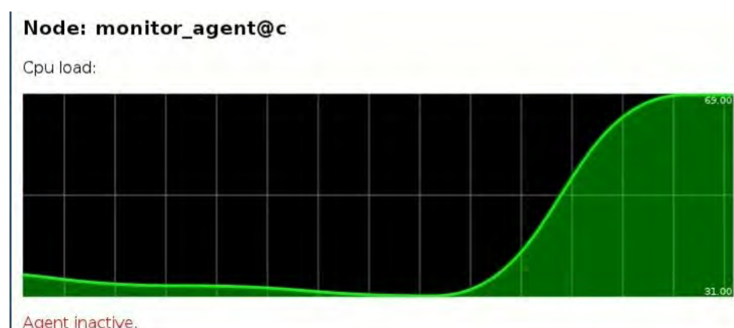


Fig. 5. Inactive agent discovered.

of one of the agents, including mentioned metrics, in the cluster is presented in figure 8.

When the root switch has been shut down, agents started to detect ping-timeouts, which resulted in creation a new spanning tree, as presented in figure 9. Agents were able complete this task in around 15 seconds, since each attempt of building a spanning tree resulted in multiple timeouts. A full knowledge base of one of the agents after shutdown of a switch is illustrated in figure 10.

Separately we have collected measurements at the root switch, to estimate the bandwidth overhead introduced by our monitoring system. With the pinging interval set to 1 second, the average bandwidth overhead was approximately 70 kbps. Broadcasting overhead was changing linearly depending on the pinging interval.

After the root switch has been restarted, agents were able to detect their presence again, and were able to rebuild a new full spanning tree in around 7 seconds.

These experiments, performed in a realistic situation (though admitting, that the network was somewhat small) show that mechanisms outlined in the paper are working reasonably well and can be useful in practical use cases as a “smarter” alternative to software like Ganglia. Let us also note, that in the near future we plan to experiment with much larger network to test the scalability of the proposed approach.

7. Experiences with eXAT

Since no other projects realized in eXAT are known to us, and no descriptions of experiences regarding writing agents with this framework could be found in the literature, we have decided to present some thoughts on working with this tool.

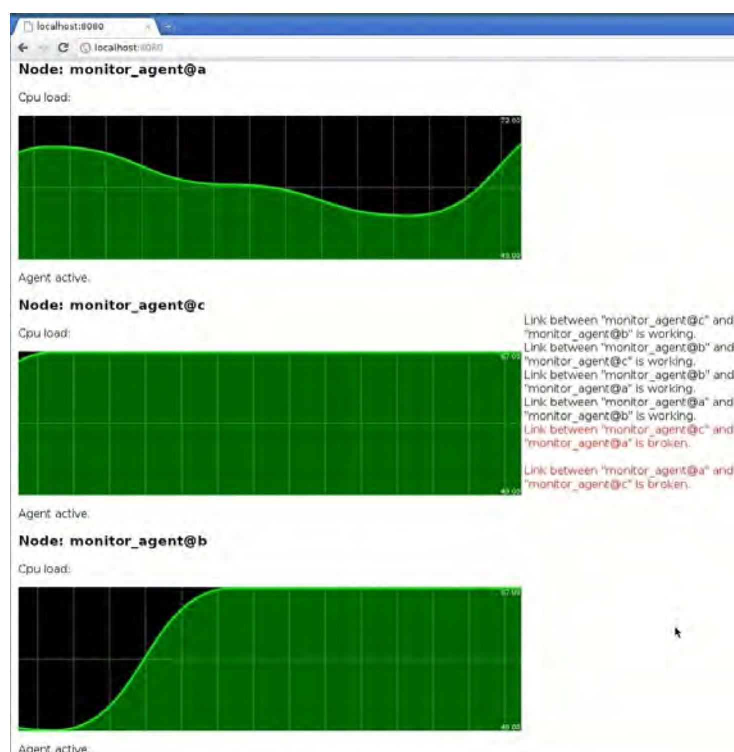


Fig. 6. Broken links discovered.

7.1. eXAT-JADE interoperability

Since eXAT uses the FIPA compliant ACL messaging model, it should be possible to communicate with other agents platforms supporting the standard. We verified this assumption experimentally, by establishing a bidirectional communication between an eXAT agent and a JADE agent. The connection proved to be easy to set and no platform modifications or special syntax were necessary (see, below). However, the original version of eXAT relied on a custom HTTP server which was not fully reliable. In our project we switched to an external solution — the Misultin [9], which helped with handling most of the JADE MTP messages. We haven't tested all possible communication types between the two systems, hence some other interoperability problems can still be present.

Despite possible inconveniences caused by the immaturity of eXAT, it seems to be feasible to build heterogeneous agents systems based on eXAT and JADE. Furthermore, using an eXAT agent to create an interface for the existing Erlang applications is also an option (which, for instance, could be used if eXAT agents would be incorporated into the *AiG* project).

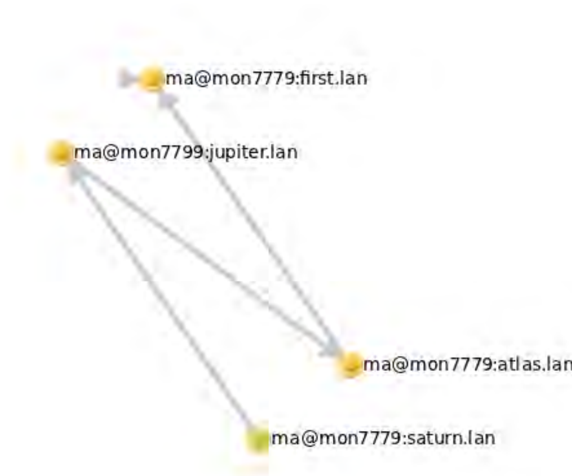


Fig. 7. Initial state of the spanning tree.

We shall now demonstrate how to make eXAT and JADE agents exchange messages. In the eXAT all ACL-related functions are defined in the module *acl*. The basic function for sending a message is acl:sendacl(message), where the *message* is the following Erlang record (defined in the acl.hrl header file):

```
#aclmessage {speechact, sender, receiver, 'reply-to', content,
language, encoding, ontology, protocol, 'conversation-id',
'reply-with', 'in-reply-to', 'reply-by'}
```

To pass a message between the agents we have to fill correctly the sender and the receiver fields. Both of them should be records of type:

```
#'agent-identifier' {name, addresses}
```

where the name has the form agent_name@platform_name, while the addresses is a list of network addresses in the form http://ip:port/acc. Finally, the #'agent-identifier' is defined in the fipa_ontology.hrl.

For example a command for sending simple ping message to an agent from a different platform (another instance of eXAT, or any other FIPA-compliant platform) running on the local host, could look as follows:

```
acl:sendacl(#aclmessage{speechact = query, content = "ping",
sender = #'agent-identifier'{john_the_agent@platform1,
[http://localhost:7778/acc]},
```

```
["agent","ma@mon7779:atlas.lan"]
["agent","ma@mon7779:first.lan"]
["agent","ma@mon7779:saturn.lan"]
["agent","ma@mon7799:jupiter.lan"]
["agent_node","ma@mon7779:atlas.lan","mon7779:atlas.lan"]
["agent_node","ma@mon7779:first.lan","mon7779:first.lan"]
["agent_node","ma@mon7779:saturn.lan","mon7779:saturn.lan"]
["agent_node","ma@mon7799:jupiter.lan","mon7799:jupiter.lan"]
["agent_ping","ma@mon7779:atlas.lan","ma@mon7779:first.lan","success",22330]
["agent_ping","ma@mon7779:atlas.lan","ma@mon7799:jupiter.lan","success",11020]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:atlas.lan","success",49065]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:saturn.lan","success",38888]
["agent_ping","ma@mon7779:first.lan","ma@mon7799:jupiter.lan","success",26564]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7779:first.lan","success",11104]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7799:jupiter.lan","success",17932]
["agent_ping","ma@mon7799:jupiter.lan","ma@mon7779:atlas.lan","success",13237]
["agent_ping","ma@mon7799:jupiter.lan","ma@mon7779:first.lan","success",13966]
["agent_ping","ma@mon7799:jupiter.lan","ma@mon7779:saturn.lan","success",17928]
["agent_status","ma@mon7779:atlas.lan","alive"]
["agent_status","ma@mon7779:first.lan","alive"]
["agent_status","ma@mon7779:saturn.lan","alive"]
["agent_status","ma@mon7799:jupiter.lan","alive"]
["cpu_state","atlas.lan",8]
["cpu_state","first.lan",340]
["cpu_state","jupiter.lan",282]
["cpu_state","saturn.lan",3]
["host","atlas.lan"]
["host","first.lan"]
["host","jupiter.lan"]
["host","saturn.lan"]
["host_ping","atlas.lan","first.lan","success"]
["host_ping","first.lan","atlas.lan","success"]
["host_ping","first.lan","jupiter.lan","success"]
["host_ping","first.lan","saturn.lan","success"]
["host_ping","saturn.lan","first.lan","success"]
["local_agent","ma@mon7779:first.lan"]
["local_node","mon7779:first.lan"]
["neighbours","ma@mon7779:atlas.lan",["ma@mon7799:jupiter.lan"]]
["neighbours","ma@mon7779:first.lan",["ma@mon7779:atlas.lan"]]
["neighbours","ma@mon7779:saturn.lan",[]]
["neighbours","ma@mon7799:jupiter.lan",["ma@mon7779:saturn.lan"]]
["node","mon7779:atlas.lan"]
["node","mon7779:first.lan"]
["node","mon7779:saturn.lan"]
["node","mon7799:jupiter.lan"]
["node_host","mon7779:atlas.lan","atlas.lan"]
["node_host","mon7779:first.lan","first.lan"]
["node_host","mon7779:saturn.lan","saturn.lan"]
["node_host","mon7799:jupiter.lan","jupiter.lan"]
["parent","ma@mon7779:atlas.lan","ma@mon7779:first.lan"]
["parent","ma@mon7779:first.lan","ma@mon7779:first.lan"]
["parent","ma@mon7779:saturn.lan","ma@mon7799:jupiter.lan"]
["parent","ma@mon7799:jupiter.lan","ma@mon7779:atlas.lan"]
["stree","children",["ma@mon7779:atlas.lan"]]
["stree","origin","ma@mon7779:first.lan"]
["stree","parent","ma@mon7779:first.lan"]
["stree","status","initial"]
```

Fig. 8. Initial state of agent's knowledge base.

```
receiver = #'agent-identifier'{tom_the_agent@platform2,
[http://localhost:7779/acc]}).
```

Here, an agent named *john_the_agent* sends a message to an agent *tom_the_agent*. These two agents are running on two separate platforms started on the localhost and distinguished by their port numbers.

Knowing the API of the *acl* module, we can create two simple agents – an eXAT agent (see listing 1.1) and a JADE agent (see listing 1.2)—and make them exchange messages.

Let us now analyse the listing in Figure 1.1. The eXAT code starts with the necessary header declarations. Next, follows the declaration of the function extends, which is a part of the eXAT object system syntax—it means that the

Listing 1.1. eXAT ping agent

```
-module(exat_agent).
-export([extends/0]).
-export([pattern/2,event/2,action/2,on_starting/1,
  do_request/4,start/0]).
-include_lib("exat/include/acl.hrl").
-include_lib("exat/include/fipa_ontology.hrl").

extends()-> nil.

pattern(Self, request)-> [#aclmessage{speechact='REQUEST'}].

event(Self, evt_request)-> {acl, request}.

action(Self, start)-> {evt_request, do_request}.

fellow_agent()-> #'agent-identifier'{
  name = "jadeagent@jadeplatform",
  addresses = ["http://localhost:7778/acc"]}.

on_starting(Self)->
  io:format("[Agent:~w] Starting\n", [object:agentof(Self)]),
  acl:sendacl(#aclmessage{speechact = 'REQUEST',
    content = "ping", sender = Self,
    receiver = fellow_agent()}).

do_request(Self, EventName, Message, ActionName)->
  io:format("[Agent:~w] Request received from agent ~p\n",
    [object:agentof(Self), Message#aclmessage.sender]),
  object:do(Self, start).

start()->
  agent:new(the_exat_agent,[{behaviour, exat_agent}]).
```

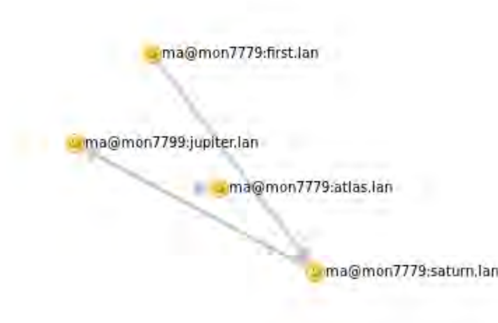


Fig. 9. Spanning tree after a switch shutdown.

agent class has no explicit parent. The next three definitions are necessary for the event mapping. The fellow_agent definition is a convenient way of storing the agent address—it is a functional equivalent of a global (class) variable. The on_starting is a function executed by the framework as soon as an agent is created. It contains the code for sending the message. The do_request is executed after receiving a REQUEST message. It returns a special construct, the object:do(Self, start), which informs that an agent is still in the state start.

Let us now take a look into the JADE agent code (in figure 1.2). Functionality of this agent is exactly the same as that of the eXAT agent, but the structure is slightly different. The setup is equivalent to the on_starting. There are no events for the message reception, so we have to manually fetch messages with the receive method. To do this in a loop-like manner, we exploit the JADE CyclicBehaviour.

7.2. eXAT vs. JADE—implementation details

Let us now take a look at the eXAT and the JADE agent platforms side-by-side. Since JADE is substantially more mature than eXAT it, obviously, has a much richer set of features. Nevertheless, their basic FIPA-compliant functionalities are roughly equivalent.

One of JADE features nonexistent in eXAT is the possibility of sending Java objects in ACL messages, without special encoding, between agents of the

Listing 1.2. JADE ping agent

```
import jade.core.Agent;
import jade.core.AID;
import jade.lang.acl.ACLMessage;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.MessageTemplate;

public class JADEAgent extends Agent {

private MessageTemplate template =
   MessageTemplate.MatchPerformative(ACLMessage.REQUEST);

protected void setup() {
   System.out.println("[Agent: "+this.getLocalName() +
         "] Starting");

   addBehaviour(new CyclicBehaviour(this) {
      public void action() {
         ACLMessage msg = myAgent.receive(template);
         if (msg != null) {
            System.out.println("[Agent: " +
               myAgent.getLocalName() +
               "] Request received from agent " +
               msg.getSender().getName());
         }
         else {
            block();
         }
      }
   } );

   sendMessage();
}

private void sendMessage() {
   AID r = new AID ("the_exat_agent@exatplatform", AID.ISGUID);
   r.addAddresses("http://localhost:7779/acc");
   ACLMessage aclMessage = new ACLMessage(ACLMessage.REQUEST);
   aclMessage.addReceiver(r);
   aclMessage.setContent("ping");
   this.send(aclMessage);
}
}
```

```
["agent","ma@mon7779:atlas.lan"]
["agent","ma@mon7779:first.lan"]
["agent","ma@mon7779:jupiter.lan"]
["agent","ma@mon7779:saturn.lan"]
["agent_node","ma@mon7779:first.lan","mon7779:first.lan"]
["agent_node","ma@mon7779:jupiter.lan","mon7779:jupiter.lan"]
["agent_node","ma@mon7779:saturn.lan","mon7779:saturn.lan"]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:atlas.lan","fail"]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:jupiter.lan","success",65696]
["agent_ping","ma@mon7779:first.lan","ma@mon7779:saturn.lan","success",15853]
["agent_ping","ma@mon7779:jupiter.lan","ma@mon7779:first.lan","success",23201]
["agent_ping","ma@mon7779:jupiter.lan","ma@mon7779:saturn.lan","success",61054]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7779:atlas.lan","fail"]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7779:first.lan","success",65940]
["agent_ping","ma@mon7779:saturn.lan","ma@mon7779:jupiter.lan","success",21071]
["agent_ping_fail","ma@mon7779:atlas.lan",1337029505126351,15]
["agent_status","ma@mon7779:atlas.lan","left"]
["agent_status","ma@mon7779:jupiter.lan","alive"]
["agent_status","ma@mon7779:saturn.lan","alive"]
["cpu_state","atlas.lan",8]
["cpu_state","first.lan",20]
["cpu_state","jupiter.lan",12]
["cpu_state","saturn.lan",43]
["host","atlas.lan"]
["host","first.lan"]
["host","jupiter.lan"]
["host","saturn.lan"]
["host_ping","first.lan","atlas.lan","success"]
["host_ping","first.lan","jupiter.lan","success"]
["host_ping","first.lan","saturn.lan","success"]
["host_ping","jupiter.lan","atlas.lan","success"]
["host_ping","jupiter.lan","first.lan","success"]
["host_ping","jupiter.lan","saturn.lan","success"]
["host_ping","saturn.lan","atlas.lan","success"]
["host_ping","saturn.lan","first.lan","success"]
["host_ping","saturn.lan","jupiter.lan","success"]
["local_agent","ma@mon7779:first.lan"]
["local_node","mon7779:first.lan"]
["node","mon7779:atlas.lan"]
["node","mon7779:first.lan"]
["node","mon7779:jupiter.lan"]
["node","mon7779:saturn.lan"]
["node_host","mon7779:atlas.lan","atlas.lan"]
["node_host","mon7779:first.lan","first.lan"]
["node_host","mon7779:jupiter.lan","jupiter.lan"]
["node_host","mon7779:saturn.lan","saturn.lan"]
["parent","ma@mon7779:atlas.lan","ma@mon7779:atlas.lan"]
["parent","ma@mon7779:first.lan","ma@mon7779:saturn.lan"]
["parent","ma@mon7779:saturn.lan","ma@mon7779:jupiter.lan"]
```

Fig. 10. Knowledge base of one of agents after a switch shutdown.

same platform. This feature is not FIPA-compliant, but allows to greatly optimize communication between local agents. In fact one can use it to share persistent data across local platform. This is impossible to achieve with eXAT and only partially possible with standard Erlang messages (only one special type of binary data can be shared). While lack of shared memory is one of foundations of Erlang concurrency model (processes vs. threads), skipping unnecessary objects encoding is JADE's advantage over eXAT. Since this feature is only useful in local systems, we can assume that eXAT, like the whole Erlang, is designed to build systems distributed between physically different machines.

Keeping this in mind, and considering the threading model, we can speculate about ideal agent size encouraged by the frameworks. JADE authors sug-

gest that there should be only a few "bigger agents" running within the platform, as each of them is connected with its own thread (in most Java implementations, a native thread) and increasing the number of agents would considerably increase the resource consumption. All the small-scale multi-tasking needed in the system should be based on JADE behaviours, which are lightweight and scheduled by the framework. However one has to remember, that behaviours of one agent have to be executed on one CPU as they belong to a single thread.

It is quite different in the eXAT, where all multi-tasking is based on the Erlang processes, running within the Erlang Virtual Machine (Erlang VM). They are lightweight and can be executed on any CPU (Erlang VM does the scheduling and workload balancing). This means that the eXAT agents can be as small or as big as needed, and this should not influence the performance of the system. Additionally the Erlang VM can handle millions of processes [68], hence there is no hard limits on the number of agents started in the eXAT.

7.3. Integration with ERESYE

Another topic that needs to be discussed is the ERESYE, and its integration with the eXAT. As stated above, the ERESYE is a full-feature rule-based inference engine implementing the RETE algorithm. It allows stating facts and specifying rules connecting these facts.

Facts are represented with standard Erlang data types, especially tuples. Rules are written using a normal function declaration form. The general syntax has been illustrated in section 4.4.

The inference algorithm is similar to an automatic logical inference, known from Prolog, but is optimized for the situation where asserted facts change dynamically, and exploits a form of eager evaluation. As soon as a new fact is asserted, all rules depending on it, are marked as partially satisfied. If the fact was the last not asserted prerequisite of a rule, the action is executed immediately.

It is also possible to synchronously wait for a certain fact to be asserted in the main code of the agent. This corresponds to an agent behaviour, which can be expressed as: "do not do a thing until you are sure that A is true."

This approach makes possible effective reasoning, in real-time, in a dynamically changing environment. It is well-suited for the needs of responsive intelligent agents, as it is possible to use ERESYE as the central decision-making unit, which controls agent actions—a true equivalent of agent's "brain."

However, usage of ERESYE in eXAT goes further than that. Included tool makes it possible to translate an ontology (as defined by FIPA [55]) expressed in a simple hierarchical syntax, to a set of Erlang records suitable for storing as resolution engine facts. This is the basis for the so-called semantic layer of the eXAT platform. When the fipa_semantics_simple semantics is enabled for the eXAT agent, all "INFORM" speech acts are automatically added to the knowledge base of a given agent (called "mind" in eXAT). Additionally the fipa_semantics_simple is able to automatically check feasibility condition of an ACL message and perform a rational effect in accordance with the message's

communicative act. For example, a rational effect upon receiving a "CONFIRM" message is to assert it's content in the agent's knowledge base. With the semantic layer it is done automatically without any explicit message processing code [64]. For more information about handling of ontologies in ERESYE, please refer to [62].

In our opinion, the union between the eXAT and the ERESYE works very well. However, we found this system to be overly complicated to use. For instance, in our case, it was easier to explicitly use the ERESYE reasoner through it's API. However, we find the concept of integrating of eXAT agents with a reasoner, which is capable of parsing ontologies and processing ontological concepts, a very promising and interesting solution. Furthermore, integrating reasoners directly with the platform is something that should be considered also in other agent platforms.

7.4. eXAT not eXATly perfect

Unfortunately the eXAT, as an experimental tool, has some serious drawbacks. Some of them are our subjective opinions, stemming from thoughts on the eXAT design. This system is based on a custom implemented object-orientated emulation layer, which adds considerable complexity into the project. Constructing it feels a bit like "swimming against the current," because Erlang is designed as a purely functional language, and according to our beliefs, every agent aspect can be expressed in that fashion. Furthermore, it adds additional, unnecessary, overhead to message passing between (at least two) processes, which are used by a single agent. Finally, it adds an unnecessary learning curve, and decreases programming efficiency compared to the pure Erlang code, which is arguably a very efficient language to write in [69].

To clarify our opinion let us analyze a single case in depth. In eXAT, agents are modelled after the finite-state machine, which allows to bind different actions as a response to specific events, depending on the agent current state, thus introducing branching in code execution.

Listing 1.3. eXAT finite state machine description example

```
pattern(Self, request)-> [#aclmessage{speechact='REQUEST'}].

event(Self, evt_request)-> {acl, request}.

action(Self, start)-> {evt_request, do_request}.

action(Self, finalizing)-> {evt_request, decline_request}.
```

In 1.3, an agent executes the action do_request when in state start, but upon reception of the REQUEST message in state finalizing, it will respond with the decline_request.

For a finite-state machine, its single state is the only memory it has. In the case of a computer program, the state is distributed and represented as separate states of multiple variables. Each step of execution of a computer program

can be a conditional clause, depending on the value of any stored variable. Therefore, the eXAT event mapping can be considered "syntactic sugar," i.e., element that does not bring new functionality, but makes the code easier to read. It is an alternative to wrapping event handling in explicit conditional instructions. However, Erlang has native mechanism for handling such cases, called pattern matching. It is possible to write multiple variants of the same function with different parameter patterns, and during the execution of the code, the correct version will be chosen, depending on the actual parameters. Therefore, the same goal is achieved, while maintaining brevity and sticking to Erlang's functional style.

Listing 1.4. Erlang finite state machine description example

```
handle_request(start, Request)->
    % do_request code
    return_value;
handle_request(finalizing, Request)->
    % decline_request code
    return_value.
```

The code in 1.4 handles or declines the request depending on the value of the first parameter. As we can see, the eXAT obscures this syntax with its own mechanism. We consider it a drawback, because it does not take full advantage of Erlang's strengths and provides a "replacement mechanism," which (particularly, for the Erlang programmers) may feel unfamiliar and unnecessarily complicated. To eliminate such complications introduced by the framework, we have implemented a simple_agent OTP behaviour, closely modelled after the OTP industry-standard gen_server behaviour[32]. It simplifies implementing agents, while following the Erlang coding style, and the OTP standards.

There are also other places where the eXAT was not written according to the Erlang/OTP coding standards, which are widely accepted in the Erlang community. Here are a few examples from eXAT source code demonstrating bad practices:

- Using non-OTP standard indentation style[7] and whitespace usage (spaces after function names):

```
inform (Message) ->
  sendacl (Message#aclmessage {speechact = 'INFORM'}).
```

- Using lists instead of tuples as messages, and not using records for storing state of long-running processes:

```
handle_call([acl_erl_native, Acl], From,
            [AgentName, AclQueue, AgentDict, ProcessQueue]) ->
    %%io:format("[Agent] Received ACL=~w\n", [Acl]),
    {AclQueue1, ProcessQueue1} =
        perform_re(AgentName, AgentDict,
                   Acl, AclQueue, ProcessQueue),
    {reply, ok, [AgentName, AclQueue1, AgentDict, ProcessQueue1]}.
```

– Excessive use of *if* conditional, and using the *length* function where a simple *case* with pattern matching would be much more succinct and efficient:

```
handle_cast([getmessage, From],
            [AgentName,AclQueue,AgentDict,ProcessQueue])->
    if
        length(AclQueue) > 0 ->
            ProcessQueue1  = ProcessQueue,
            [Message | AclQueue1] = AclQueue,
            catch(From ! Message);
        true ->
            ProcessQueue1 = ProcessQueue ++ [{nil, From}],
            AclQueue1 = AclQueue
    end,
   {noreply,[AgentName,AclQueue1,AgentDict,ProcessQueue1]};
```

– Excessive defensive programming, discouraged by the Erlang Programming Rules and Conventions [35] and research [19]:

```
get_conds ({Module, Func}, Ontology, ClauseID) ->
  File = lists:concat ([Module, '.erl']),
  case epp:parse_file (File, ["."], []) of
    {error, OpenError} ->
     io:format(">> Errore!!!~n~w:~w~n",[{Module,Func},OpenError]),
      error;
    {ok, Form} ->
      Records = get_records (Form, []),
      %%io:format (">> Records ~p~n", [Records]),
      case search_fun (Form, Func, Records) of
        {error, Msg} ->
         io:format(">> Errore!!!~n~w:~s~n",[{Module,Func},Msg]),
          error;
        {ok, CL} ->
          ClauseList =
            if
              ClauseID > 0 -> [lists:nth (ClauseID, CL)];
              true -> CL
            end,
          %%io:format ("Clauses ~p~n", [ClauseList]),
          SolvedClauses =
            if
              Ontology == nil -> ClauseList;
              true -> eresye_ontology_resolver:resolve_ontology (ClauseList,
                                                                  Ontology)
            end,
          %%io:format (">>> ~p~n", [SolvedClauses]),
          case read_clause(SolvedClauses, [], Records) of
            {error, Msg2} ->
              io:format(">> Errore!!!~n~w:~s~n",[{Module,Func}, Msg2]),
              error;
            CondsList -> CondsList
          end
      end
  end.
```

– Occasionally implementing features, which are already present in the Erlang/OTP standard library. The following code duplicates the features of the ETS tables.

```
property_server (Dict) ->
  receive
    {From, get, AttributeName} ->
```

```
    case catch (dict:fetch (AttributeName, Dict)) of
      {'EXIT', _} -> From ! {ack, undef};
      Other -> From ! {ack, {value, Other}}
    end,
    property_server (Dict);
  {From, set, AttributeName, AttributeValue} ->
    From ! {ack, ok},
    property_server (dict:store (AttributeName, AttributeValue, Dict));
  {From, list} ->
    X = dict:fetch_keys (Dict),
    From ! {ack, X},
    property_server (Dict);
  {From, list_values} ->
    X = dict:to_list (Dict),
    From ! {ack, X},
    property_server (Dict);
  {From, exit } ->
    From ! {ack, ok};
  Other ->
    property_server (Dict)
end.
```

Finally, note that the eXAT project was effectively discontinued by it's authors, since there were no updates to it since 2005; it has no community support, and no production systems using the eXAT platform could have been found. Furthermore, eXAT has a very sparse documentation. Additionally, it uses some custom libraries for well known tasks like implementing HTTP servers, when reusing existing libraries would lead to better tested and a more stable code.

While working on this project we made a number improvements to the eXAT, which can be found at the github.com/gleber/exat. Among others, we have replaced the internal custom-made HTTP server with the well-established Erlang implementation of the HTTP server called Misultin. As mentioned, we simplified creation of agents with the simple_agent behaviour. We switched to the rebar-managed compilation process, which is the current de-facto standard in the Erlang community. Finally, we have updated the eXAT to work with latest version of the Erlang/OTP distribution.

8. Concluding remarks

In this paper we have introduced an agent-based monitoring system for LAN / Grid / Cloud infrastructure. The prototype of the system has been implemented using an Erlang-based eXAT agent platform. After implementing the system, we found Erlang to be a good fit for the task at hand. Furthermore, the eXAT agent platform was acceptable as the tool to implement the monitoring system, though in needed at least some improvements (which we have completed). For the agent reasoning we have used the ERESYE rule-based expert system, natively integrated with the eXAT. This integration worked very-well and we plan to use it to cover more extensive cases of reasoning about the state of the system. In the near future we plan to: (a) fix additional issues with the eXAT, primarily continue refreshing it to match the state of the art of Erlang today (these improvements will be made available to the community); (b) expand the set of

network topologies and detected problems; (c) integrate the eXAT monitoring system with the resource managing agents in the *AiG* project, and (d) complete additional research outlined in the paper. We will report the results of our work in subsequent publications.

References

1. http://msdn.microsoft.com/en-us/library/windows/desktop/aa373083
2. http://code.google.com/p/parfait/
3. http://developer.gnome.org/libgtop/stable/libgtop-GlibTop.html
4. Avahi homepage. http://avahi.org/
5. Axum - microsoft's actor programming language, http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx
6. Boinc project. http://boinc.berkeley.edu/
7. Erlang: submitting patches. https://github.com/erlang/otp/wiki/submitting-patches
8. libactor project documentation. http://www.chrismoos.com/
9. Misultin. https://github.com/ostinelli/misultin
10. Smoothie-Charts library. http://smoothiecharts.org/
11. Theron - c++ concurrency library. http://www.theron-library.com/
12. Zero Configuration Networking (Zeroconf). http://www.zeroconf.org/
13. Jess website. http://www.jessrules.com/ (2011), sandia National Laboratories
14. Agha, G., Hewitt, C.: Concurrent programming using actors: Exploiting large-scale parallelism. In: FSTTCS. pp. 19–41 (1985)
15. Aloisio, G., Cafaro, M., Fiore, S., Mirto, M., Vadacca, S.: Greic data gather service: a step towards p2p production grids. In: Proceedings of the 2007 ACM Symposium on Applied Computing (SAC). pp. 561–565. Seoul, Korea (2007)
16. Apple, I.: Bonjour overview. http://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/NetServices/Introduction.html#//apple_ref/doc/uid/10000119i, december, 2011
17. Arcangeli, J., Maurel, C., Migeon, F.: An api for high-level software engineering of distributed and mobil applications. In: Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems. pp. 155–. IEEE Computer Society, Washington, DC, USA (2001), http://dl.acm.org/citation.cfm?id=874065.875773
18. Arcieri, T.: (2008), http://revactor.github.com/
19. Armstrong, J., Helen, T.: Making reliable distributed systems in the presence of software errors (2003)
20. Aversa, R., Di Martino, B., Mazzocca, N., Venticinque, S.: Magda: A mobile agent based grid architecture. Journal of Grid Computing 4, 395–412 (2006)
21. Ayres, J., Eisenbach, S.: Stage: Python with actors. In: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering. pp. 25–32. IWMSE '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/IWMSE.2009.5071380
22. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: Jade—a white paper. Tech. rep., Telecom Italia Lab, EXP Online (2003), december 2011

23. Brazier, F.M.T., Mobach, D.G.A., Overeinder, B.J., van Splunter, S., van Steen, M., Wijngaards, N.J.E.: Agentscape: Middleware, resource management, and services. In: Proceedings of the 3rd International SANE Conference (SANE 2002). pp. 403–404. Maastricht, The Netherlands (May 2002)
24. Bruno, Gibbons, J.: Scala for generic programmers. In: Proceedings of the ACM SIGPLAN workshop on Generic programming. pp. 25–36. WGP '08, ACM, New York, NY, USA (2008), http://dx.doi.org/10.1145/1411318.1411323
25. Cao, J., Jarvis, S.A., Saini, S., Kerbyson, D.J., Nudd, G.R.: Arms: An agent-based resource management system for grid computing. Sci. Program. 10(2), 135–148 (2002)
26. Dominiak, M., Ganzha, M., Gawinecki, M., Kuranowski, W., Paprzycki, M., Margenov, S., Lirkov, I.: Utilizing agent teams in grid resource brokering. International Transactions on Systems Science and Applications 3(4), 296–306 (2008)
27. Dominiak, M., Kuranowski, W., Gawinecki, M., Ganzha, M., Paprzycki, M.: Utilizing agent teams in grid resource management—preliminary considerations. In: Proc. of the IEEE J. V. Atanasoff Conference. pp. 46–51. IEEE CS Press, Los Alamitos, CA (2006)
28. Drozdowicz, M., Ganzha, M., Kuranowski, W., Paprzycki, M., Alshabani, I., Olejnik, R., Taifour, M., Senobari, M., Lirkov, I.: Software agents in adaj: Load balancing in a distributed environment. In: Todorov, M. (ed.) Applications of Mathematics in Engineering and Economics'34. AIP Conf. Proc., vol. 1067, pp. 527–540. American Institute of Physics, College Park, MD (2008)
29. Drozdowicz, M., Ganzha, M., Paprzycki, M., Olejnik, R., Lirkov, I., Telegin, P., M.Senobari: Parallel, distributed and grid computing for engineering. chap. Ontologies, Agents and the Grid: An Overview, pp. 117–140. Saxe-Coburg Publications, Stirlingshire, UK (2009)
30. Drozdowicz, M., Wasielewska, K., Ganzha, M., Paprzycki, M., Attaui, N., Lirkov, I., Olejnik, R., Petcu, D., Badica, C.: Trends in parallel, distributed, grid and cloud computing for engineering. chap. Ontology for Contract Negotiations in Agent-based Grid Resource Management System. Saxe-Coburg Publications, Stirlingshire, UK (2011)
31. Dynia, M., Korzeniowski, M., Kutyłowski, J.: Competitive maintenance of minimum spanning trees in dynamic graphs. In: Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science. pp. 260–271. SOFSEM '07, Springer-Verlag, Berlin, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-69507-3_21
32. Ericsson, A.: Erlang/otp system documentation (2010), http://www.erlang.org/doc/pdf/otp-system-documentation.pdf
33. Ericsson, A.: Os_mon reference manual. http://www.erlang.org/doc/apps/os_mon/os_mon.pdf (2011)
34. Eriksen, M.: Scaling scala at twitter. In: ACM SIGPLAN Commercial Users of Functional Programming. pp. 8:1–8:1. CUFP '10, ACM, New York, NY, USA (2010), http://doi.acm.org/10.1145/1900160.1900170
35. Eriksson, K., Williams, M., Armstrong, J.: Program development using erlang - programming rules and conventions. http://www.erlang.se/doc/programming_rules.pdf (1996)
36. Foster, I., Jennings, N.R., Kesselman, C.: Brain meets brawn: Why grid and agents need each other. Autonomous Agents and Multiagent Systems, International Joint Conference on 1, 8–15 (2004)
37. Foster, I., Kesselman, C. (eds.): The Grid 2, Second Edition: Blueprint for a New Computing Infrastructure. The Elsevier Series in Grid Computing, Elsevier (2004)

38. Franklin, S., Graesser, A.: Is it an agent, or just a program?: A taxonomy for autonomous agents. pp. 21–35. Springer-Verlag (1996)
39. Galstad, E.: Nagios website. http://www.nagios.org/ (2011)
40. Jennings, N., Wooldridge, M.: Agent technology: foundations, applications, and markets. Springer (1998)
41. Kafura, D., Mukherji, M., Lavender, G.: Act++ 2.0 : A class library for concurrent programming in c++ using actors (1992)
42. Karmani, R.K., Shali, A., Agha, G.: Actor frameworks for the jvm platform: A comparative analysis. In: PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. pp. 11–20. ACM, New York, NY, USA (2009)
43. Kim, W.: Thal: An actor system for efficient and scalable concurrent computing (1997)
44. Kuranowski, W., Ganzha, M., Gawinecki, M., Paprzycki, M., Lirkov, I., Margenov, S.: Forming and managing agent teams acting as resource brokers in the grid—preliminary considerations. International Journal of Computational Intelligence Research 4(1), 9–16 (2008)
45. Kuranowski, W., Ganzha, M., Paprzycki, M., Lirkov, I.: Supervising agent team an agent-based grid resource brokering system—initial solution. In: Xhafa, F., Barolli, L. (eds.) Proceedings of the Conference on Complex, Intelligent and Software Intensive Systems. pp. 321–326. IEEE CS Press, Los Alamitos, CA (2008)
46. Kuranowski, W., Paprzycki, M., Ganzha, M., Gawinecki, M., Lirkov, I., Margenov, S.: Agents as resource brokers in grids—forming agent teams. In: Proceedings of the LSSC Meeting. vol. 4818, pp. 472–480. Springer, Berlin (2007)
47. Letuchy, E.: Erlang at facebook: Chat architecture. Presented at the Erlang Factory 2009, San Francisco, CA (2009)
48. Makki, S., Havas, G.: Distributed algorithms for depth-first search. Information Processing Letters 60(1), 7–12 (1996)
49. Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: Design, implementation and experience. In: Parallel Computing, vol. 30, pp. 817–840. Elsevier (2004)
50. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima Inc, 2 edn. (Jan 2011), http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0981531644
51. Ponci, F., Cristaldi, L., Monti, A., Ottoboni, R.: Multi-agent based power systems monitoring platform: a prototype. In: Power Tech Conference Proceedings, IEEE. vol. 2, p. 5. Bologna, Italy (2003)
52. Ponci, F., Deshmukh, A., Cristaldi, L., Ottoboni, R.: Interface for multi-agent platform systems. In: IEEE-Instrumentation and Measurement Technical Conference. vol. 3, pp. 2226–2230. Ottawa, Canada (2005)
53. Rehak, M., Pechoucek, M., Grill, M., Stiborek, J., Bartos, K., Celeda, P.: Adaptive multiagent system for network traffic monitoring. IEEE Intelligent Systems 24, 16–25 (May 2009)
54. Remy, J., Souza, A., Steger, A.: On an online spanning tree problem in randomly weighted graphs. Combinatorics, Probability and Computing p. 2005 (2005)
55. Ribičre, M., Charlton, P.: Ontology overview. Motorola Labs, Paris (2002), http://www.fipa.org/docs/input/f-in-00045/f-in-00045.pdf
56. Richardson, J.E., Carey, M.J., Schuh, D.T.: The design of the e programming language. ACM Transactions on Programming Languages and Systems 15, 494–534 (1993)

57. Riley, G.: CLIPS website. http://clipsrules.sourceforge.net/ (2011)
58. Santoro, C.: exat: Software agents in erlang. http://www.erlang.org/euc/05/ (2005), december 2011
59. Scheurer, C.A., Scheurer, H.K., Kropf, P.G.: Load balancing driven process migration. Tech. rep., Inst (1995)
60. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for java. In: Proceedings of the 22nd European conference on Object-Oriented Programming. pp. 104–128. ECOOP '08, Springer-Verlag, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-70592-5_6
61. Stefano, A.D., Santoro, C.: Designing Collaborative Agents with eXAT. Enabling Technologies, IEEE International Workshops on pp. 15–20 (2004)
62. Stefano, A.D., Gangemi, F., Santoro, C.: Eresye: an erlang expert system engine. In: Fourth ACM SIGPLAN Erlang Workshop. Tallin, Estony (2005)
63. Stefano, A.D., Santoro, C.: exat: an experimental tool for programming multi-agent systems in erlang. In: AI*IA/Taboo Joint Workshop on Objects and Agents. Villasimius, Italy (2003)
64. Stefano, A.D., Santoro, C.: Building semantic agents in exat. In: WOA. pp. 28–36 (2005)
65. Tismer, C.: Continuations and stackless python. Tech. rep.
66. Tunnell-Jones, A.: dnssd_erlang website. https://github.com/andrewtj/dnssd_erlang
67. Varela, C.A., Agha, G., Wang, W., Desell, T., Maghraoui, K.E., LaPorte, J., Stephens, A.: The SALSA programming language: 1.1.2 release tutorial. Tech. Rep. 07-12, Dept. of Computer Science, R.P.I. (Feb 2007)
68. Vinoski, S.: Concurrency with erlang. IEEE Internet Computing 11(5), 90–93 (2007), http://doi.ieeecomputersociety.org/10.1109/MIC.2007.104
69. Wiger, U.: Four-fold increase in productivity and quality—industrial-strength functional programming in telecom-class products. http://www.erlang.se/publications/Ulf_Wiger.pdf (2001)

Gleb Peregud is an MS student at the Warsaw University of Technology, where he is researching synergies of agent-oriented programming and the actor model, in practical applications. He is an Erlang enthusiast, looking for practical application of actor model and massive concurrency in commercial and academic fields.

Julian Zubek is an MS student at the Warsaw University of Technology in Poland. In his MS thesis, he is developing an experimental Ruby to C compiler. His research interests include also artificial intelligence and programming paradigms (including agent-oriented programming).

Maria Ganzha obtained M.S. and her Ph.D. in Applied Mathematics from the Moscow State University, Moscow, Russia in 1987 and 1991 respectively. Her initial research interests were in the area of differential equations, solving mixed wave equations in space with disappearing obstacles in particular, currently she works in the areas of software engineering, distributed computing and agent systems in particular. She has published more than 100 research papers and is

on editorial boards of 5 journals and a book series and was invited to Program Committees of over 100 conferences.

Marcin Paprzycki (Senior Member of the IEEE, Senior Member of the ACM, Senior Fulbright Lecturer, IEEE CS Distinguished Visitor) has received his M.S. Degree in 1986 from Adam Mickiewicz University in Poznan, Poland, his Ph.D. in 1990 from Southern Methodist University in Dallas, Texas and his Doctor of Science Degree from Bulgarian Academy of Sciences in 2008. His initial research interests were in high performance computing and parallel computing, high performance linear algebra in particular. Over time they evolved toward distributed systems and Internet-based computing; in particular, agent systems. He has published more than 350 research papers and was invited to Program Committees of over 400 international conferences. He is on editorial boards of 14 journals and a book series.

Received: January 8, 2012; Accepted: June 7, 2012

DOI:10.2298/CSIS111223031P

Modeling a Holonic Agent based Solution by Petri Nets*

Carlos Pascal and Doru Panescu

"Gheorghe Asachi" Technical University of Iasi,
Department of Automatic Control and Applied Informatics,
Blvd. Prof. Dr. Doc. Dimitrie Mangeron 27, Iasi 700050, Romania
{cpascal, dorup}@ac.tuiasi.ro

Abstract. One of the key design issues for distributed systems is to find proper planning and coordination mechanisms when knowledge and decision capabilities are spread along the system. This contribution refers holonic manufacturing execution systems and highlights the way a proper modeling method – Petri nets – makes evident certain problems that can appear when agents have to simultaneously treat more goals. According to holonic organization the planning phase is mainly dependent on finding an appropriate resource allocation mechanism. The type of weakness is established by means of the proposed Petri net models and further proved by simulation experiments. A solution to make the holonic scheme avoid a failure in resource allocation is mentioned, too.

Keywords: HMES, Petri nets, multiagent systems, planning, resource allocation.

1. Introduction

Design and implementation of appropriate mechanisms to control manufacturing execution systems are still open problems, and research in Artificial Intelligence (AI) has had an important impact for these subjects [2-4]. With respect to this, an example is the holonic approach, which is considered in this paper; it combines benefits of hierarchical and heterarchical manufacturing control architectures. Holonic Manufacturing Execution Systems (HMESs) are clearly influenced by planning and coordination mechanisms established in AI, primarily in the field of multi-agent systems [3-6]. An HMES regards a control scheme for the shop-floor level of a manufacturing company that is developed around autonomous, co-operative, intelligent entities, named holons. The most often used holonic

* This is a revised and extended version of a paper originally presented at the ICSTCC 2011, Sinaia, Romania [1].

taxonomy is derived from the PROSA reference architecture [7]. This takes into account four types of holons: product, resource, order and staff.

Without giving all details of the proposed holonic scheme construction, this paper aims at showing how an appropriate modeling and analysis method can reveal certain problems for the HMES functioning. Namely, due to the distributed nature of control within HMES, special coordination mechanisms are needed. Though some protocols from multi-agent systems can be considered, these can determine definite drawbacks for manufacturing control systems, requiring an appropriate tuning.

Regarding our paper organization, after presenting some related works, a generic structure of a holon and a basic Petri net model for the holonic decisional component, the holonic agent, are discussed. Then, the Petri nets modeling the inter-holonic communication are presented. About the internal holonic agent operation, this comprises two distinct phases: planning and execution. These are also modeled by Petri nets, which are used to reveal certain drawbacks possible to appear during the planning stage. The theoretical points are illustrated through experiments that were conducted by means of a complex HMES model, obtained as a coloured Petri net. A solution to eliminate the holonic faulty operation is sketched, too.

2. Related Work

While modeling of automated manufacturing systems by different classes of Petri nets is described and commented in a great number of papers and concentrated in some books [8, 9], fewer articles are dedicated to the use of Petri nets for HMES modeling. Nevertheless, the interest for implying Petri nets in holonic and multiagent systems modeling is justified, as they represent a powerful tool for dealing with concurrent processes, which is the case of HMESs. Thus, in [10] some Petri net models of holons were proposed in order to explain the holonic interaction mechanism in PROSA. These models are specific, being provided for certain types of holons (resource, order) and for different kinds of interactions. They highlight some aspects regarding cooperation (synchronization of holons, progress of parallel activities), being restricted to the relation between two holons, mainly between an order and a resource holon. Some benefits are got by the Petri nets application, as these can describe the structure of PROSA holonic components, conducting to logical and temporal analysis of their behavior. For example, by using Petri nets, it was possible to model and evaluate the coupling between a reactive scheduler holon and a holon with special tasks, the on-line manufacturing control holon. In this way an improved adaptability to disturbances was obtained. It results that the proposed Petri net models of holons ensure some guiding points for a PROSA based holonic system design and implementation.

In ADACOR holonic architecture the dynamic behavior of holons is modeled by Petri nets, too [11]. In the same way as in PROSA, models are

developed for each kind of holon and for various holonic behaviors in correspondence with the considered manufacturing environment. The proposed models catch the coordination process based on the Contract Net Protocol (CNP), too. As an advanced possibility, a top-down approach is used in ADACOR. Specifically, some transitions of a Petri net providing a first abstraction can be replaced by more detailed Petri nets in order to assure the description of additional aspects, as needed for the holonic system deployment. Such a successive decomposition in Petri nets and sub-Petri nets allows the incorporation into holonic models of details concerning the production plans and resource allocation.

Another methodology involving Petri nets is described in [12-15]. A formalism on holarchy formation and optimization, as well as on the management of coordination process is facilitated by the use of Petri nets. As a main point, an aggregated Petri net model of a holarchy is proposed, which is augmented with cost functions so that some conditions on holarchy feasibility could be formulated. Enhancements on handling by Petri nets the order constraints and reconfiguration abilities of holonic systems are developed, too. All these aspects are discussed only with regard to the execution phase, without considering details on the planning process and the link between planning and execution.

It thus results that Petri nets were already employed in HMES modeling. Even so, this paper together with the research published in [16, 17], aim at fulfilling some new, distinct goals: to underline all types of events that appear during the operation of any type of holon, to obtain a model of holonic communication so that, in conjunction with the holons' models, the complete HMES model should be obtained and to better reveal the dependence between planning and execution phases.

3. A Petri Net Model of Holonic Agent Operation

As the main operation unit of HMESs, a holon is composed of three components [18] (see Fig. 1): a decisional part in charge with managing the received goals and finding solutions for them; this is materialized under the form of a holonic agent. It has to apply a combination of planning and coordination procedures, as within HMESs a goal is always solved by the cooperation of several holons. The holonic agent's decisions are put into practice by the holon's structural component. For a resource holon this is a proper physical device: a robot or machine tool controller, a PLC commanding a conveyor, etc. In the case of an order or a product holon the structural component becomes a holarchy, which is a temporary construction, namely a group of holons that are conducted by the respective holon in order to solve a goal by cooperation. It is also possible for a resource holon to extend its structural component with a holarchy, when the holon asks the collaboration of other holons. The information is changed between the holonic

agent and the structural component by means of a proper communication interface.

The mechanism for formation of holarchies is based on the CNP, the common coordination method of multi-agent systems [19, 20]. It supposes that a holon not able to solve a goal by itself becomes a manager, asking the cooperation of the other holons, by sending appropriate goals/sub-goals. Those holons able to provide a solution reply with corresponding bids, the best one being selected by manager. The holon that made the respective offer receives a contract from the manager holonic agent, to put into practice the solution it proposed. The common multi-agent CNP has to be adapted and enhanced for a holonic use, in order to obtain a reliable and near to optimum HMES operation [21]. Moreover, a further tuning is needed when the inference mechanism of agents is based on the Belief Desire Intention (BDI) architecture [22, 23]. Certain details regarding the way planning and coordination are supported by holonic agent inference process will be presented in sections 4 and 5.

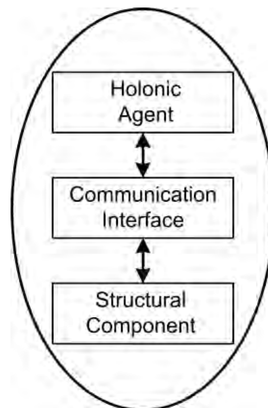


Fig. 1. Generic structure of a holon

HMES operation is both goal and event driven: it has to solve all the goals received at the shop floor level, taking into account events happening in the manufacturing environment (states of various devices, raw parts that are supplied, etc.). It is clear that the whole system operation is determined by the behavior of holons, which is dictated by holonic agents. All these conduct to the necessity of a model for the holonic agent operation and indicate the Petri net formalism as a good choice [12, 24].

The proposed Petri net basic model is a general one (see Fig. 2), being applicable independent of the type of holon, be it an order, product or resource one. The model shows the two main processes that the holonic agent has to pass through as the decisional component, namely planning when it finds a plan for solving a goal, and execution when it applies the decided plan and monitors its carrying out. Besides these, an idle state of

agent is present, which shows its availability, allowing switching between the holonic agent's processes, too.

In this Petri net model the corresponding places are: P_0 for the idle state, P_1 for the planning phase and P_2 represents the execution one. Transitions that determine passing to planning are t_1 representing a goal receiving and t_3 modeling the receipt of a set of bids. This is explained by the way a goal is solved: besides the easy case when the goal can be worked out by a single resource holon through its own physical device activity, the other cases imply cooperation between several holons. Knowing the main steps of CNP, it is clear that the planning stage is interrupted or finalized at two types of transitions: t_2 regards sending of a goal (sub-goal) and t_4 appears when the agent releases an internal contract allowing the start of execution phase or when it issues a bid. Because the model aims at being a general one, it considers the case when the same holonic agent can be contractor and manager, too. That is way the agent can receive and announce goals, while it can also propose bids and issue internal contracts. There are two cases when internal contracts are used: when the holonic agent is only manager, as it can be for an order holon, and when the holonic agent of a resource holon can solve a goal by commanding its structural component. Usual contracts, those sent to other holons, are named external contracts or simply, contracts.

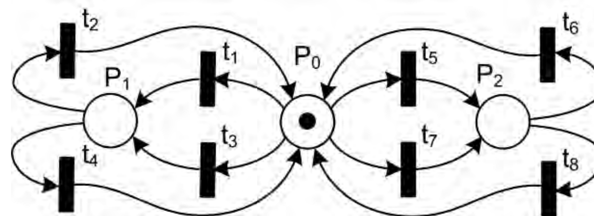


Fig. 2. Basic Petri net model of holonic agent operation

This Petri net model reflects both above mentioned cases, when the agent is able to solve by itself a goal (the corresponding succession is: $P_0t_1P_1t_4P_0$), and when it has to apply for collaboration, this evolution being modeled by the succession: $P_0t_1P_1\{t_2P_0t_3P_1\}t_4P_0$. The notation within curly brackets means that the sequence $t_2P_0t_3P_1$ can appear no time, when there is no goal considered for collaboration, or it can be used one or more times, depending on the number of goals that the agent issued and for which it is waiting cooperation.

Execution stage represents the carrying out of a previous holonic agent's commitment, represented by a bid it made. This is started from the agent's idle state when it receives a contract. The respective event is modeled by the transition t_5. After that, according to the transition t_8 the holonic agent sends the contracts to its sub-contractors or, for a resource holon, commands towards its physical device. Transition t_7 models the feedback from contractors or from the controlled device regarding the ending (with success or failure) of a contract or an action.

When an entire contract is ended, transition t_6 designates the finalization of execution phase. If execution regarded an external contract, according to t_6 the holonic agent provides a feedback towards manager, and this includes the case when the contract could not be accomplished and the respective holon is not able to solve the failure by itself. When the holon can try a further solution to a failed contract or action, during transition t_6 the holonic agent issues an internal (that is not sent by another holonic agent) goal, thus re-starting its planning phase. To conclude, the execution sequence is reflected in the proposed model as follows: $P_0t_5P_2t_8P_0t_7P_2\{t_8P_0t_7P_2\}t_6P_0$. It is to further underline that the proposed model catches (through transitions t_1 and t_3, respectively t_5 and t_7) all the cases when a planning/execution stage is started or resumed for the holonic agent, thus being a general one. The devised Petri net model reflects holonic agent behavior, but the whole HMES activity must be supported by a proper communication mechanism, as shown in the next section.

4. Model of Holonic Agent Communication

When each holonic agent is represented by the Petri net model of Fig. 2, the communication between holons can be modeled according to Fig. 3a and b. It is considered a holonic interaction with three holons, named H_1, H_2 and H_3 (see Fig. 3a). In this example, communication starts when H_1, as manager within the CNP, issues a goal according to the transition $t_{2(H1)}$ (notations of Fig. 2 are respected, too). The result of this transition, as the agent's message, is placed in a buffer represented by the place P_{OUT2} in Fig. 3a. From here, by means of a communication network, it is transmitted to all the possible contractor holonic agents, being inputted into their buffers, marked as the places P_{IN1}. Thus, transition t_1 regarding the presence of a new goal can be fired (in our example, two holons, H_2 and H_3, activate the corresponding transitions: $t_{1(H2)}$ and $t_{1(H3)}$). After that, these contractor holons start their planning process, which is abstracted at the manager holon level by the place $P_{W(H1)}$ (see Fig. 3b).

Each contractor enters in the planning phase only when the holonic agent is freed from other activities. As an example of the proposed model application, for the holon H_3 this means a token is present in the place $P_{0(H3)}$ (see Fig. 3b). Planning phase will have as result a message containing a bid (an output buffer is used), corresponding in our model with the placement of a token in the place P_{OUT4} when the transition t_4 (the one for bid's sending) is fired ($t_{4(H3)}$ in Fig. 3a, b). From this buffer, the communication network transfers the message to the buffer (the place P_{IN3}) of the manager agent; thus, the transition t_3 can be fired for this agent ($t_{3(H1)}$ in our example). In Fig. 3b the places $P_{1->3}$ and $P_{3->1}$ abstract information (goals and bids) transfer from one holon to the other by means of their buffers and the communication network.

A similar mechanism exists for transmission of contracts and feedbacks issued at the end of contracts. Agents' communication buffers should allow several goals/contracts to be received, and thus it can happen that a holonic agent has to treat more goals and/or contracts. Therefore, the issue of treating several goals by the same holon has to be discussed, and also the case when the solutions provided to a set of goals by some holons interfere; these issues are discussed in the next sections.

To understand these problems for a BDI based holonic agent (this kind of agents was used in our approach), it is necessary to be aware of the BDI mechanism operation principle.

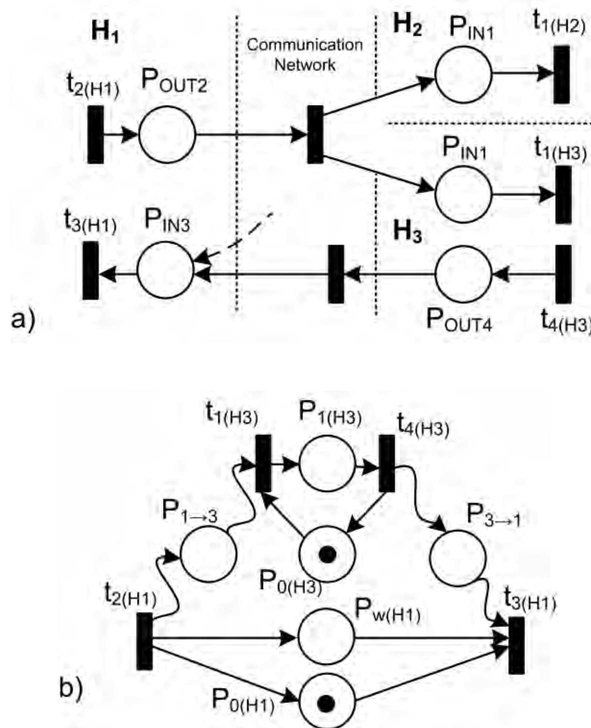


Fig. 3. Model of inter-holonic communication

5. Planning and Execution Processes for a Holonic Agent

As already discussed in section 2, the holonic agent operation covers two distinct phases: planning and execution. The holonic agent functioning starts with planning. The use BDI agents determines a certain influence on the way planning is treated. A specific aspect regards the library of plans that endows

the BDI holonic agents (see, for example, JACK software platform dedicated to implementing BDI agents, which allows the definition and use of plans [23]). For each type of goal the BDI holonic agent must possess a set of plans that it can try when faced with the respective type. In the proposed approach these plans are un-instantiated execution workflows. Here, we name as workflow the whole sequence of actions that a holonic agent uses for an entire execution phase. An un-instantiated execution workflow specifies the sequence of actions that can solve a certain goal, but with the actors to carry out the actions being un-specified. Thus, during the planning process the holonic agent tries to validate an execution workflow by establishing (mostly through the means of CNP) which will be the entities (other holons or its own physical device) to perform the actions of execution workflow. Taking into account all these aspects, the planning cycle for the holonic agent is conducted according to a sequence consisting in the following steps:

Step 1. Choose an execution workflow that could solve the goal. If there is no further choice in the agent's library of plans, then the cycle is ended by declining the goal.

Step 2. Try to validate the selected execution workflow (this means to find resources able to carry out the actions of workflow).

Step 3. If the selected execution workflow has been validated then the planning phase is successfully ended (the corresponding bid/internal contract is sent), else the cycle is restarted with the Step 1.

Both planning and execution can be modeled by Petri nets. As an example, Fig. 4 shows the Petri nets of the execution workflow and the related planning process for solving a goal g (this appears as superior index for the entities of models). The execution workflow contains two actions – a_1 and a_2, for which the holon that received the goal, acting as manager, has to find actors (contractors). These can be viewed as resources and they are modeled by the places ${}^gP_{Rai}$ added to the basic Petri net model (see Fig. 4a). The places ${}^gP_{Rai}$ abstract the state of contractors; for example, the token in ${}^gP_{Ra1}$ marks the commitment made at the moment of bidding, regarding the engagement of achieving the action a_1. The execution workflow is chosen at the beginning of planning process (selection is based on the relevance for a goal and on certain optimum criteria). By that time, no tokens are present in the places ${}^gP_{Rai}$.

The planning process can be modeled according to Fig. 4b, where the same notations of the models of Figs. 2 and 3 are used. The model highlights the two possible outcomes for planning. When the manager holon receives at least a bid for an action of the goal g, the appropriate contractor is allocated and a token will be present in the corresponding place ${}^gP_{Rai}$. If the manager receives only negative bids for a proposed goal, the planning process is abandoned; this case is indicated through the transitions ${}^gt^a_{4(i)}$ in Fig. 4b, and the manager continues its activity in accordance with the above presented

cycle. The planning process of Fig. 4b regards a sequential planning of two actions, and the succession of states and transitions is: $P_0t_1P_{1(1)}t_{2(1)}P_0t_{3(1)}P_{1(2)}t_{2(2)}P_0t_{3(2)}\ P_{1(3)}t_4P_0$, which is in accordance with the general expression presented in section 2 for solving a goal by cooperation. By the places ${}^gP_{W(i)}$ in Fig. 4b, the planning process at the level of contractors is modeled.

After the planning phase, the instantiated execution workflow (tokens are present in the places ${}^gP_{Ra1}$ and ${}^gP_{Ra2}$) is used to entirely guide the execution phase, as the model of Fig. 4a shows. One has to notice how this complies with the Petri model of Fig. 2. During execution no decision points appear, except for the case of an action failure, when planning phase must be restarted. The transition ${}^gt_5$ starts a first execution stage (marked as ${}^gP_{2(1)}$ in

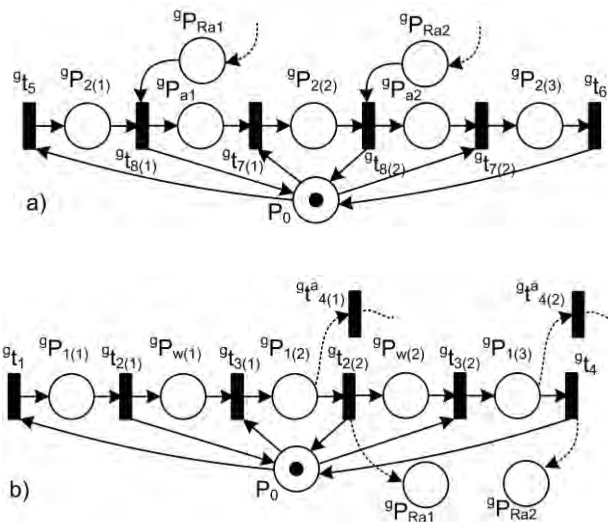


Fig. 4. Petri net models of planning and execution processes for a goal to be solved by cooperation; a) Model of execution workflow; b) Model of planning process

Fig. 4a) as result of contract awarding (see t_5 in Fig. 2). According to the validated workflow, execution continues with awarding a contract towards the chosen sub-contractor – transition ${}^gt_{8(1)}$. The place ${}^gP_{a1}$ reflects the execution of the needed action (a_1) by sub-contractor, being ended at the transition ${}^gt_{7(1)}$, when the holonic agent receives the feedback concerning action end. The place ${}^gP_{2(2)}$ represents a second execution stage, with the same progress as the first one. The place ${}^gP_{a2}$ and transitions ${}^gt_{8(2)}$, ${}^gt_{7(2)}$ have the same meaning as for the previous execution stage, this time regarding the action a_2. The considered example illustrates the general case of a holon being both contractor and manager: it is contractor for the received goal g, and it is manager with respect to finding solutions to achieve the actions a_1 and a_2. One has to note that the Figs. 4a and b are to be regarded together. This means that there is a single place P_0 in the holonic model and the places

${}^{g}P_{Rai}$ are common for planning and execution. All these models allow the HMES analysis, as shown in the next section.

6. Holonic Interaction Analysis; Experimental Results

6.1. Possible drawbacks for the holonic agent planning activity

The planning and execution activities are not continuous: after sending a goal to potential contractors the agent has to wait for bids, after sending a bid the agent waits for manager's answer, after sending a contract the agent has to wait its accomplishing. Thus, it can happen that processes regarding several activities are interleaved for the same holonic agent. Three types of combinations are possible: two planning processes are interleaved, two execution workflows are simultaneously undertaken, or one planning and one execution process are handled by the holonic agent. The last two cases do not need a special attention, because as long as an execution workflow was validated by planning phase it cannot faultily influence another process.

The significant combination is when two planning processes of the same holon are in progress in the same time. Each planning process is started by a distinct goal. If the agent has received two goals and their treatment is interleaved, then the case of Fig. 5 can happen, where the superior indices *1* and *2* refer the two goals (see the index *g* in Fig. 4). The agent works with two execution workflows, which it tries to validate for the two goals. With respect to this, the agent has announced goals (sub-goals) in order to find contractors for the actions of execution workflows (in our example all these actions need other holons to carry them out). The problem is that the actions represented by the places ${}^{1}P_{a1}$ and ${}^{2}P_{a1}$ need the same type of resources, the same condition being true for the places ${}^{1}P_{a2}$ and ${}^{2}P_{a2}$. If the contractors managing the two types of resources happen to make bids for the two execution workflows as shown by the tokens placed in Fig. 5, both planning processes fail, as they cannot transform the corresponding execution workflows into live Petri nets.

The above case happens even the HMES could provide a solution at least for one goal. This occurs if a single planning process is allowed to start and only after its finalization the second planning process begins. This rule is not to be always applied, because it reduces HMES flexibility: it can also be possible for two planning processes to be treated in the same time without deadlock. The solution is to restrict the simultaneous activation for validation (for planning) of more execution workflows that could determine a blockage, as they refer to common resources. This can be implemented by correspondingly marking plans (execution workflows) within the agent's planning library. The proposed Petri net model can help this marking

operation, by using it in a simulation developed before the start of holon's activity within HMES, during which the interaction of plans can be revealed.

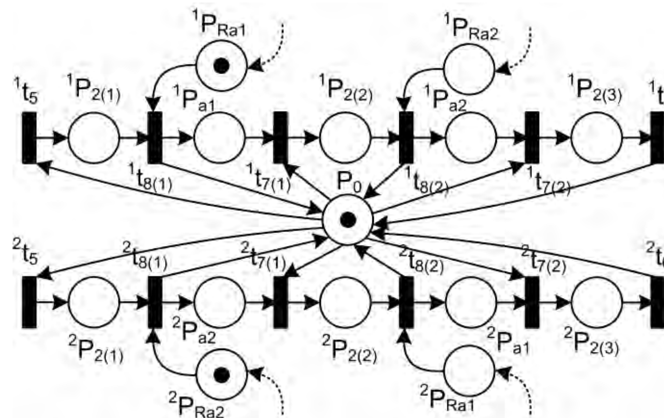


Fig. 5. An example of interaction resulting from the planning process

A similar situation can happen when the two goals that create a conflict are handled by two distinct holonic agents. The case presented in Fig. 5 reflects these circumstances too, except for the fact that two places P_0 will exist, one for each holon. A solution for such situations is beyond the product/order holon possibilities, as they possess local knowledge. There is the need of another component, and this can be a staff holon in the PROSA architecture, which is supposed to take the decision on the management of goals [16]. This type of interaction must be taken into account only between product and order holons, because for resource holons the solution can be obtained at their own level, as they can distinguish goals so that deadlock is avoided [24]. When product/order holons need to announce goals towards resource holons, they should require the acceptance of a dedicated staff holon. This will send the approval only to one requesting holon and keep in a queue the other enquiries that refer to the same type of resources. At the moment of planning finalization, an agent of an order or product holon has to announce the staff holon about this, so that it can consider the next request. In this way two planning processes that refer some common resources are never interleaved and any deadlock is avoided.

In order to conduct significant experiments for proving these theoretical points, a complex model and a simulation environment able to represent an entire HMES were developed [17], by the means of Coloured Petri Nets (CPNs) [25].

6.2. The HMES developed model

The constructed simulation environment appears as a hierarchical CPN, with the highest layer abstracting various entities of the HMES in the form of transitions and places. These are expanded on successive layers, taking into account the proposed basic Petri net model (the one of Fig. 2), but with tokens that can carry different information, according with the formalism of CPNs. An important propriety of CPNs is the way they combine the capabilities of monochrome Petri nets with the support of a high-level programming language, such as the Standard ML [26]. So, it was possible to obtain a highly configurable model-prototype, close to the real HMES implementation.

The top layer of the HMES model is presented in Fig. 6. It comprises one product holon and some resource holons; a staff holon was also introduced in certain experiments. The network necessary to handle the communication between holons is included, too. These entities are represented by transitions that hide the models of lower layers; these materialize by the proposed models for planning and execution (as the ones in Fig. 4). In comparison with the elements presented in Fig. 3, all the input places of a holon were integrated in a single input position. One has to understand that the transitions t_1 - t_8 (see Fig. 2) have attached either an input or an output place (buffer). By using CPNs all the input buffers of a model are represented by a single position, which regards a buffer, marked as *In_k* in Fig. 6. The same is true for the output positions, named *Out_k*. In this way in the CPN model each transition is fired when its attached condition is satisfied, according to the information of its buffer. The transfer of information between holons is achieved through the transition *Communication Network*. The model that this transition is substituting is presented in Fig. 7.

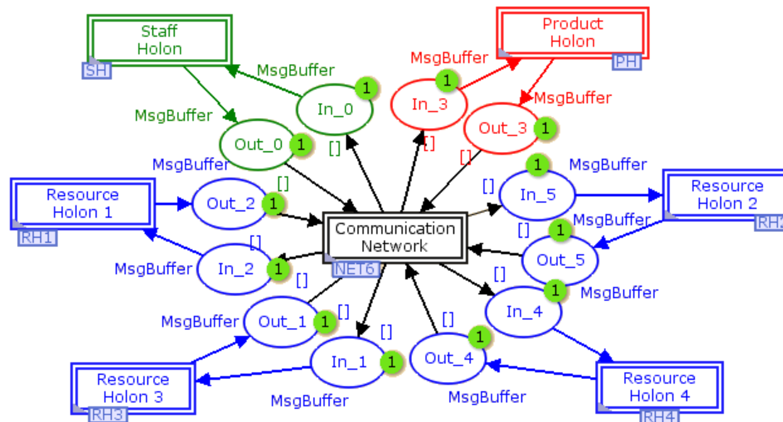


Fig. 6. Top layer of HMES model

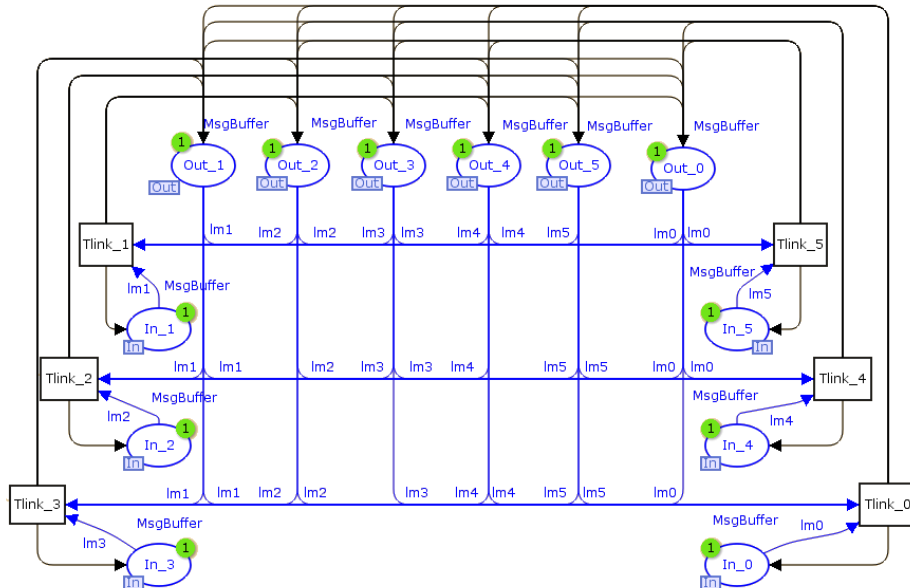


Fig. 7. Communication network model

Thus the part regarding the communication network that was not shown in Fig. 3a is displayed. It is to understand that *Out_k* in Fig. 6 and *In_k* in Fig. 7 are the same positions in the constructed model, according to the use of hierarchical CPNs. The transitions *Tlink_k* in Fig. 7 transfer information at their firing, when a message is present in the output buffer of an entity. Such a buffer is a queue, so that messages are processed in the order they are received. To model the normal operation when the messages are treated in the same order as they were sent, the transitions *Tlink_k* have the greatest salience. In this way, the influence of the communication network on the reachability graph is minimized; this is important because the analysis of HMES performance is obtained by using it.

In principle, a simulation experiment can be used to explore only a finite number of HMES executions. By using the reachability graph of a Petri net a possibility to surpass this difficulty is offered. Indeed, this covers the entire state space of the modeled system, starting from a given initial state [25]. It means the reachability graph can be a powerful tool for assessing the properties of HMESs, and that is way it was chosen. More specific, regarding the problem of establishing the output of holonic agents' planning process, the leaf nodes of reachability graph contain information about the results of resource allocation process. The leaf nodes represent the final or dead markings of the analyzed Petri net, and by their examination the outcome of holonic planning process can be understood.

6.3. Experiments for a holonic system with a product holon

The experiments carried out to prove the above analysis considered the interaction between two planning processes of the same holon (see the explanations concerning Figs. 4b and 5). Five distinct cases were evaluated; the difference between them is given by the availability of resource holons and the treatment of received goals. In cases 1 and 2, the product holon has to face a manufacturing environment with limited resources, allowing only one goal to be fulfilled; this restriction is removed in cases 3 and 4. On the other hand, the product holon tries similar plans in cases 1 and 3, respectively 2 and 4 (it attempts to instantiate similar execution workflows). The final case, the fifth one, regards the HMES operation when a staff holon is also included into the holonic architecture.

The results of the considered experiments are presented in Tables 1-5. These contain three data types. The first two rows give the initial HMES state: the goals received by the product holon, the actions included into the execution workflows that the product holon uses to solve goals and the resource holons able to carry out the respective actions. This information is used to establish the initial marking of the Petri net model. Data on the reachability graph (number of nodes, arcs and dead markings) are presented in the next two rows of the tables, while the following rows summarize the results of planning process. Having two goals to solve, the planning results, reflected in resource allocation, are classified into three classes. These mean the planning process succeeded to instantiate two execution workflows, one of them (with two sub-cases) or none. For each class, the indicated percentage represents the number of final states (dead markings) that correspond to the respective class from the total number of final states.

Table 1. An experiment with two goals and reduced resources

Product Holon's State		Goal 1 (a1-a2)	Goal 2 (a1-a2)
Resource Holons' State		$RH_1 \rightarrow a_1$, $RH_2 \rightarrow a_2$	
Nodes	13525	Arcs	23030
Dead Markings		80	
Goal 1		Goal 2	%
successful		successful	0
successful		unsuccessful	50
unsuccessful		successful	50
unsuccessful		unsuccessful	0

When both goals are solved by the same execution workflow (the succession of actions is a_1, a_2) and two resource holons (RH_1, RH_2) are present in the HMES, only one goal can be fulfilled, as Table 1 shows. In this first experiment no conflict between the planning processes appears. When the HMES context is the same, except for the execution workflows used to

treat the goals, the results of Table 2 are obtained. These indicate that a conflict is possible: there are 3.23% of the total number of dead markings that represent cases when neither of the two goals is solved. It means the product holon fails to accomplish at least one goal, despite the fact that a solution exists. This is the case of Fig. 5, when during the planning process two execution workflows are simultaneously treated and it happens that resource holons make bids for the first action in each workflow; thus, no resource is available to complete a planning process. This situation does not appear in the experiment considered in Table 1, because in that case one of the execution workflows is already abandoned when the product holon does not receive a bid for its first action, and thus the failure situation is avoided.

Table 2. An experiment with two goals and different execution workflows

Product Holon's State		Goal 1 (a1-a2)	Goal 2 (a2-a1)
Resource Holons' State		$RH_1 \rightarrow a_1$, $RH_2 \rightarrow a_2$	
Nodes	55271	Arcs	116622
Dead Markings		186	
Goal 1		**Goal 2**	**%**
successful		successful	0
successful		unsuccessful	48.39
unsuccessful		successful	48.39
unsuccessful		unsuccessful	3.22

Table 3. An experiment with two goals and enough resources

Product Holon's State		Goal 1 (a1-a2)	Goal 2 (a1-a2)
Resource Holons' State		$RH_1 \rightarrow a_1$,$RH_2 \rightarrow a_2$, $RH_3 \rightarrow a_1$ $RH_4 \rightarrow a_2$	
Nodes	63257	Arcs	139552
Dead Markings		164	
Goal 1		**Goal 2**	**%**
successful		successful	39.02
successful		unsuccessful	30.49
unsuccessful		successful	30.49
unsuccessful		unsuccessful	0

The case in Table 3 is similar to the first experiment, but this time there are dead markings representing the fulfillment of both goals, because more resource holons exist in the HMES. When the same experiment is made with different workflows used in planning, the results are worse: the percentage of 23.26% dead markings representing the accomplishment of both goals in Table 4 is less than the value in Table 3. The explanation for the difference between the results of Table 3 and Table 4 is the same as for the cases in Table 1 and Table 2. It is to notice that the number of dead markings is much

higher than the number of planning results, which is explained by the internal mechanism that labels with distinct identifiers messages sent between holonic agents, conducting to different markings in the Petri net. Anyhow, the proper reading of the results obtained with these simulation experiments proves the theoretical points.

Table 4. An experiment with enough resources and different execution workflows

Product Holon's State		Goal 1 (a1-a2)		Goal 2 (a2-a1)
Resource Holons' State		$RH_1 \rightarrow a_1$,$RH_2 \rightarrow a_2$, $RH_3 \rightarrow a_1$ $RH_4 \rightarrow a_2$		
Nodes	174575	Arcs		396964
Dead Markings		324		
Goal 1		Goal 2		%
successful		successful		23.26
successful		unsuccessful		38.37
unsuccessful		successful		38.37
unsuccessful		unsuccessful		0

As already indicated from the theoretical point of view in section 6.1 and practically by the second case, the HMES can fail in solving both received goals when these regard common resources. As mentioned, the introduction of a staff holon can eliminate these drawbacks. Thus, Table 5 presents the result of a further experiment that is conducted for the same case as in Table 2, but with the presence of the staff holon. The interaction diagram in Fig. 8 shows how the product holon communicates with the staff holon when it has to solve two distinct goals. The messages labeled CI (Contractor Information) are those by which the product holon requests from the staff holon the list of available contractors for the plans it has chosen for the two goals. Because the staff holon detects a possible conflict between the contractors of the two plans, it provides a positive answer for one request (the message $CI_{203\text{-}401}$) and the second answer is given only after the finalization of the first planning

Table 5. An experiment with the staff holon

Product Holon's State		Goal 1 (a1-a2)		Goal 2 (a2-a1)
Resource Holons' State		$RH_1 \rightarrow a_1$, $RH_2 \rightarrow a_2$		
Nodes	11243	Arcs		15486
Dead Markings		76		
Goal 1		Goal 2		%
successful		successful		0
successful		unsuccessful		50
unsuccessful		successful		50
unsuccessful		unsuccessful		0

process (the message EI_{401} in Fig. 8). The obtained result (see Table 5) shows that in this approach the drawback of both goals' failure is eliminated. More details and experiments for the operation of an HMES including a staff holon are presented in [16].

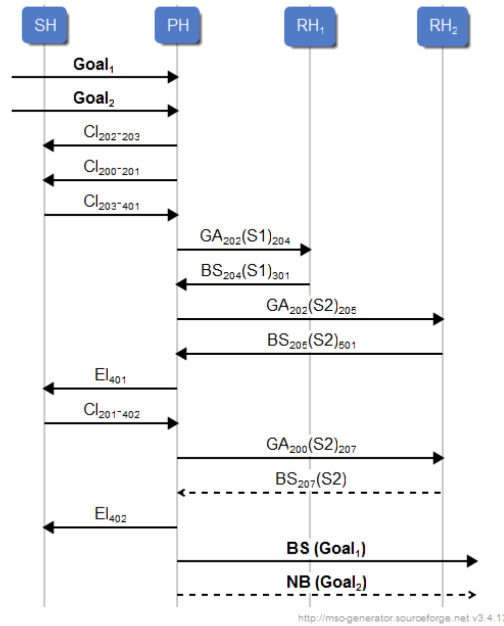


Fig. 8. Interaction diagram for the experiment with the staff holon

7. Conclusion

The work described in this paper addresses modeling and analyzing techniques capable of revealing certain planning and coordination issues of multi-agent systems included in HMESs. The proposed Petri net models describe the internal and external behavior of holons. It is used to construct the reachability graph, which provides important data on the states the HMES can pass through. The analysis has shown the necessity of an appropriate protocol that holonic agents should use for handling of plans, and moreover the need of a centralized component to manage the possible conflicts among planning processes of different holons. These results led us to considering the staff holon as a required entity into an HMES, with the ability to protect the system against potential conflicts. A general planning cycle appropriate for resource, product and order holons was settled, while the staff holon should have a distinct operation, coordinating the other types of holons when

a planning conflict is detected; thus, the whole operation of an HMES is covered.

The planned future work aims at better formalizing and evaluating the BDI mechanism when this is involved in the operation of holonic agents. Thus we are supposed to complete a systematic method for the application of multi-agent systems in holonic manufacturing control.

References

1. Pascal, C., Panescu, D.: On Resource Allocation in a Holonic Manufacturing Execution System. In Proceedings of the 15th International Conference on System Theory, Control, and Computing. Politehnium, Iasi, Romania, 427-432. (2011)
2. Candido, G., Barata, J.: A Multiagent Control System for Shop Floor Assembly. In Proceedings of 3nd International Conference on Industrial Application on Holonic and Multi-Agent Systems. Springer-Verlag, Regensburg, Germany, 293-302. (2007)
3. Jarvis, J., Jarvis, D., Ronnquist, R., Jain, L. (eds.): Holonic Execution: A BDI Approach, Studies in Computational Intelligence, Vol. 106. Springer-Verlag, Berlin, 1-32. (2008)
4. Morel, G., Valckenaers, P., Faure, J. M., Pereira, C., Diedrich, C.: Manufacturing plant control challenges and issues. Control Engineering Practice, Vol. 15, 1321-1331. (2007)
5. Giret, A., Botti, V.: Engineering Holonic Manufacturing Systems. Computers in Industry, Vol. 60, 428-440. (2009)
6. Cheng, F. T., Chang, C. F., Wu, S. L.: Development of holonic manufacturing execution systems. Journal of Intelligent Manufacturing, Vol. 15, 253-267. (2004)
7. Van Brussel, H., Wyns, J., Valckenaers, P., Bongaerts, L., Peeters, P.: Reference architecture for holonic manufacturing systems: PROSA. Computers in Industry, Vol. 37, 255-274. (1998)
8. David, R., Alla, H.: Discrete, continuous, and hybrid Petri nets. Springer Verlag. (2005)
9. Li, Z. W., Zhou, M. C.: Deadlock resolution in automated manufacturing systems: a novel Petri net approach. Springer Verlag. (2009)
10. Bongaerts, L.: Integration of scheduling and control in holonic manufacturing systems. Ph.D. Thesis, Katholieke Universiteit Leuven. (1998)
11. Leitao, P.: An Agile and Adaptive Holonic Architecture for Manufacturing Control. Ph.D. Thesis, Faculty of Engineering of University of Porto. (2004)
12. Hsieh, F. S.: Holarchy formation and optimization in holonic manufacturing systems with contract net. Automatica, Vol. 44, 959-970. (2008)
13. Hsieh, F. S.: Model and control holonic manufacturing systems based on fusion of contract nets and Petri nets. Automatica, Vol. 40, 51-57. (2004)
14. Hsieh, F. S.: Collaborative reconfiguration mechanism for holonic manufacturing systems. Automatica, Vol 45, 2563-2569. (2009)
15. Hsieh, F. S.: Dynamic composition of holonic processes to satisfy timing constraints with minimal costs. Engineering Applications of Artificial Intelligence, Vol. 22, 1117-1126. (2009)
16. Panescu, D., Pascal, C.: HAPBA – a Holonic Adaptive Plan-Based Architecture. In: Borangiu, T., Thomas, A., Trentesaux, D. (eds.): Service orientation in holonic

and multi-agent manufacturing control. Studies in Computational Intelligence, 402, Springer, Berlin, 61-74. (2012)
17. Panescu, D., Pascal, C.: On a holonic adaptive plan-based architecture: planning scheme and holons' life periods, International Journal of Advanced Manufacturing Technology, Springer. (2012)
18. Panescu, D., Sutu, M., Pascal, C.: On the Design and Implementation of Holonic Manufacturing Systems. In Proceedings of the WRI World Congress on Computer Science and Information Engineering. IEEE, Los Angeles, CA, 456-461. (2009)
19. Smith, R. G.: The contract net protocol: High level communication and control in a distributed problem solver. IEEE Transactions on Computers, Vol. C-29, No. 12, 1104-1113. (1980)
20. Hsieh, F. S.: Analysis of contract net in multi-agent systems. Automatica, Vol. 42, 733-740. (2006)
21. Panescu, D., Pascal, C., Sutu, M., Varvara, G.: Collaborative Robotic System Obtained by Combining Planning and Holonic Architecture. In Proceedings of Advanced Technologies for Enhanced Quality of Life. IEEE, Iasi, Romania, 138-143. (2009)
22. Rao, A. S., Georgeff, M. P.: Modeling rational agents within a BDI-architecture. In: Huhns, M. N., Singh, M. P. (eds.): Readings in agents. Morgan Kaufmann, 317-328. (1997)
23. Evertsz, R., Fletcher, M., Jones, R., Jarvis, J., Brusey, J., Dance, S.: Implementing Industrial Multiagent Systems using JACKTM. In: Dastani, M. M., Dix, J., Fallah-Seghrouchni, A. E. (eds.): Programming multi-agent systems. Lecture Notes in Computer Science, Vol. 3067. Springer-Verlag, Berlin Heidelberg New York, 18-48. (2004)
24. Panescu, D., Pascal, C.: Some Issues on Holonic Systems Analysis, Design and Implementation. In Proceedings of the 19th International Workshop on Robotics in Alpe-Adria-Danube Region (RAAD 2010), Budapest, 199-204. (2010)
25. Jensen, K., Kristensen, L.: Coloured Petri Nets: Modeling and Validation of Concurrent Systems. Springer-Verlag, New York. (2009)
26. Harper, R.: Programming in Standard ML. Carnegie Mellon University. (2005)

Carlos Pascal is a young researcher mainly interested in intelligent manufacturing systems. As an early adopter, he tries to bring up a new architecture for manufacturing systems based on autonomous and cooperative entities, by applying several concepts, like holons, Petri nets and BDI agents. With respect to this, in 2012, he received a PhD degree in Systems Engineering at “Gheorghe Asachi” Technical University of Iasi, Romania.

Doru Panescu is professor within the Department of Automatic Control and Applied Informatics, “Gheorghe Asachi” Technical University of Iasi, Romania. He received two master degrees (in Electrical Engineering and in Computer Science), and a PhD degree in Systems Engineering at “Gheorghe Asachi” Technical University of Iasi, Romania. His present teaching and research activities are in the fields of Robotics and Artificial Intelligence, with a focus on intelligent manufacturing systems.

Received: December 23, 2011; Accepted: April 9, 2012.

DOI: 10.2298/CSIS120117029G

Information resource management in an agent-based virtual organization—initial implementation

Maria Ganzha[1,2], Adam Omelczuk[4], Marcin Paprzycki[1,3], and Mateusz Wypysiak[4]

[1] Systems Research Institute Polish Academy of Sciences, Warsaw, Poland
Maria.Ganzha, Marcin.Paprzycki@ibspan.waw.pl
[2] University of Gdansk, Gdansk, Poland
[3] Warsaw Management Academy, Warsaw, Poland
[4] Warsaw University of Technology, Poland
omelczuka,wypysiakm@student.mini.pw.edu.pl

Abstract. In this paper we describe an early stage prototype of a system for information resource management in an agent-based virtual organization. We focus our work on support for human resource adaptability (e.g. knowledge, skills) as a tool for assisting project managers. In the proposed approach, a virtual organization is functionalized in terms of roles played by agents, while organization structure and information flow are represented in terms of agent-agent and agent-human interactions. Finally, all resources (e.g. workers, skills, training resources, etc.) are ontologically demarcated, and information is semantically processed. Discussion of organizational needs is followed by the outline of the system. Finally, basic capabilities of the implemented prototype are illustrated.

Keywords: agent systems, ontologies, virtual organization, resource management, Information Resources, e-learning, adaptability.

1. Introduction

Development of new ICT technologies often influences provisioning of information needed to support workers in an organization. Among recent promising technologies, one can identify software agents [22], as well as ontologies and semantic data processing [13]. In this work, we consider how an organization can combine them to provide workers with the needed *Information Resources* (*IR*) [5]. In the proposed approach a real-world organization is modeled in terms of identified roles played by various entities within it. These roles are then represented by software agents. Furthermore, the organizational hierarchy is mapped into (hierarchical) relationships between agents, while the information

[5] Let us remark that the abbreviation *IR* stands for *Information Resource*, not for *Information Retrieval*, as it is often used elsewhere

flow within an organization is conceptualized in terms of agent-agent (and, possibly, agent-human) message exchanges. In this way we follow, conceptually, the main tenets of the Gaia methodology for agent system development [36]. Furthermore, in the proposed approach, all resources are ontologically demarcated (e.g. workers, articles, courses, books, projects, tasks, etc.), and information is semantically processed. Although in this paper, we focus only on issues involved in delivery of a class of *Information Resources*, presented results can be naturally extended to involve other resources, by modifying the ontology and the communication patterns. Note that, first, the proposed approach is in line with the vision for development of systems combining agents and semantic data processing, outlined in the seminal paper by J. Hendler [20]. Second, it follows the guidelines for agent system development proposed in [27]. There, a well presented argument supports the claim that to achieve progress in use of agent system in the real world, various approaches have to be tried and practically experimented with, to be able to gain the necessary experience.

Let us start with an overview of organizational adaptability that provides the use cases for the system prototype.

1.1. Adaptability in an organization—brief overview

Let us consider a *Virtual Organization* (*VO*; [5, 9, 10, 14, 19, 34]), where workers need to access various *Information Resources* to complete their tasks/projects. Obviously, access to resources should be (a) *adaptive*, matching the specific projects that the workers are involved in at a given moment, and changing with the project (as it evolves), and (b) *personalized*, different workers, depending on their knowledge, experience, and assigned task(s), require access to different resources (and their needs also change over time).

For instance, assume that two workers (W1 and W2) are a part of a team designing and implementing a knowledge management portal, combined with a dedicated mobile application. Here, worker W1 who is designing and implementing the back-end of the system (which is based on the Oracle database) needs different resources than her colleague (W2) preparing a dedicated front-end application geared for mobile operating systems (iOS, Android, and Windows Phone). However, because the new version of the Windows Phone system differs considerably from its previous versions, worker W2 needs additional resources to complete her job (e.g. she may need training modules related to working with the new Windows Phone API). Separately, if a graphics / Web interface specialist (worker W3) is added to the team, to create an application layout for the mobile devices (with much smaller screens), he may also need extra *IR*s to complete his task (e.g. he may need to extend his knowledge about most common screen resolution on mobile devices, including some technical details, like screen contrasts, or just an information about layout usability for the small screens).

In this example, the following situations of interest can be identified:

- Decision, made by the management of the organization, about acceptance / rejection of a project is based on availability of workers with given skills, whose schedule is such that they can participate in the project.
- Some workers, who have been selected to participate in the project, need extra training (access to *IR*'s) to be ready when the project starts (worker W2).
- During the project, some workers need extra training (access to *IR*'s) to improve their skills (worker W3).

Note that, in this work, we use terms *training* and *Information Resources* rather broadly (and somewhat informally). Specifically, by *training* we mean access to various types of *IR*'s that can improve knowledge of the worker, while the term *Information Resource* includes both e-learning modules and publications.

The above listed situations represent cases of, broadly understood, *adaptability* taking place within a system (for more details, see [8, 18]). They can be divided into two orthogonal groups. First, we can observe *institutional* and *individual* adaptability. Here, if workers with appropriate skills (and/or schedule) cannot be found, they have to be hired and/or trained, representing the general case of *institutional adaptability*. Obviously, an organization may select to not to adapt, and reject a possible project. At the same time, worker who is being trained, or provided with an *IR* (e.g. a book) learns, and this represents the case of *individual adaptability* (i.e. her/his skills change). Second, we can distinguish *reactive* and *proactive* adaptability. Here, the *reactive adaptability* means that the *IR* provisioning occurs in response to a specific situation (e.g. during a project a need for extra resource(s) is recognized; as in the case of worker W3, above). At the same time, if we assume that training of worker W2 takes place before the project starts (the organization recognizes the need to train its developers in the use of the new API of the operating system that they will work with), this would represent the case of *proactive adaptability*.

Let us observe that, from the point of view of functionalities planned for the prototype, the proactive and the reactive adaptability resemble each other very closely. As soon as it is established that there exists a gap between the *needed* and the *existing* skills, the system should proceed in exactly the same way. First, the gap has to be assessed, on a case-by-case basis. Second, appropriate *IR*s (assumed to help closing this gap) should be found and delivered to the worker. Therefore, at this stage, we have concentrated our attention on the reactive scenario (leaving the proactive one for the future). Let us therefore describe, in some details, the two use cases that follow from the above general example, and provide foundation for the development (and testing) of the system prototype.

1.2. Use case scenarios

Scenario 1: Project Acceptance Decision. Let us assume that a new project is considered by an organization. Let us also assume that, after an initial analysis, this project is specified through: (a) set of required tasks, (b) set of skills

required by workers to complete these tasks, and (c) time constraints on each task. To be able to decide whether to accept the project, these requirements are matched against human resources available in the organization. Here, the following constraints are considered: (i) "schedule-availability" of workers within the organization, (ii) their individual skills, and (iii) possibility to hire extra workers (if needed).

After the initial analysis, project specification is send to the selected *Project Manager* (*PM*) who will assess whether it is possible to complete it. For each task, the *PM* seeks *Workers* with appropriate skills and schedule. In our current approach, such assessment is based on *PM* $\leftrightarrow$ *Worker* interactions. Here, *Workers* assess their own skills and schedule against the proposed task(s) and send responses to the *PM*. In the case when the *Worker* would need additional knowledge to accomplish a given task, (s)he looks for suitable *IR(s)*. Depending on their availability and time needed to acquire knowledge, a positive or a negative response is send to the *PM*.

Upon receiving responses, the *PM* checks if all tasks can be covered by the available *Workers*. When some tasks cannot be assigned to the *Workers* available in the organization, the *PM* asks the *Human Resource Manager* (*HRM*) to try to hire employees with needed skills and availability. If such *Worker(s)* can be hired, the *HRM* "sends them" to the *PM*. When all tasks within the project have assigned *Workers*, the *PM* accepts the project. Next, she confirms assigned tasks to the selected *Workers*. In the case when one or more tasks are left without assigned *Worker*, the *PM* is forced to reject the project.

Scenario 2: Employee Support. Let us now assume that a team is working on a project. At some moment during that time, one of *Workers* (e.g. W3 in the example above) faces a problem that she never encountered before. Obviously, she starts to look for information to solve this problem. She can look for someone that was dealing with a similar problem (by asking fellow *Workers*—W1 and W2, posting request on mailing list/forum, or search in the Internet). There, she may find solutions that are too complex for her, because they assumed that the person that will use them possessed higher level of knowledge. Other resources can, in turn, cover the issue on such an introductory level that they will be a waste of her time (they will cover topics that she is already familiar with). Thus, the need for personalized support for each *Worker* arises. The system should be able to provide *Workers* with needed *IR(s)* as a reaction to their needs.

Note that, the proactive scenario is very similar to the one described above. The only difference is in the fact that the decision to expand knowledge arises (proactively) on the basis of predicted future needs. Thus the differentiation between them belongs to a different (meta) level of the worker support system.

1.3. Related work

Information management with utilization of agent technology is not a new idea. There are many published works related to this topic (see, for instance, [8–10,

14, 17, 18, 31]), but most of them focus only on theoretical discussion how such systems can benefit from agent technology, and provide the initial design of such system (possibly including some suggestions concerning its implementation). Specifically, they discuss how organization can be transformed to the form that actually can be used in agent systems, with ontologically demarcated *Information Resources*. They consider how to map the needed resources to the missing skills. For instance, in [30] authors' discuss various methods of ontological matchmaking (understood as matching instances within a single ontology) an propose their own approach. However, this novel approach has not been tested in a realistic application.

In [25] authors present a platform that was designed during the EU project "Platform for Organizationally Mobile Public Employees" (project *Pellucid*). Main goal of this project was to create adaptable platform for assisting organizationally mobile employees. Such platform was to improve organization effectiveness and efficiency by formalizing, recording, and storing information about experience and knowledge, and allowing easy access to such data in a mobile environment. The proposed system was to combine software agents and semantic data processing. Unfortunately, as in case of so many other similar projects, after the end of EU-funding the promising research has stopped.

Another approach worthy mentioning was presented in [24]. In his thesis, author discuses use of ontology based knowledge representation in multi-agent systems. Author focuses on the issue how to create stronger connection between those two with utilization of the semantic web. Again, the research does not seem to be continued after completion of the thesis.

In Poland, between 2005 and 2007, there existed (sponsored by the Polish government) project called WKUP (eng. Virtual Public Service Consultant; [6]). The goal of this project was to create interactive, personalized platform to support citizens in finding assistance how to deal with tasks required by various administrative processes. The support was to be based on ontological representation of knowledge about the Polish legal system. Specifically, the WKUP system was to be able to understand questions in natural language, use ontologies describing domain of public administration, and be able to cooperate with a semantic registry of public services based on the UDDI (Universal Description Discovery and Integration) specification. Even though the initial prototype has been developed around 2007, the project has been later abandoned [12].

Overall, while many publications consider the idea of combining software agents and semantic data processing to support workers in a *Virtual Organization*, in most cases only initial stages of system design have been completed, and promising prototypes have been abandoned (e.g. due to the lack of continued funding).

2. System overview

As stated above, the aim of our work was to develop a prototype system supporting autonomous *IR* provisioning in a *Virtual Organization*. This system should

automatically detect *Worker* needs, and attempt at provisioning targeted *IR*s. One of key assumptions is that the system will be agent-based, and will use semantic data processing. Following the ideas discussed in [31, 17], we have decided that each *Worker* in the organization will be supported by her/his personal *Worker Agent*, which will represent her/his interests in the system. The *Worker Agent* is a form of a *Personal Agent*, idea of which was proposed for the first time by P. Maes in [26]. Furthermore, a number of auxiliary, agents will be added to the system. These agents will facilitate roles that either can be fulfilled autonomously (without need for human intervention), or will emulate roles that, in an actual organization, are typically fulfilled by humans supported by their *Worker Agent*s. Based on these assumptions, in the context of the above presented example, and the two use case scenarios (that define the scope of our initial work), in Fig. 1 we present the AML use case of the system (for more details about AML, see [11]).

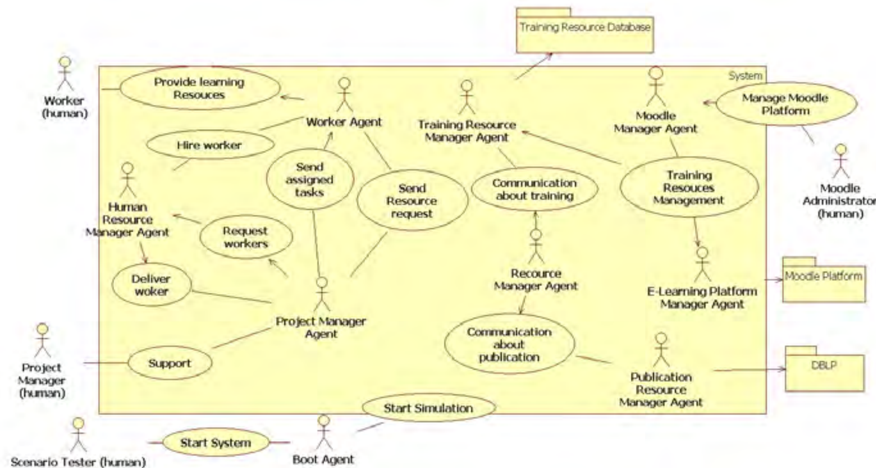


Fig. 1. AML Use case—functionality of the system

Let us now describe most important features of the proposed system in some detail.

2.1. Agents in the system

Let us start with a brief discussion of agents and their roles (see, Fig. 1). For the initial system prototype, supporting use cases described in section 1.2, we have designed and implemented the following agents:

- *Personal Agent* (*PA*); an artificial meta-name for a group of agents controlled by human user. An instance of such agent is provided to each *Worker*

in the *VO*. The *PA* supports: *Workers* in a form of *Worker Agent* (*WA*); *Project Managers*, as the *Project Manager Agents*; and *E-learning System Administrators* as the *Moodle Administrator Agents*.

– *Resource Agent* (*RA*); auxiliary autonomous agents that can play the following roles in the system:
  - *Human Resource Manager Agent* (*HMRA*), which emulates the situation in which a human *HRM* would be supported by an extended version of the *WA* (prepared to support the *HRM* activities); in our system, due to its limited scope, this role is emulated (completed autonomously).
  - *Publication Resources Manager Agent PRMA*), which is interfacing with the Digital Bibliography and Library Project (DBLP; [32]),
  - *Training Resource Manager Agent* (*TRMA*), which is responsible for providing the system-side interface to the e-learning platform,
  - *E-learning Platform Manager Agent* (*EPMA*), which provides the platform-side interface to the e-learning platform (here, it is assumed that multiple e-learning platforms are located "somewhere within the Internet" and thus both sides—system and platform—have to be represented by separate agents – rather than having a single platform-side agent that would be contacted directly from the system).

Let us summarize agents and their roles in the form of the AML Role Diagram, in Fig. 2.

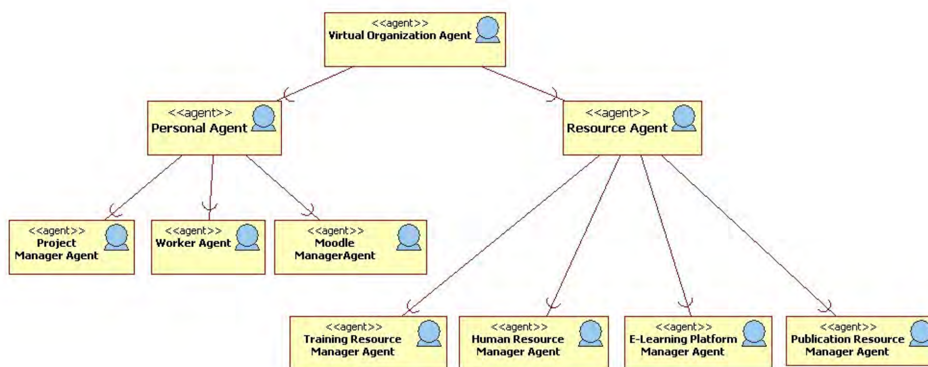


Fig. 2. Agent's roles hierarchy

2.2. Information Resources

Let us now consider the *Information Resources* that are needed to complete the use case scenarios described in section 1.2. In the current system prototype we have introduced two types of resources: (i) *training resources*, and

(ii) *publication resources*. The *training resources* comprise online courses supplied for the *Worker* training. The *publication resources* are publications that provide the needed information.

As stated above, in addition to software agents, we will use semantic data processing. Therefore, both types of *IR*s have been demarcated using a simplistic ontology, which was designed to support selected functionalities of the prototype. Let us now look in some detail into the use of ontology in our system.

2.3. Ontology in the system

When designing our prototype we were faced with decisions concerning development and use of ontologies. For instance, we could have used the existing ontologies, e.g. the Dublin Core [1] for demarcation of publication resources, and adapt the *VO ontology* found in [33, 29, 23] to describe the organization. However, first, this would still leave us us with 0the need to develop an ontology of training resources. Second, combining the Dublin Core, the VO ontology and the training resource ontology is a research task in its own right, likely resulting in a rather large and complex combined ontology. Third, we were not able to locate a mainstream e-learning platform that would be ontology enabled. Overall, focusing on ontologies, would postpone implementation of the system, which would contradict our main goal—development and experimentation with a working prototype (see, also [27]). Therefore, we have decided to develop our own simplistic ontology and use it in the prototype. In Fig. 3 we present the overall structure of this ontology.

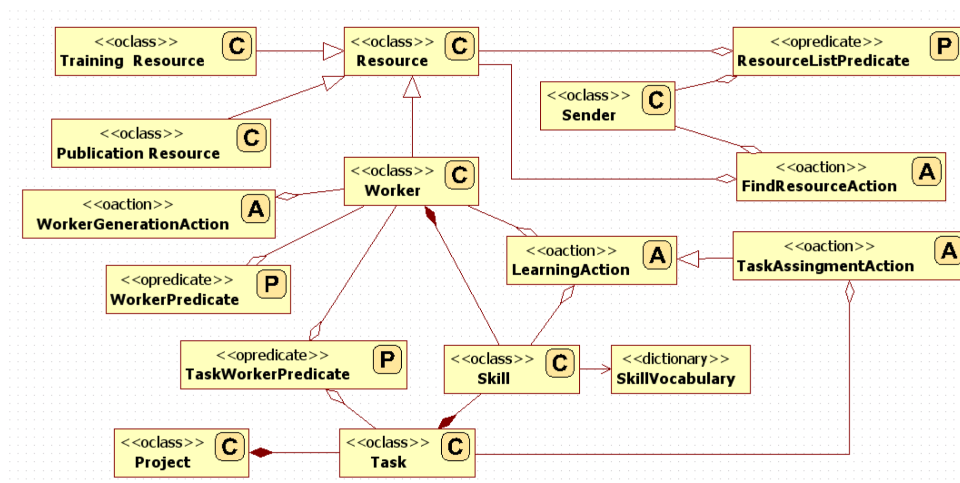


Fig. 3. Ontology design (C - Concept, P - Predicate, A - Action)

Here, the central concept is the *Resource*. For the prototype, we have decided to focus on the field of information technology, as conceptualized by the Association for Computing Machinery Computing Classification System [7]. The *Skill* concept consists of two parts. The name of the discipline, and a value from the interval $(0, 100)$, which indicates the skill-level. Note that the *Skill* class is the only one used in both the *Resource* and the *Worker* modules (see, section 3).

The *Resource* concept describes the *Information Resources* as well as *Human Resource*. It is used mostly in communication between the *Worker Agents* and the *Resource Module*. It contains the main *Resource* class, the implementation of the *Concept* interface, and three child classes: the *Training Resource*, the *Publication Resource* and the *Worker*. The *Worker* class stores the information about *Worker's* skills (implemented as the *Skill concept*). The *Skill* concept establishes connection between the *Worker* and the *Task* concepts, as well as between the *Worker* concept and the *LearningAcion* action.

In case of concepts *TrainingInformation* and *PublicationResource* these classes store specific information about those *IR*s, like author, title and publisher, for the *publication resources*, or names, localization and cost for participating in the course, for the *Training Resources*. In this structure, thanks to the inheritance, subclasses could use the *Resource* class in all messages for all *IR*s. Note that the only place where there is a need to distinguish between the different *IR*s is when the *Worker Agent* displays found resources to the *Worker*. In other words, within the system both forms of *Information Resources* are processed exactly in the same way, only in the user interface they are distinguished.

There exists also an artificial class called *Sender*. It was created because, despite the fact that the FIPA message envelope contains information about the sender of the message, when the request for resources from the *Worker Agent* is forwarded (by the *Resource Manager Agent*) to the *Training Resource Manager Agent* (or to the *Publication Resource Manager Agent*), the sender of the request has to be preserved in order to be able to return the found resources to the proper *Worker Agent*. Such requests are implemented using the *AgentAction* class *FindResourceAction*. This class contains information about the sender, and a list of topics that should be covered by the returned resources. This list is stored as a list of *Resource objects*, with the skill field assigned. As an answer to the request, represented by the *FindResourcesAction*, the *Resource Module* sends the *ResourceListPredicate* class, which implements the *Predicate* interface. This class is used as a facade for the list of found *Resources*. This is done, because the JADE framework does not allow sending the *Concept* implementations directly. They have to be wrapped in an *Action* or a *Predicate*.

The second part of the ontology is used to describe the *Worker*. This module focuses on the project, by describing *Workers*, tasks within project, skills possessed by the *Workers* and required for tasks, the project itself and actions that can be performed. Those objects are used mostly in communication between the *Worker Agent* and the *Project Manager Agent*. The two main, separated, concepts are the *Project* and the *Worker* classes. The first describes the actual project, in terms of name, list of tasks (understood as a “to do” list;

e.g. creating a database, or designing application layout for a mobile device), available time (here, we have decided to use the standard man-hours), and the information if this project is to be considered to be within the scope of a proactive or a reactive scenario (i.e. if it is a project for the future, or if it is a project that the organization has committed to). The *Worker* class stores basic information about the employee, like: first name, last name, list of skills of the represented person. This section contains also two implemented *Predicates*, called the *WorkerPrediacte* and the *TaskWorkerPredicate*. The first predicate is used in the transport of the *Worker* concept (for the same reasons as mentioned earlier for the *ResourceListPredicate*). The *TaskWorkerPredicate* has a more complex job to do. It contains the *Task* and the *Worker* concepts, to be used in the communication process, and parameters defining how well skills of a given *Worker* match skills required for completing a specific *Task*. Here, we use simple metrics, based on the assessment of how many skills were on the exactly right level, how many above, and how many below the required skill level (for a given task). Additionally, the system stores the number of skill levels that are above the requirements, and number of those skills that did not fulfill the requirements.

To send requests, three implementations of the *Action* interface are used. The *WorkerGenerationAction* is used by the *Project Manager Agent*, when it decides that a new employee has to be hired. It contains the *Task* concept, because such decision is made when some of the tasks, in the specific project, are not assigned (and cannot be assigned to the currently employed workers). Thus, a new *Worker* should be hired to complete this task. The remaining two actions are connected by the base-child class relation. The base class, called the *LearningAction*, stores the list of *Skills* levels of which the recipient (*Worker Agent*) must improve, and/or list of brand new *Skill*(s) to be acquired. The child class is called *TaskAssignmentAction*, and stores the *Task* that is assigned to this *Worker Agent*. We decided to create such relation, because, sometimes, when the *Project Manager Agent* assigns a *Task* to a particular *Worker Agent*, a given *Worker* must learn something new or increase knowledge in one of already possessed *Skills*, while at other times learning activities are required without assigning any task.

3. Implementation details

As a result of the technical requirements analysis, completed in [28], we have selected the following technologies to implement the initial system prototype.

- All agents within the system are written in Java using the JADE framework [2], with addition of the log4j framework [3], used for the error logging purpose. During the actual implementation process, we used Java version 1.6.0_22. The runtime environment was provided by the standard Oracle Java Virtual Machine.
- For storing data used to describe the *training resources*, the MySQL database engine was used.

- As the e-learning platform we have selected the Moodle Platform [4]. It is an open source Learning Management System. While Moodle is written in PHP we decided not to connect with it through its web interface, but created a dedicated agent connected directly to the Moodle database. Note that this design decision follows the basic principles of agent system design, where software agents provide the basic abstraction for design modularity and component encapsulation (see, also [21]).
- The MySQL server and the Apache server, required to host the Moodle platform, were provided by the XAMPP version 1.7.3.
- Information retrieved from the *Training Resource Database* and the DBLP, for further use by different agents, was described using ontology created using the Ontology Bean convention (see, section 2.3 for more details).

Lest us make a comment concerning the last decision. One of the weaknesses of the JADE agent platform is its limited ability to deal with ontologies. As a matter of fact, at the time of working on our prototype the only reasonable way to combine JADE agents and ontologies was by representing ontologies as Java classes. Therefore, we have decided to implement our simplistic ontology directly into Java classes using the Ontology Beans (without the OWL demarcation). It is only now that we have developed a JADE add-on that allows agents communicate using OWL snippets (see, [35] for more details). This will allow us to actually use OWL ontologies directly, in the next release of the system.

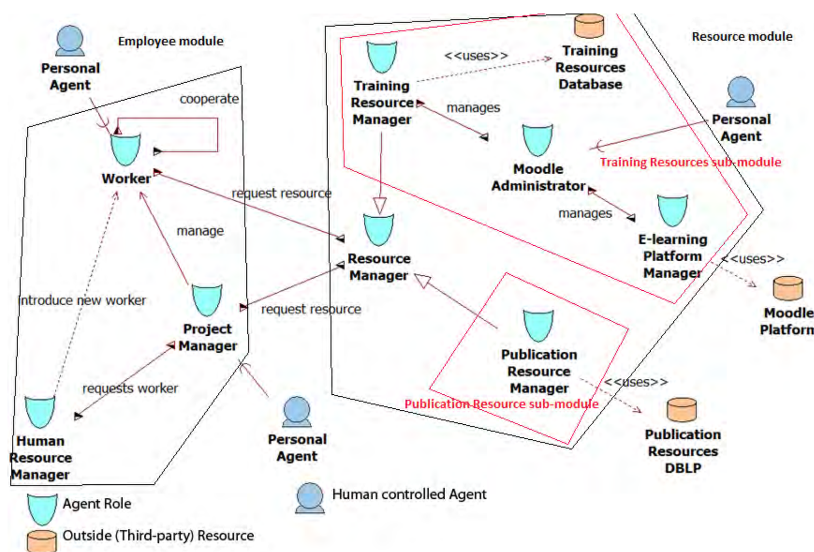


Fig. 4. System modules design

Let us now look in more details into the overall structure of the system and the implemented modules, which have been represented in Fig. 4.

3.1. Resource Module

Let us start with the *Resource Module*. It deals with all aspects of searching and obtaining different *Information Resources*, including managing databases, and e-learning platforms. It was divided into the following sub-modules: the *Publication Resource* module, the *Training Resource* module, and the *Connector*.

Here, the *Connector* consists of the *Resource Manager Agent*, which connects the *Resource Module* and the *Employee Module*. On the setup it starts two cyclic behaviors that are responsible for periodically refreshing the list of available *Publication/Training Resource Manager Agents*. This is achieved by asking the JADE Directory Facilitator (*DF*) for the list of such agents. All requests from the *Worker Agent* (to search for the *IR*s), received by the *Resource Manager Agent*, are copied into identical messages send to the *Training Resource* module and the *Publication Resource* module. Each of these messages contains added information about the original sender. Next, the *Resource Manager Agent* awaits responses from all specialized *Resource Managers* and merges answers received from them into a single *ResourceList* object. Here the inheritance relation between the *Resource* and the *PublicationResource*, as well as the *TrainingResource*, is used. The resulting list is forwarded to the proper *Worker Agent* (identified through the *Sender* object included in the message). In this way, the *Employee Module* is not (and does not have to be) aware about possible sources of *Information Resources* (which, therefore, can be dynamically added/removed, with only localized changes in the systems).

The *Training Resource* module is responsible for managing content of the e-learning platform—in our case the Moodle Platform—and searching for courses matching the requested topics. It is composed of three agents: (i) *E-learning Platform Manager Agent*, (ii) *Training Resource Manager Agent* and, human controlled, (iii) *Moodle Administrator Agent*. The *E-learning Platform Manager Agent* is directly connected to the database of the Moodle Platform and is responsible for translating requests received from the *Moodle Administrator Agent* to the SQL queries, and executing them. On the startup, in addition of the standard setup, this agent establishes connection with the database (using data retrieved from a file). Such solution allows developers to create different configuration for different Moodle Platforms by starting multiple instances of the same agent. Note that, during system development, we used the standard Java JDBC method to interface the database.

The *Moodle Administrator Agent* is a user-driven agent. It has two cyclic behaviors. First, it is responsible for updating the list of available Moodle Platforms (by periodically asking the JADE Directory Facilitator about agents that advertise themselves as agents interfacing the Moodle Platforms). The second cyclic behaviour looks for existing *Training Resources Manager Agents*. The remaining behaviors are triggered by user actions and deal with: (1) creating a new Moodle User Account, (2) selecting a Moodle User Account, (3) updating a

Moodle User Account, (4) deleting a Moodle User Account, (5) creating a new Moodle Course, (6) selecting a Moodle Course, (7) updating a Moodle Course, and (8) deleting a Moodle Course.

The last agent of the *Training Resource* module is the *Training Resource Manager Agent*. On setup, it connects to an auxiliary course database, created to speed-up the searching process and to lower the cost of communication between the system and the Moodle Platform. This course database allows also for a single *Training Resources Manager Agent* to handle more than one e-learning platform and/or different types of them. It contains description sof available courses in a format corresponding to the ontology of training resources (see, Fig. 5, for more details).

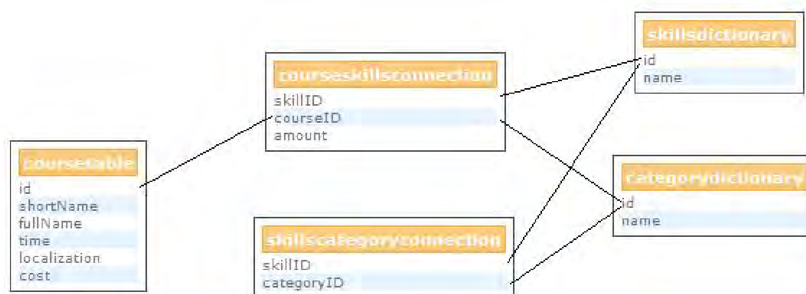


Fig. 5. Training Resource Database

When searching for the *Training Resources* concerning a given subject (identified by the *Skill* name) the *Training Resource Manager Agent* looks in the *Skill-Dictionary* table for the ID of such *Skill* and then creates a query that will return all *IR* that cover the selected topic(s) and has the *Skill* level above the received requirement. This means that, after attending a specific e-course, worker should obtain the corresponding knowledge. All courses found as a result of the query, are send back to the *Resource Manager* (without any knowledge to whom they will be forwarded to). Next, the *Resource Manager Agent* forwards them to the requesting *Worker Agent*.

In the system prototype, the second source of the *Information Resources* is the *Publication Resource* module. Currently, it contains only one agent, which is responsible for communicating with the DBLP database. However, in the future, it is planned (and it is a natural extension of the proposed system design) to add additional agents that will facilitate information from other / additional sources. Here, the keywords (*Skill* names) received from the *Worker Agent* are transformed into an appropriate query-string, and executed on the DBLP interface. The received response is parsed from the HTML code, and filtered using the *Skill level* parameter (to select these resources that are actually needed). Next,

it is stored in the *ResourceList*. If required, the *ResourceList* may be trimmed to the requested length (note that the query to the DBLP—as well as to any other data source—may return hundreds of responses). Next, the *ResourceList* is send back to the requesting *Worker Agent*.

3.2. Employee module

The second specialized module is the *Employee Module*, which is composed of three agents: (a) *Project Manager Agent*, (b) *Worker Agent*, and (c) *Human Resource Manager Agent*. The last agent is completely autonomous, while the first two require human interactions (at least in the current design of the system).

The *Human Resource Manager Agent* emulates functions of a human *HRM*, supported by an appropriate *Worker Agent*. Its only role is to fetch (from a file) a list of "not hired workers" and, when requested, search for a workers with needed skills. This is to emulate situation when the *PM* finds out that she needs to hire extra workers to complete the project and requests help from the *HRM*.

The *Worker Agent* supports the *Worker* and represents her/him in the system. It stores the *Worker* profile and, if appropriate, the *Task* objects representing tasks currently assigned to its owner. This is the only agent (in this module) that communicates with the *Resource Manager Agent*, to ask for *Information Resources*. This behavior can be triggered manually by the *Worker*, or autonomously during the project management process. It is facilitated by creating a list of needed *Skills*. This list is send to one, or more, *Resource Manager Agents*. On demand, this agent can send the list of *Skills* of its *Worker* (owner), to the *Project Manager Agent*, so that it can be used in the task-worker matching process (using data as actual as possible). Furthermore, the *WA* responds to the queries from the *Project Manager Agent*, asking which tasks, stored in the *Project* predicate object, can be completed by the *Worker*. Second, it accepts the information that its *Worker* was assigned a task, and appropriately modifies her/his profile (availability schedule).

Note that, in the case when a "not hired" employee receives a message containing an information that a task was assigned to it, it changes its registration type, in the JADE Directory Facilitator, from "not hired" to "hired."

The last agent in this module is the *Project Manager Agent*. As a matter of fact, this is a *PA* extended with the project management related capabilities (see, also [15, 17, 16]). This agent is responsible, first, for communication between the "outside world" and the system. Here, it accepts the *Project* objects, representing projects to be dealt with. It also communicates with the *Human Resource Manager Agent* and the *Worker Agents* (representing potential project workers). On the startup, it initiates two cyclic behaviors responsible for updating the list of available *Human Resource Manager Agents*, and the list of hired *Worker Agents*. This is done by cyclically asking the JADE *DF* for the current list of such agents. Currently, the *Project Manager Agent* handles project management from the moment of receiving a new project, to sending assigned tasks to the selected *Worker Agents*, or to declining its completion (due to the lack of resources / workers).

3.3. Matchmaking in the system

While in [30] a complex method for ontological matchmaking was proposed, here we decided for a more simplistic approach. As stated above (sections 2.3, and 3), we have deliberately elected to apply a very simple, Java class-based ontology; rather than a full blown OWL-based one (similar to that discussed in [17]). This being the case, at current stage of system design, applying complex ontological matchmaking would make no sense.

The implemented matchmaking aims at minimizing the gap between the needed and the existing skills. It assigns *Workers* to *Tasks* by picking an employee with the skill-level at least as high as that required by the task. The assessment of the skill-level is done independently by each *Worker Agent*, and is coordinated by the *Project Manager Agent*. The *Workers* that do not match the needed *Skill* set, can be (self)assigned to the training activities that should remove the gap and allow them to complete the task. In the process of establishing if the *Worker* can complete the task, the current schedule of the *Worker*, the time needed to accomplish the task, and (if necessary) the time of completion of the needed training activities are jointly taken into account. Matchmaking is also used by the *Human Resource Manager Agent* to find the best *Workers* to be hired, in the case that a project needs them. Agents that use the matchmaking engine include: the *Project Manager Agent*, the *Worker Agent*, and the *Human Resource Manager Agent*.

4. Experimental evaluation

Let us now illustrate work of our prototype, using a sample scenario that matches the first use case, discussed in section 1.2. Let us assume that the execution environment contains the followings units: single instances of *Project Manager Agent*, *Resource Manager Agent*, *Publication Resources Manager Agent*, *Training Resource Manager Agent*, *Human Resource Manager Agent*, and five *Worker Agents*, three employees (WA1, WA2, WA3), with the following skills: Oracle back-end developer, front-end mobile platforms developer, and graphics / web designer. Furthermore, we have two workers in the pool managed by the *Human Resource Manager Agent* (WA4 and WA5), both graphics / web designers. Here, recall that the *Human Resource Manager Agent* emulates the process of hiring additional employees. The *Publication Resources Manager Agent* and the *Training Resource Manager Agent* are connected to their respective sources of *Information Resources* (DBLP and Moodle). Furthermore, the *Training Resource Manager Agent* is connected with the *Training Resource Database* storing information about current course offerings. The sample project under consideration matches the example presented in section 1.1. It consists of four tasks: T1—implementation of a back-end that uses an Oracle database, T2—implementation of a mobile application for the iOS, Android and Windows Phone, T3—performing functional tests, and T4 - creation of the layout of the graphic interface for the new application.

Let us now assume that the WA1 and the WA2 have skills connected with the task T1 (and that the WA1 is a better match—more skills satisfy task requirements); however, the WA2 also can be assigned to complete task T2. Furthermore, worker WA3 may execute either task T3 or task T4, but with a skills preference favoring execution of task T3. Finally, both workers WA4 and WA5 have skills corresponding to tasks T3 and T4 (however, WA5 is a better match for either one of them). The fact that a worker has skills connected to the task does not mean that all of the required skills are present. This can also mean that some of needed skills are not at the required level, or that some of them are missing (see, also Fig. 6). We will now discuss what is happening when the project is introduced to the system, from the "point of view" of each major agent / module.

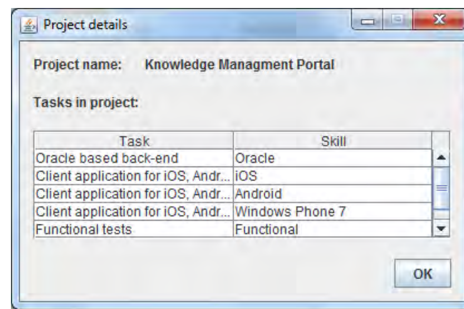


Fig. 6. Skills connected to tasks—without skills levels

4.1. *Project Manager Agent*

From the perspective of the *Project Manager Agent*, the scenario begins when it receives a message containing an ontologically described project to manage. As its first activity, the *Project Manager Agent* refreshes the list of workers (*Worker Agents*) available in the organization and, in the considered example, finds out that these are WA1, WA2 and WA3. Now it can send a call for proposals to all of them, seeking task executors. Upon reception of responses (let us omit the case of non-responsive *WA*s) the *Project Manager Agent* matches proposals to tasks, to select the best *Worker Agent* for each of them. In our situation, after the matching is completed, the T1 will be preliminarily assigned to the WA1, T2 to the WA2 and T3 to the WA3; while task T4 will be without a worker assigned to it. Now, the *Project Manager Agent* will send a message to the *Human Resource Manager Agent*, with a request to hire a new employee that will match the requirements of task T4. After obtaining information that worker WA5 should be hired, the *Project Manager Agent* can change the preliminary assignment of tasks to the final one (by re-evaluating the available *Workers* vis-a-vis the required tasks), and send to appropriate *Worker Agents* their assigned

tasks—T1 to WA1, T2 to WA2, T3 to WA5 and T4 to WA3. In this moment this scenario is completed for the *Project Manager Agent*; see, also Fig. 4.1.

Task	Worker
Oracle based back-end	Ava Roberts
Client application for iOS, And...	Emily Phillips
Functional tests	Landon Phillips
Interface creation	Landon Garden

Fig. 7. Task-worker matching

4.2. *Worker Agent*

Now let's consider the *Worker Agent* perspective (here, we will use the WA2 as the example). Since all *Worker Agents* execute the same behaviors, the specific case of W2 represents the general *Worker Agent* behavior. Here, our discussion will need a more detailed set of assumptions. Let us, therefore, assume that the WA2 has the following skills (called *Worker Skills* WS; see, Fig. 8): WS1, iOS skill at level 53, WS2, Android skill at level 61, WS3, Oracle skill at level 67. The WA2 can, of course, have also other skills, but in this example they will by irrelevant, so we omit them. The specification of the T1 states that the needed skills are (*Task Skills* TS, where Sn in WSn and TmSn denote the same skill n): T1S3 at level 70; and T2: T2S1 at level 69, T2S2 at level 60 and T2S4, Windows Phone skill, at level 34.

For each *Worker Agent* this scenario starts when it receives a message with a request to perform the individual task matching process. This message can be sent by the *Project Manager Agent* or the *Human Resource Manager Agent*. However, this is inconsequential for the actions undertaken by the *Worker Agent*. Each task will be checked against the *Worker* skills. In this stage the *Worker Agent* can observe that it matches all skills needed for task T1, but it needs to improve one skill by a small amount ($T1S3 - WS3 = -3$). In the case of task T2, one new skill has to be acquired (S4) and one has to be improved by a significant amount (16). To check if it is possible to acquire/improve skills, the *Worker Agent* will communicate with the *Resource Manager Agent* seeking resources for topics related to S1, S3 and S4. After obtaining the information about appropriate resources, the *Worker Agent* will create suggestions, which tasks can be assigned to it. In our case, both the T1 (see, Fig. 9) and the T2 (see, Fig. 10) will be acceptable. Now, the *Worker Agent* replies to the *Project Manager Agent* with its suggestions, and waits for further requests. This scenario can end with

no task assigned to the worker (here, WA4), or with a reception of a message containing assigned task(s), and (if required) list of found *IR*s (see, Fig. 11).

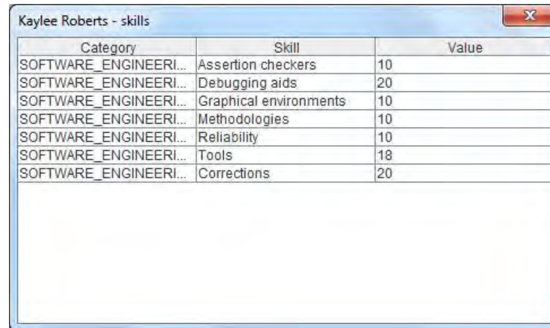


Fig. 8. Worker skills with levels

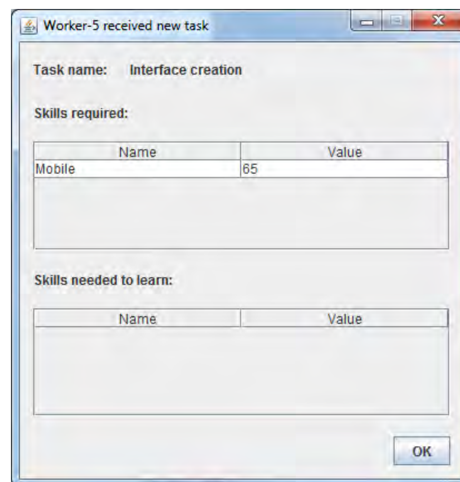


Fig. 9. Task assigned to worker with none learning activities to perform

4.3. *Human Resource Manager Agent*

For the *Human Resource Manager Agent* the scenario starts when it receives a message-request, from the *Project Manager Agent*, to hire a new employee that could work on the task T4. Work of the *Human Resource Manager Agent* consists of comparing skills required for the T4 to skills provided by workers

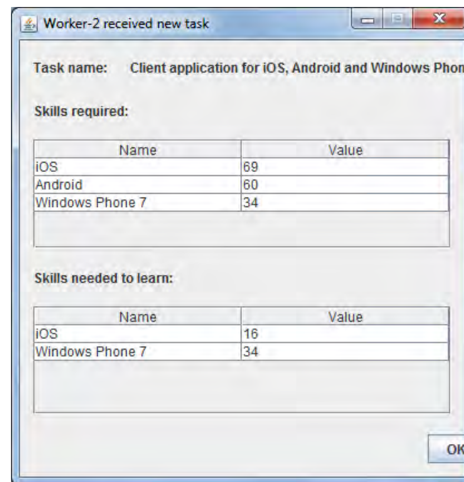


Fig. 10. Task assigned to worker with learning activities

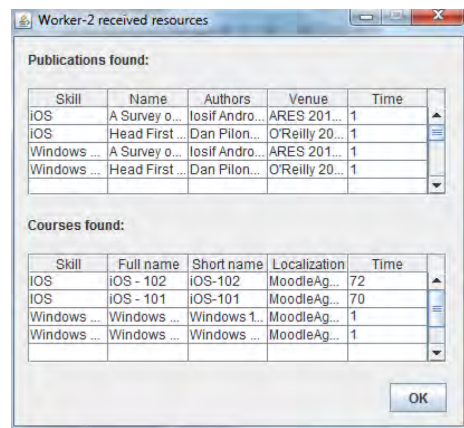


Fig. 11. Worker with found resources

WA4 and WA5. Upon completing such comparison, it decides that WA5 is more suitable for this task. Therefore, the *Human Resource Manager Agent* sends a message with the WA5 contact information and its skills to the *Project Manager Agent*.

4.4. *Resource Module*

Now let us see how this scenario looks from the perspective of the *Resource Module*. First, the *Resource Manager Agent* is contacted by a *Worker Agent* (here, agent representing worker WA2) with a request to search for resources that can improve its skills S1, S3 and S4. It forwards this message to the *Training Resource Manager Agent* and to the *Publication Resources Manager Agent*. Contact information for both agents is obtained from the JADE Directory Facilitator. The dispatched message contains the list of *Skills* and an added information that it was the WA2 that originated the query. After obtaining both answers (again, we omit the case of non-responsive entities) the*Resource Manager Agent* forwards the combined *ResourceList* back to the WA2.

When processing the request, both the *Training Resource Manager Agent* and the *Publication Resources Manager Agent* work in a very similar way. Here, the *Training Resource Manager Agent*, after obtaining request to search for courses concerning S1, S3 and S4, will search the course database. Suppose that for skills S1 and S3 there was only a single course available for each of them, but for skill S4 there were 6. Now the the *Training Resource Manager Agent* will trim this number, and selects only a predefined number of courses, say two. Selection of those two courses is based on the skill level, duration, and cost of each one of them (see, Fig. 13). The *ResponseList* (consisting of courses applicable for each of the three skills) will be send back to the *Resource Manager Agent*. At the same time, the *Publication Resources Manager Agent* will perform similar actions, but instead of searching the course database it will build a query that will be passed to the DBLP search engine. Next it will parse the obtained answer (see, Fig. 12). Finally, the resulting (possibly trimmed) list of publications will be send back to the WA2, as the *ResponseList*.

Note that this scenario captures characteristics of both use cases described in section 1.2. Specifically, the individual support case is subsumed by the above scenario. The *Worker Agent* could act not only in response to the message from the *Project Manager Agent*. It could also work either on request of its owner, or autonomously (proactively or reactively); in all cases it would be searching for the needed resources. Thus the use case 1.2, would simply start from a message send from the *Worker Agent* to the *Resource Manager Agent*.

Let us complete our description by the depiction of an actual communication between the above described JADE agents. This communication was captured in a running system using the JADE Sniffer Agent, and is represented in Fig. 14.

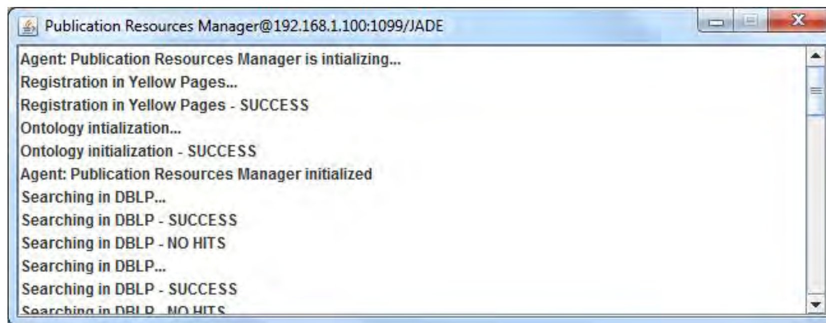


Fig. 12. Searching in Digital Bibliography and Library Project—none resources found

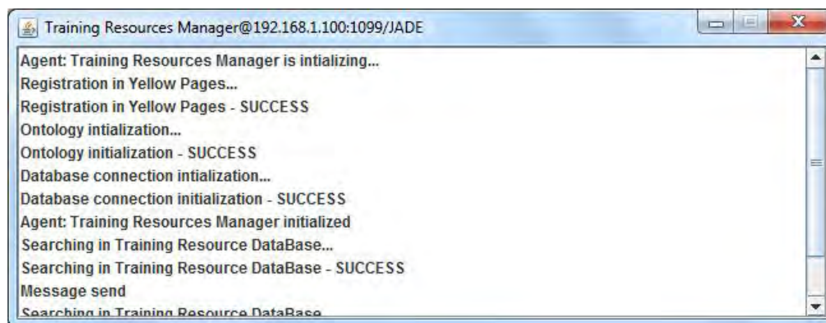


Fig. 13. Searching in Training Resources Database—resources found

Fig. 14. Communication process—messages sent in the system—captured by JADE Sniffer Agent

5. Concluding remarks

The aim of the presented work was to describe a prototype of an agent-based *Information Resource*s provisioning system, facilitating support for workers in a virtual organization. The implemented prototype consists of agents directly supporting workers and project managers (interacting with humans), as well as autonomous agents that facilitate services needed for functioning of the system. The implemented prototype has been interfaced with the DBLP library (as an example of publication resource provisioning), and with the Moodle platform (as an example of e-learning content provisioning).

While the system works with ontologically demarcated resources and semantic information processing, we have select a restricting ontological representation, based on Java classes. This decision was a deliberate one and was caused by factors outlined in sections 2.3 and 3.

In view of known shortcomings of the initial prototype, we plan to proceed as follows. First, we will evaluate lessons learned during system development and testing. Second, we will replace Java class-based ontology by an OWL-based one, and adapt all needed information processing and communication procedures. Here, we will use our JADE add-on that allows JADE agents to communicate by sending OWL snippets (without compiling ontologies into Java classes, see [35]. We will also introduce a newly designed front-end based on the Play framework (for more details, see [35]. Furthermore, we will expand the list of sources of *Information Resources* (both publication and training), to make the system truly heterogeneous. We will report on our progress in subsequent publications.

References

1. Dublin core webpage. `http://dublincore.org/`
2. Java Agent DEvelopment framework. `http://jade.tilab.com/`
3. Log4JADE Agent-based Logging Service. `http://log4jade.sourceforge.net/`
4. Moodle. `http://moodle.org/`
5. `http://www.businessdictionary.com/definition/virtual-organization.html` (2008)
6. `http://www.mwi.pl/badania-i-innowacje/projekty/wkup.html` (2011)
7. The acm computing classification system. `http://www.acm.org/about/class/ccs98-html`
8. Badica, C., Popescu, E., Frackowiak, G., Ganzha, M., Paprzycki, M., Szymczak, M., Park, M.W.: On human resource adaptability in an agent-based virtual organization. In: N.T. Nguyen, R.K. (ed.) New Challenges in Applied Intelligence Technologies. Studies in Computational Intelligence, vol. 134, pp. 111–120. Springer, Heidelberg, Germany (2008)
9. Barnatt, C.: Office space, cyberspace and virtual organization. Journal of General Management 20(4), 78–92 (1995)
10. Bleeker, S.: The Virtual Organization. Sage Thousand Oaks (CA) (1998)

11. Cervenka, R., Trencansky, I.: The Agent Modeling Language - AML. Birkhuser Basel (2007)
12. Czerniejewski, B.: Personal Communication
13. Dieter, F.: Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce. Springer-Verlag, New York (2003)
14. Dunbar, R.: Virtual Organizing, pp. 6709–6717. Thomson Learning, London (2001)
15. Frackowiak, G., Ganzha, M., Gawinecki, M., Paprzycki, M., Szymczak, M., Bădică, C., Han, Y.S., Park, M.W.: Adaptability in an agent-based virtual organization. Internetional Journal Accounting, Auditing and Performance Evaluation (2008), in press
16. Frackowiak, G., Ganzha, M., Paprzycki, M., Szymczak, M., Han, Y.S., Park, M.W.: Adaptability in an agent-based virtual organization—towards implementation. In: WEBIST (Selected Papers). pp. 27–39 (2008)
17. Ganzha, M., Gawinecki, M., Szymczak, M., Frackowiak, G., Paprzycki, M., Park, M.W., Han, Y.S., Sohn, Y.: Generic framework for agent adaptability and utilization in a virtual organization—preliminary considerations. In: Cordeiro, J., et al. (eds.) Proceedings of the 2008 WEBIST conference. pp. IS–17–IS–25. INSTICC Press (2008)
18. Ganzha, M., Paprzycki, M., Gawinecki, M., Szymczak, M., Frackowiak, G., Badica, C., Popescu, E., Park, M.W.: Adaptive information provisioning in an agent-based virtual organization—preliminary considerations. In: Nguyen, N. (ed.) Proceedings of the SYNASC Conference. LNAI, vol. 4953, pp. 235–241. IEEE Press, Los Alamitos, CA (2007)
19. Goldman, S., Nagel, R., Preiss, K.: Agile Competitors and Virtual Organizations. Van Nostrand Reinhold, New York (1995)
20. Hendler, J.: Agents and the semantic web. IEEE Intelligent Systems 16(2), 30–37
21. Jennings, N.: An agent-based approach for building complex software systems. Commun. ACM 44(4), 35–41 (2001)
22. Jennings, N., Wooldridge, M.: Agent technology: foundations, applications, and markets. Springer (1998)
23. Kim, H., Fox, M., Gruninger, M.: An ontology for quality management—enabling quality problem identification and tracing. BT Technology Journal 17(4), 131–140 (1999)
24. Laclavik, M.: Ontology and agent based approach for knowledge management
25. Laclavik, M., Balogh, Z., Hluchy, L., Nguyen, G., Budinska, I., Dang, T.: Pellucid agent architecture for administration based processes
26. Maes, P.: Agents that reduce work and information overload. Commun. ACM 37(7), 30–40 (1994)
27. Nwana, H.S., Ndumu, D.T.: Software agents: an overview. In: Knowledge Engineering Review, vol. 11, pp. 205–244. Cambridge University Press (1999)
28. Omelczuk, A., Wypysiak, M.: Managing human resource adaptability in an agent-based virtual organization (2011)
29. http://ontoweb.aifb.uni-karlsruhe.de/Ontology/index.html
30. Rhee, S.K., Lee, J., Park, M.W., Szymczak, M., Frackowiak, G., Ganzha, M., Paprzycki, M.: Measuring semantic closeness of ontologically demarcated resources. Fundam. Inform. 96(4), 395–418 (2009)
31. Szymczak, M., Frckowiak, G., Ganzha, M., Gawinecki, M., Paprzycki, M., Park, M.W.: Resource management in an agent-based virtual organization—introducing a task into the system. In: Proceedings of the MaSeB Workshop. pp. 458–462. IEEE CS Press, Los Alamitos, CA (2007)
32. http://dblp.uni-trier.de/

33. http://www.eil.utoronto.ca/enterprise-modelling/index.html
34. Warner, M., Witzel, M.: Zarzadzanie organizacja wirtualna. Oficyna Ekonomiczna (2005)
35. Wasielewska, K., Drozdowicz, M., Szmeja, P., Paprzycki, M., Ganzha, M., Lirkov, I., Petcu, D., Badica, C.: Agents in grid system—design and implementation. Springer (2011), to appear
36. Wooldridge, M., Jennings, N.R., Kinny, D.: The gaia methodology for agent-oriented analysis and design. Autonomous Agents and Multi-Agent Systems 3(3), 285–312 (2000)

Maria Ganzha obtained M.S. and her Ph.D. in Applied Mathematics from the Moscow State University, Moscow, Russia in 1987 and 1991 respectively. Her initial research interests were in the area of differential equations, solving mixed wave equations in space with disappearing obstacles in particular, currently she works in the areas of software engineering, distributed computing and agent systems in particular. She has published more than 100 research papers and is on editorial boards of 5 journals and a book series and was invited to Program Committees of over 100 conferences.

Adam Omelczuk is an MS student at the Warsaw University of Technology, where he is applying agent-oriented programming paradigm to the personalized information delivery process. His research interests focus on use of ontologies and semantic data processing.

Marcin Paprzycki (Senior Member of the IEEE, Senior Member of the ACM, Senior Fulbright Lecturer, IEEE CS Distinguished Visitor) has received his M.S. Degree in 1986 from Adam Mickiewicz University in Poznan, Poland, his Ph.D. in 1990 from Southern Methodist University in Dallas, Texas and his Doctor of Science Degree from Bulgarian Academy of Sciences in 2008. His initial research interests were in high performance computing and parallel computing, high performance linear algebra in particular. Over time they evolved toward distributed systems and Internet-based computing; in particular, agent systems. He has published more than 350 research papers and was invited to Program Committees of over 400 international conferences. He is on editorial boards of 14 journals and a book series.

Mateusz Wypysiak is an MS student at the Warsaw University of Technology, where he is applying agent-oriented programming paradigm to the information delivery process. His research interests focus on practical aspects of delivering of personalized information.

Received: January 17, 2012; Accepted: May 25, 2012

DOI: 10.2298/CSIS120101030F

Decentralized Management of Building Indoors through Embedded Software Agents

Giancarlo Fortino[1] and Antonio Guerrieri[2]

DEIS - University of Calabria
Via P. Bucci, cubo 41c, Rende (CS), 87036, Italy
[1]g.fortino@unical.it,[2]aguerrieri@deis.unical.it

Abstract. In order to support personalized people comfort and building energy efficiency as well as safety, emergency, and context-aware information exchange scenarios, next-generation buildings will be smart. In this paper we propose an agent-oriented decentralized and embedded architecture based on wireless sensor and actuator networks (WSANs) for enabling efficient and effective management of buildings. The main objective of the proposed architecture is to fully support distributed and coordinated sensing and actuation operations. The building management architecture is implemented at the WSAN side through MAPS (Mobile Agent Platform for Sun SPOTs), an agent-based framework for programming WSN applications based on the Sun SPOT sensor platform, and at the base station side through an OSGi-based application. The proposed agent-oriented architecture is demonstrated in a simple yet effective operating scenario related to monitoring workstation usage in computer laboratories/offices. The high modularity of the proposed architecture allows for easy adaptation of higher-level application-specific agents that can therefore exploit the architecture to implement intelligent building management policies.

Keywords: Smart Buildings, Multi-Agent Systems, Wireless Sensor and Actuator Networks, Building Management Systems.

1. Introduction

Nowadays, due to advances in communication and computing technologies, the need to have high comfort levels together with an optimization of the energy consumption is becoming important for inhabitants of buildings. Moreover, buildings should also support their inhabitants with automatic emergency and safety procedures as well as context aware information services. To meet all these requirements, future buildings have to incorporate diversified forms of intelligence [7].

We believe that agent-based computing [20] can be exploited to implement the concept of intelligent buildings due to the agent features of autonomy, proactiveness, reactiveness, learnability, mobility and social ability. Specifically agents can continuously monitor building indoors and their living inhabitants to gather useful data from people and environment and can cooperatively achieve

even conflicting specific goals such as personalized people comfort and building energy efficiency.

A few research efforts based on agents have been to date proposed to design and implement intelligent building systems [25] [17] [8] [28] [27] [23]. However, none of them provide agents embedded in the sensor and actuator devices that would introduce intelligence decentralization and improve system efficiency. This is due to the exploitation of conventional sensing and actuation systems that do not offer distributed computing devices for sensing and actuation. To overcome this limitation, wireless sensor and actuator networks (WSAN) [26] can be adopted. WSANs represent a viable and more flexible solution to traditional building monitoring and actuating systems (BMAS), which require retrofitting the whole building and therefore are difficult to implement in existing structures. In contrast, WSAN-based solutions for monitoring buildings and controlling equipment, such as electrical devices, heating, ventilation and cooling (HVAC), can be installed in existing structures with minimal effort. This should enable monitoring of structure conditions, and space and energy (electricity, gas, water) usage while facilitating the design of techniques for intelligent device actuation.

The implementation of the proposed architecture is based on MAPS (Mobile Agent Platform for Java Sun SPOTs) [3] at sensor/actuator node side and on Jade [5] OSGi-based application at coordinator side.

The main contribution of this paper is the definition of *A-BMF* (Agent based Building Management Framework), a decentralized and embedded agent oriented architecture for the management of intelligent buildings that is based on WSANs and overcomes the limitations of the aforementioned solutions [25] [17] [8] [28] [27] [23]. In particular, the aim of our architecture is to optimize and fully decentralize the sensing and actuation operations through distributed cooperative agents both embedded in sensor/actuator devices and running on more capable coordinators (PC, plug computers, PDA, smartphones). This would enable more effectiveness in programming the sensing and actuation operations and more efficiency in the management of distributed sensor and actuator nodes. Moreover, the proposed architecture can be easily programmed to support a wide range of building management applications integrating comfort, energy efficiency, emergency, safety, and context-aware information exchange aspects.

The rest of this paper is organized as follows. Section 2 describes approaches related to our work. In Section 3 the proposed agent-based architecture for building management is defined. Section 4 presents the MAPS-based implementation of the low-level architecture, specifically the sensor/actuator agents. Section 5 shows the system GUI and a system deployment for monitoring the workstation usage in computer laboratories. Finally, conclusions are drawn and directions of future work elucidated.

2. Related Work

In [25] the authors present the MASBO (Multi-Agent System for Building cOntrol) architecture that aims to provide a set of software agents to support both on-line and off-line applications for intelligent work environments. MASBO is used to develop a multi-agent system (MAS) able to tradeoff energy saving and inhabitants' preferences where preferences can be learnt and predicted through an unsupervised online real-time learning algorithm (analyzing inhabitants' behavior). MASBO agents reside on a server and constantly monitor data from sensors and eventually actuate some commands. MASBO works as an enhancement to an existing building automation system by adding learning, reasoning and autonomous capabilities. The responsibility of controlling sensors and actuators, and keeping a requested environmental value constant is not addressed by MASBO.

In [17] the authors propose a working solution to the problem of thermal resource distribution in a building using a market-based MAS. Computational agents representing individual temperature controllers bid to buy or sell cool or warm air. The agents, running in a monolithic process on a workstation, are able to distribute the thermal resources so that all the building offices have an equitable temperature distribution. Temperature sensors and air flow actuators are all accessible directly through distributed hardware modules via a network connection.

In [8] the authors describe a MAS that monitors and controls an office building in order to provide added values like energy saving together with the delivery of energy. The developed system is distributed in the sense that some agents are located on PDAs and others run on the Bluetooth access points (workstations) that communicate with the PDAs. The system makes use of the existing power lines for communication between the agents and the sensing and actuation system controlling lights, heating, ventilation, etc.

In [28] a conceptual framework, namely Cyber-enabled Efficient Building Energy Management System (CEBEMS), is presented. Its intent is increasing energy efficiency, lowering dependence on the energy grid, and providing an economic incentive for the end user. It enables distributed control methodology using MAS for efficient management of both electrical and thermal energy systems for realizing maximum efficiency energy management. MAS aim to achieve system-wide objectives, which may not be solved using a single agent, but by coordination and communication among the agents.

In [27] authors do a demonstration of data gathering from a WSN. In the system proposed, users can query and view the local data in an ad-hoc manner, and possibly remotely configure and manipulate the data capture process. For the purposes of this demonstration, authors adopted Agent Factory Micro Edition (AFME) [22]. The Agents, implemented on AFME, are programmed to answer users' requests.

In [23] a simulation of a building environment where agents can manage the allocation of resources and facilitate the residents' lives is presented. In the designed system, sensors deployed in a building send their information to

agents. Agents, that reside on workstations, process information and send them to a fuzzy controller [16] that eventually transmits a proper signal to switches, valves or other actuators.

Differently from the described approaches, our agent-based architecture embeds agents both into the wireless sensor and actuator network used as infrastructure for building monitoring and control and on more capable coordinators. This important feature would provide decentralized intelligence and improve system efficiency.

Table 1 summarizes the characteristics of the works reported above.

3. Agent-based Architecture

The agent-based architecture (see Fig. 1) of A-BMF for decentralized and embedded building management is composed of a building manager agent (BMA), which is installed in the control workstation, coordinator agents (CAs), which run in the basestations, and sensor agents (SAs), which are executed in the sensor/actuator nodes. Specifically, the architecture relies on a multi-basestation approach to allow for large buildings composed of multiple floors and diversified environments. Thus, the architecture is purposely hybrid: hierarchical and peer-to-peer. Interaction between CAs is peer-to-peer whereas interactions between CAs and their related SAs (or SA cluster) and between BMA and CAs are usually master/slave. Moreover, SAs of the same cluster coordinate to dynamically form up a multi-hop ad-hoc network rooted at the master CA.

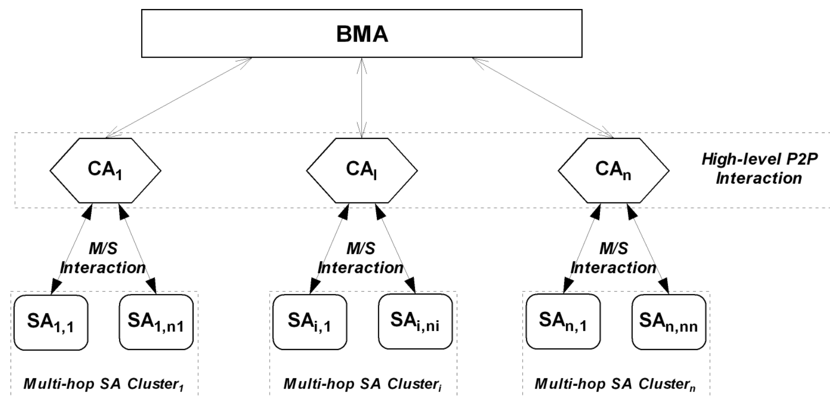


Fig. 1. Agent-based architecture for decentralized and embedded management of buildings based on wireless sensor and actuator networks.

Table 1. Related Work comparison.

	Aim of the work	**Agents location**	**WSAN support**
MASBO [23]	Tradeoff energy saving and inhabitants' preferences	Server	NO, but agents can interface to WSN
Market-based MAS [15]	Distributing the thermal resources across a building	The agents run in a monolithic process on a workstation	NO
MAS to monitor and control office building [8]	Providing energy saving together with the delivery of energy	Some agents are located on PDAs and others run on the Bluetooth access points.	NO
CEBEMS [26]	Increasing energy efficiency, lowering dependence on the energy grid, and providing an economic incentive for the end user	N/A. CEBEMS is still a conceptual framework	N/A
System of data gathering on AFME [25]	Allowing users to query and view data from a WSN	Agents are embedded	YES
MAS with fuzzy approach [21]	Simulation of an environment where agents can manage the allocation of resources and facilitate the residents' lives	Workstation	N/A
A-BMF	Optimizing and fully decentralizing the sensing and actuation operations through distributed cooperative agents	Agents are both embedded in sensor/actuator devices and running on more capable coordinators	YES, our agent-based architecture embeds agents into the WSAN used as infrastructure for building monitoring and control

In Fig. 2 the main functionalities of BMA, CA and SA are shown according to a layered organization that is partially derived from the Building Management Framework (BMF) [15].

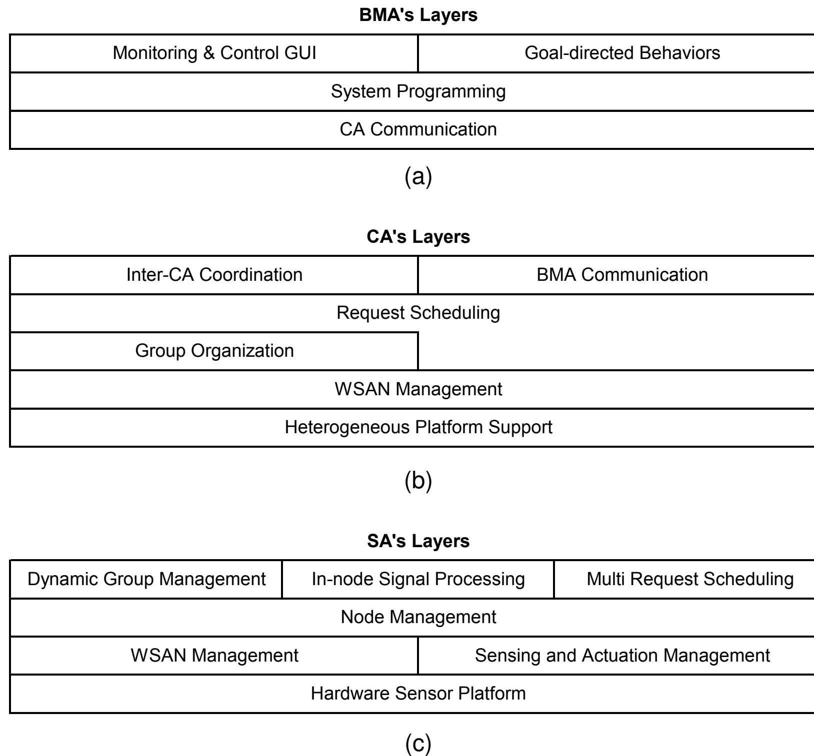


Fig. 2. The layered organization of (a) BMA, (b) CA and (c) SA.

The BMA is the top level agent that manages the distributed agent based architecture. The BMA includes the following layers:

- *CA Communication* allows the message based communication between BMA and the CAs.
- *System Programming* is the layer which allows to program the distributed agent network (SAs are reached through their CAs).
- *Monitoring & Control GUI* provides a GUI through which the building manager can issue requests to configure/program the agent-based building network and visualize its status and the monitored data.
- *Goal-directed Behaviors* permits implementing specific building monitoring and control strategies to realize specific applications (energy monitoring, comfort, etc).

The CA is the middle level agent which is able to manage a cluster of SAs which refers to a given area of the intelligent building. The CA includes the following layers:

– *Heterogeneous Platform Support* incorporates a set of adapters that allow interfacing the system with different type of sensor/actuator platforms. An adapter is linked to a specific hardware device able to communicate with a specific sensor platform in the network.
– *WSAN Management* allows to fully manage a WSAN cluster. This layer supports packet coding/decoding according to the A-BMF application-level protocol and packet transmission/reception to/from the WSAN cluster. Moreover, this layer supports device discovery within the cluster.
– *Group Organization* provides group-based programming of sensors and actuators, tracking of nodes and groups in the system, and management of node configurations and group compositions. Node organization in groups is specifically defined to capture the morphology of buildings. Nodes belong to groups depending on their physical (location) or logical (operation type) characteristics.
– *Request Scheduling* allows the support for higher-level application-specific requests. Through this layer, a CA can ask for the execution of specific tasks to single or multiple SAs or groups of SAs. Moreover, this layer keeps track of the requests submitted to the system, waits for data from the nodes and passes them to the requesting applications. A request is formalized through the following tuple: R = $<$Obj, Act, R, LT$>$, where Obj is a specific sensor or actuator belonging to a node, Act is the action to be executed on Obj, R is the frequency of each executed Act, LT is the length of time over which these actions are to be reiterated. Moreover, a request can target a single node or a group of nodes having Obj.
– *Inter-CA Coordination* offers efficient mechanisms for coordination between CAs. Specifically, CAs cooperate for submitting queries and retrieving data spanning multiple SA clusters.
– *BMA Communication* allows the message based communication between CA and the reference BMA.

The SA is the low level agent running on sensor/actuator nodes to perform given sensing/actuating operations. The SA is designed around the following layers:

– *Hardware Sensor Platform* allows to access the hardware sensor/actuator platform. In particular, the layer facilitates the configuration of the platform specific drivers and the use of the radio.
– *WSAN Management* manages the node communication with the reference CA according to the A-BMF application protocol and among the cluster nodes through the network protocol provided by the node sensor platform.
– *Sensing and Actuation Management* allows to acquire data from sensors and execute actions on actuators. In particular, this layer allows to address different types of sensors/actuators in a platform independent way.

- *Node Management* is the core of the SA and allows to coordinate all the layers for task execution. In particular, it handles events from the lower layers every time that a network packet arrives or data from sensor/actuator are available, and from the upper layers every time that data are processed or a stored request has to be executed.
- *Dynamic Group Management* provides group management functionalities to the SA. A node can belong to several groups at the same time and its membership can be dynamically updated on the basis of requests from CAs.
- *In-node Signal Processing* allows the SA to execute signal processing functions on data acquired from sensors [4]. It can compute simple aggregation functions (e.g. mean, min, max, variance, R.M.S.) and more complex user-defined functions on buffers of acquired data.
- *Multi Request Scheduling* allows the scheduling of sensing and actuation requests. In particular, it stores the requests from CAs and schedules them according to their execution rate.

4. MAPS-Based Implementation

The agent-based building management architecture of A-BMF is currently implemented through MAPS [3], our agent-based framework for developing WSN applications on the Sun SPOT sensor platform. MAPS has been selected as one of the most representative frameworks for agent oriented programming of sensor/actuator nodes [2] [11] [12]. Only two other java-based platforms currently exist: AFME [22] and MASPOT [19]. The former is based on a more complex programming model and provides basic operations less efficient than MAPS. The latter is mainly centered on agent mobility and does not provide a suitable API for programming complex agent behaviors. Thus MAPS has been adopted as the one fulfilling the needed requirements of effective agent programming and efficient operations. Moreover, currently the mobility feature of MAPS agents is not used in the current implementation of A-BMF. In this section we first provide a brief overview of MAPS (more details can be found in [3], [21]) and, then, present the MAPS-based implementation of the proposed building management architecture at sensor-node side, specifically behavior and event-based interactions of the SA.

4.1. MAPS: a brief overview

MAPS [3] [21] [1] is an innovative Java-based framework specifically developed on Sun SPOT technology for enabling agent-oriented programming of WSN applications. It has been defined according to the following requirements:

- *Component-based lightweight agent server architecture* to avoid heavy concurrency and agents cooperation models.
- *Lightweight agent architecture* to efficiently execute and migrate agents.

- *Minimal core services* involving agent migration, agent naming, agent communication, timing and sensor node resources access (sensors, actuators, flash memory, and radio).
- *Plug-in-based architecture* extensions through which any other service can be defined in terms of one or more dynamically installable components implemented as single or cooperating (mobile) agents.
- *Use of Java language* for defining the mobile agent behavior.

The architecture of MAPS (see Fig. 3) is based on several components interacting through events and offering a set of services to mobile agents, including message transmission, agent creation, agent cloning, agent migration, timer handling, and an easy access to the sensor node resources. In particular, the main components are the following:

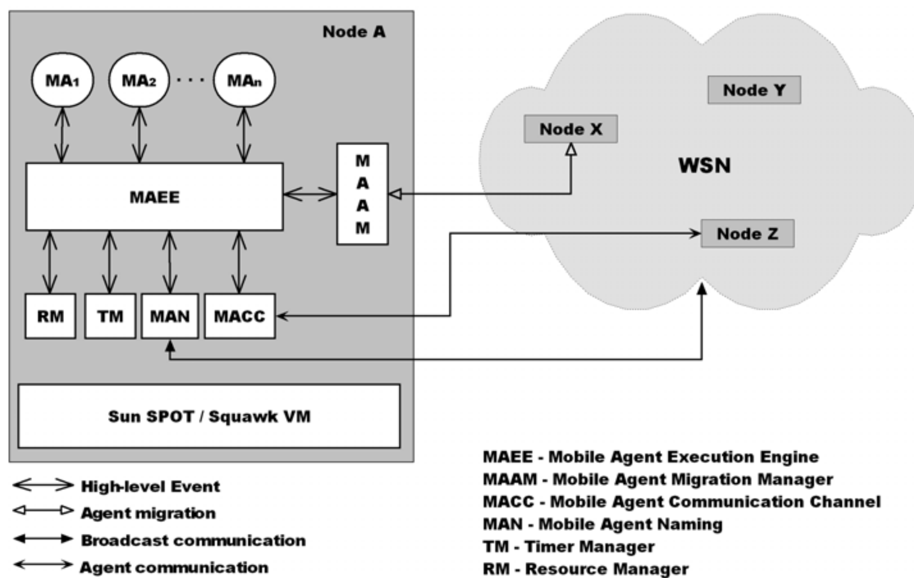


Fig. 3. The architecture of MAPS.

- *Mobile Agent (MA)*. MAs are the basic high-level component defined by user for constituting the agent-based applications.
- *Mobile Agent Execution Engine (MAEE)*. It manages the execution of MAs by means of an event-based scheduler enabling lightweight concurrency. MAEE also interacts with the other services-provider components to fulfill service requests (message transmission, sensor reading, timer setting, etc) issued by MAs.
- *Mobile Agent Migration Manager (MAMM)*. This component supports agents migration through the Isolate (de)hibernation feature provided by the Sun

SPOT environment. The MAs hibernation and serialization involve data and execution state whereas the code must already reside at the destination node (this is a current limitation of the Sun SPOTs which do not support dynamic class loading and code migration).
- *Mobile Agent Communication Channel (MACC)*. It enables inter-agent communications based on asynchronous messages (unicast or broadcast) supported by the Radiogram protocol.
- *Mobile Agent Naming (MAN)*. MAN provides agent naming based on proxies for supporting MAMM and MACC in their operations. It also manages the (dynamic) list of the neighbor sensor nodes which is updated through a beaconing mechanism based on broadcast messages.
- *Timer Manager (TM)*. It manages the timer service for supporting timing of MA operations.
- *Resource Manager (RM)*. RM allows access to the resources of the Sun SPOT node: sensors (3-axial accelerometer, temperature, light), switches, leds, battery, and flash memory.

The dynamic behavior of a mobile agent (MA) is modeled through a multi-plane state machine (MPSM). Each plane [6] may represent the behavior of the MA in a specific role so enabling role-based programming. In particular, a plane is composed of local variables, local functions, and an automaton whose transitions are labeled by Event-Condition-Action (ECA) rules E[C]/A, where E is the event name, [C] is a boolean expression evaluated on global and local variables, and A is the atomic action. Thus, agents interact through events, which are asynchronously delivered and managed by the MAEE component.

It is worth noting that the MPSM-based agent behavior programming allows exploiting the benefits deriving from three main paradigms for WSN programming: event-driven programming, state-based programming and mobile agent-based programming.

MAPS is also interoperable with the JADE framework [5]. Specifically, a JADE-MAPS gateway [9] has been developed for allowing JADE agents to interact with MAPS agents and vice versa. While both MAPS and JADE are Java-based, they use a different communication method. JADE sends messages according to the FIPA standards (using the ACL specifications), while MAPS creates its own messages based on events. Therefore, the JADE-MAPS Gateway facilitates message exchange between MAPS and JADE agents. This inter-platform communication infrastructure allows rapid prototyping of WSN-based distributed applications/systems that use JADE at the basestation/coordinator/host sides and MAPS at the sensor node side.

4.2. MAPS-based sensor agents

MAPS based SA is compliant with the SA architecture discussed in Section 3. According to MAPS agent programming, the SA is composed of a behavior and an interaction protocol based on events. In particular, the behavior defines the logic of the SA through a set of planes representing its functionalities. The

interaction protocol allows to interact with the CA to provide the requested services. In the following subsections we first describe the event based interaction protocol between CA and SA which provide a consistent snapshot of the services that an SA can offer to a CA and how such services can be exploited; then we detail the SA's behavior that shows the SA's architecture composed of management and sensing planes defined as finite state machines.

Event-based interaction protocol. The MAPS-based SA (hereafter simply named SA) interacts with its cluster CA through events as sketched in the sequence diagram of Fig. 4. Once the SA is created, it periodically emits the BM_SA_ADVERTISEMENT event until the CA sends a configuring event (group management or request scheduling). Through the BM_GROUP_MANAGEMENT event, the CA manages the membership of target SAs (see Section 3). After the SA processes the received event, it sends the BM_ACK event to the CA. The BM_SENSOR_SCHEDULE (or BM_ACTUATOR_SCHEDULE) event allows to request a specific sensing (or actuation) operation to target SAs. The SA transmits sensed (processed) data to the CA through the BM_DATA event. The CA can unschedule previously scheduled requests through the BM_UNSCHEDULE event. Finally the CA sends out the BM_SA_RESET event to reset target SAs.

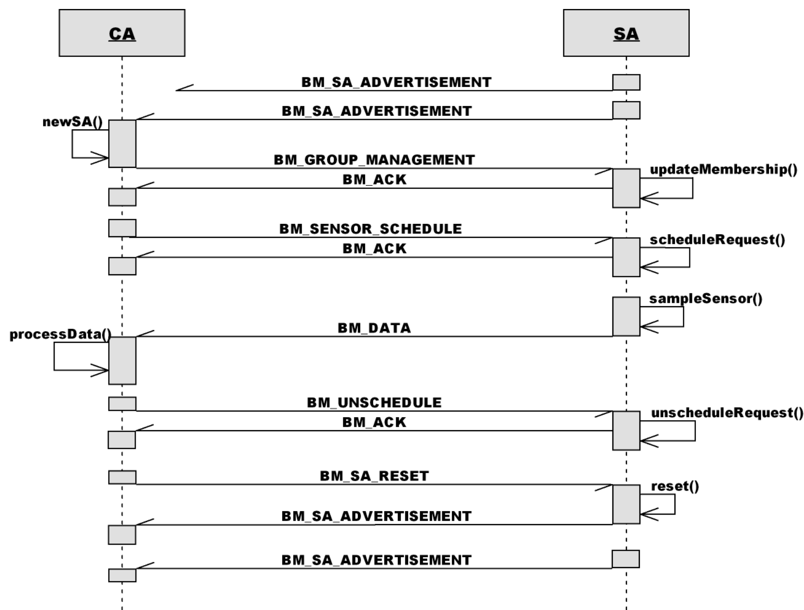


Fig. 4. Sequence Diagram of the interactions between CA and SA.

Tables 4 and 5 in Appendix A report the defined MAPS-based building management events and the predefined values of their parameters. In particular, an event is defined by its *standard parameters*: EventSender ID, EventTarget ID, Event Type, Event Occurrence. The defined events are of two possible super types: MSG (sent by CA to SA) and MSG_TO_BASESTATION (sent by SA to CA). Both types are further specialized in the defined BM events as reported in the pairs <MSG_TYPE, BM_event>of the 3rd column of Table 4 in Appendix A. Moreover, each event type has its own additional parameters, which are described in Table 5 in Appendix A. It is worth noting that the ADDRESSEE value can be set through the regular expression formalized in Eq. 1 where SA is a sensor agent of the building management architecture, G is an element from the set of defined groups, STO is a set theory operator (e.g. union, intersection, difference) and NOT is the negation. Thus, the addressee of an event can be either one or more SAs, or SAs belonging to groups or complex compositions of groups.

$$SA^{+}|(([NOT]G[STO[NOT]G]^{*}) \quad (1)$$

Sensor Agent behavior. The SA agent behavior consists of two types of planes: Manager plane and Request plane. While the Manager plane is created at the SA creation time and handles all node targeting events, a Request plane is created by the Manager plane every time that a new request schedule is received. This type of plane is removed when it completes its task or due to the reception of an unschedule event. Agent planes receive events from the MAPS dispatcher component that is programmed to deliver the events fetched from the agent queue to the plane in charge to process them according to some dispatcher rules (DR). Fig. 5 shows the SA behavior architecture. The dispatcher rules are reported in Table 2.

Table 2. Dispatcher rules.

Event	Plane
BM_SENSOR_SCHEDULE	MANAGER
BM_ACTUATOR_SCHEDULE	MANAGER
BM_UNSCHEDULE	MANAGER
BM_GROUP_MANAGEMENT	MANAGER
BM_SA_RESET	MANAGER
Event.TMR_EXPIRED <ID, ID_MANAGER_PLANE>	MANAGER
Event.TMR_EXPIRED <ID, REQUEST_PLANE_ID>	REQUEST
Event.SENSOR_CURRENT_READING <ID, REQUEST_PLANE_ID>	REQUEST

The Manager plane is reported in Fig. 6. In particular, after agent creation, the Manager plane starts a periodic timer to advertise the agent presence along

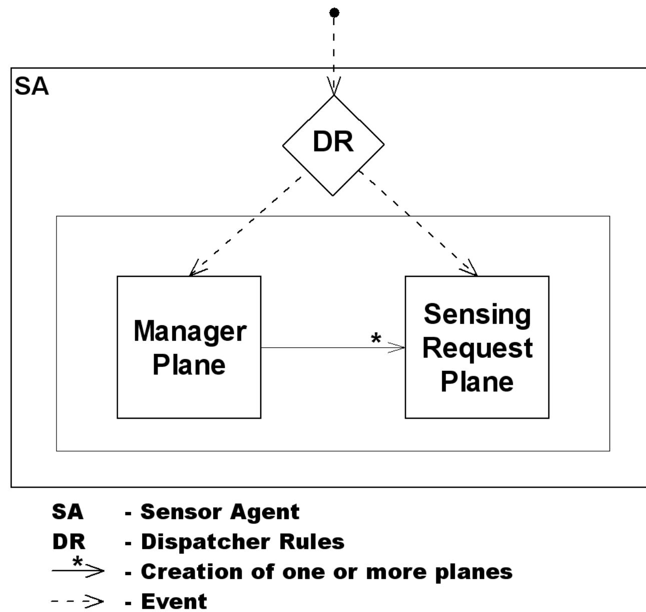


Fig. 5. The SA behavior architecture.

with its sensor/actuator available functions and waits for an incoming event from the CA. When it receives the first event, the timer is reset. Each received event is filtered against the current SA's group membership. If the filtered event is for the current SA, it is processed according to its type. A more detailed description of each action of the Manager plane is provided using both a self-explanatory pseudocode (see Fig. 7) and the MAPS code (intended for MAPS programmers; see Fig. 17 in the Appendix B).

In Fig. 8 the Sensing Request plane is portrayed. This plane is created every time that the agent receives a BM_SENSOR_SCHEDULE event. In particular, after the Sensing Request plane creation, the plane creates and submits the MAPS sensing event formalizing the sensing request. A sensing request can be either one-shot or periodic with a given lifetime. The request is scheduled until LIFETIME_ELAPSED==true after the expiration of the periodic timer driving the submission of the sensing event. A more detailed description of each action of the Sensing Request plane is provided using both a self-explanatory pseudocode (see Fig. 9) and the MAPS code (intended for MAPS programmers; see Fig. 18 in the Appendix B).

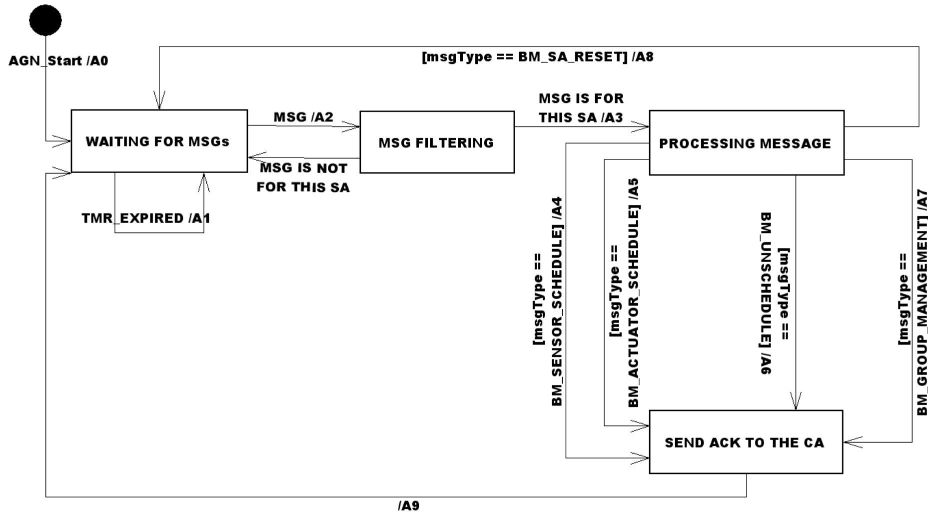


Fig. 6. The SA's Manager plane.

```
A0: firstProcessedEvent=FALSE;
    Start a periodic Event.TMR_EXPIRED to send the BM_SA_ADVERTISEMENT.
A1: Send BM_SA_ADVERTISEMENT to CA
A2: if the MSG is for this SA
    firstProcessedEvent=TRUE && resetTimer (ID_TIMER)
A3: msgType = msgEvent.getParam(ParamsLabel.MSG_TYPE)
A4: Create a new Sensor Plane:
    PlaneID = ID_REQUEST, the Request as parameter and start it.
A5: Create a new Actuator Plane:
    PlaneID = ID_REQUEST,  the Request as parameter and start it.
A6: Unschedule the Request deallocating the Plane with ID = ID_REQUEST
A7: Update current SA Group Membership
A8: Reset the SA and deallocate all the Request Planes;
    firstProcessedEvent=FALSE
A9: Send BM_ACK to CA
```

Fig. 7. The SA's Manager plane pseudocode.

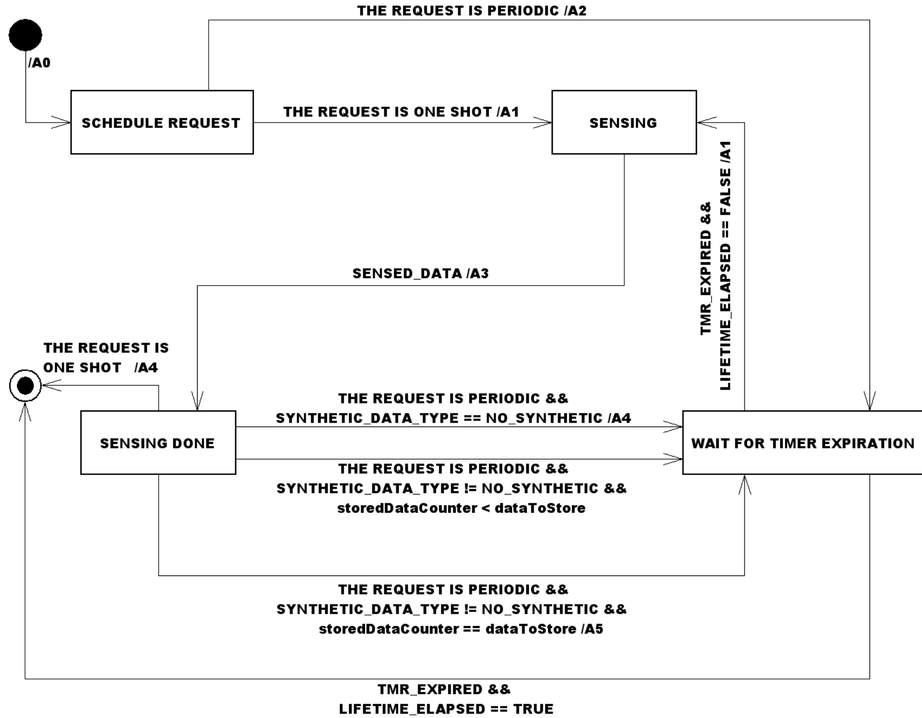


Fig. 8. The SA's Sensing Request plane.

```
A0: Process Request
A1: Create and submint a Sensing Event on the Sensor Requested
A2: Inizialize and Submit a TMR_EXPIRED Event with the Params PERIOD and LIFETIME;
    Set dataToStore
A3: Store sensed data and increase the storedDataCounter
A4: if DATA_TYPE.VALUE == "threshold notification"
        Send sensed data to CA if the threshold is verified and reset the
            storedDataCounter
    else Send sensed data to CA and reset the storedDataCounter
A5: Calculate the SYNTHETIC_DATA_TYPE requested,
    if DATA_TYPE.VALUE == "threshold notification"
        Send synthetic data to CA if the threshold is verified and reset the
            storedDataCounter
    else Send synthetic data to CA and reset the storedDataCounter
```

Fig. 9. The SA's Sensing Request plane pseudocode.

5. A system deployment: monitoring workstation usage in computer laboratories

To show the functionality and effectiveness of the proposed architecture for the management of building indoors, we present an example of system deployment for the monitoring of workstation usage in a computer laboratory or in offices. The wireless sensor network consists of heterogeneous sensor nodes based on Sun SPOTs that are used to collect information about the ambient light (through the standard Sun SPOT light sensor), the user presence (through a Wieye IR sensorboard [10]) and the electricity consumed by the workstation (through a customization of the ACme electricity sensorboard [18]). Every Sun SPOT holds a SA able to manage a set of requests while the basestation holds a CA that allows to manage the SAs. The SAs and CA in the system are shown in Fig. 10. In particular, while the interaction between SAs and CA is logically a direct interaction, SAs are organized in a multi-hop clusters which implies that a message sent by an SA may traverse such multi-hop networks before arriving at the CA.

It's worth noting that in the implementation of the case study, as one only SA cluster was defined, the BMA and CA agents were collapsed in one only agent with the goal of energy monitoring.

In Fig. 11, the main window of the Building Management GUI is shown. It is organized in five main sections supporting all the functionalities provided by the system:

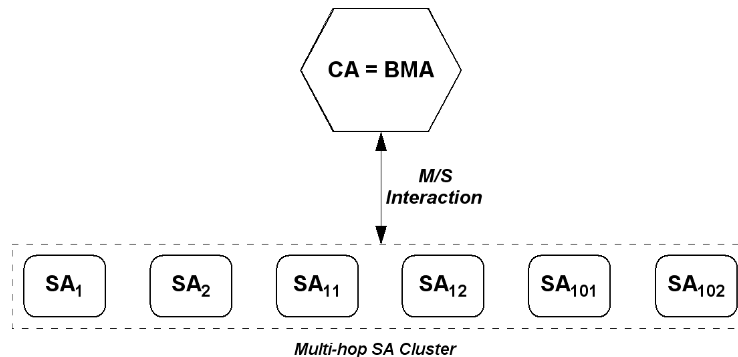


Fig. 10. The application Agents.

– *Nodes* and *Groups Management* sections allows to visualize the nodes of the WSAN and configure groups, respectively. By right clicking on the sensors/groups the user can configure sensor/actuator requests to schedule on the nodes;

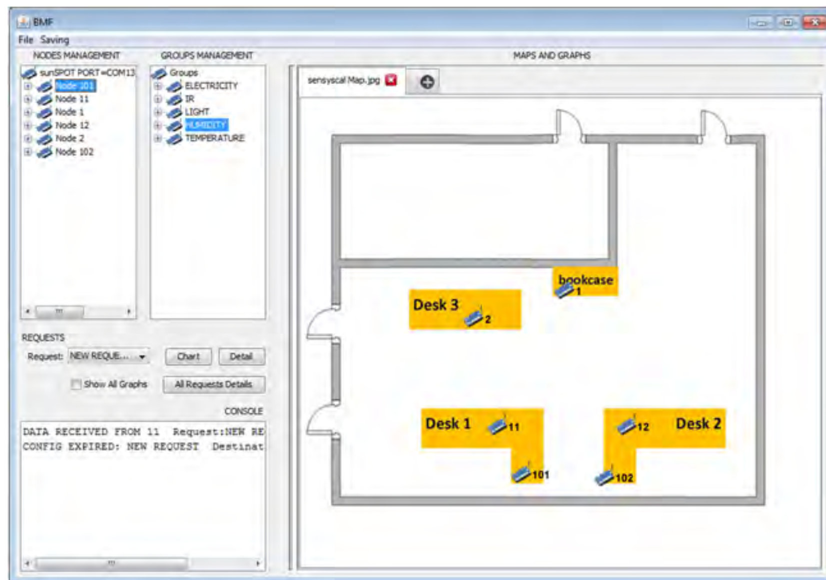


Fig. 11. The Building Management GUI.

- *Request* section allows to list details of scheduled requests, display data charts related to the scheduled requests, unschedule and re-schedule requests;
- *Maps and Graphs* section allows visualizing WSAN deployment maps and displaying charts of the data coming from the sensors (examples of charts are shown in Fig. 13, 14 and 15;
- *Console* section displays the real-time log of the activity of the system;
- *File and Saving* menu section enables to save data from the system in structured files and load stored files to display them in the GUI.

In Fig. 12, the graphical window for sensor/actuator request scheduling is shown. The window allows setting the parameter of a new request: name, destination (specific nodes or group composition), execution period, lifetime, one shot request or unlimited lifetime flags, action type and related device, possible actuator parameters, requested sensed data possibly filtered by thresholds and/or synthetic data is requested and its type (average/max/min) and eventual threshold parameters can be set.

In the experimental system deployment the following requests were set:

- the average of the ambient temperature value (in C) is collected every 60 seconds from node 1;
- the average of the ambient light value (in lux) is collected every 60 seconds from node 2;
- the mean electricity data (in watt) are gathered every minute from nodes 101 and 102;

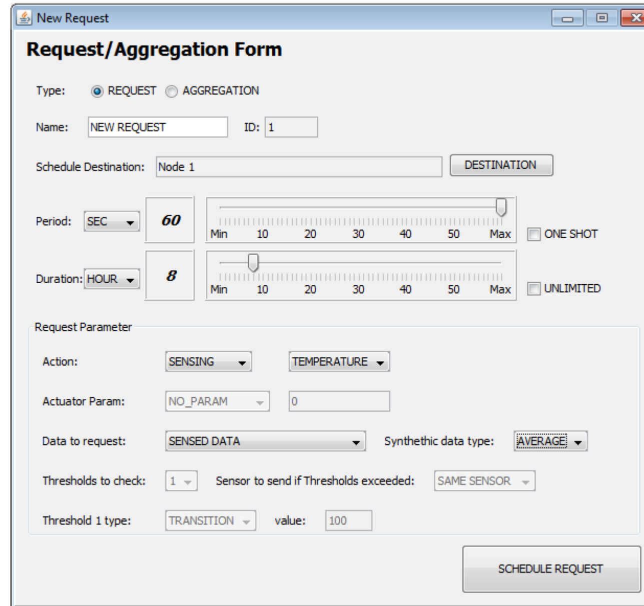


Fig. 12. The graphical window for sensor/actuator request scheduling.

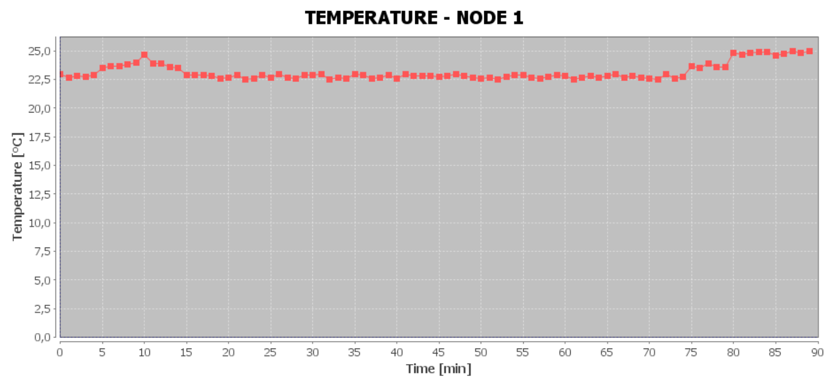
– the max IR sensor value is sensed every minute on nodes 11 and 12.

The aim of the experiment was the monitoring of two workstations in a computer laboratory of the Technest incubator at University of Calabria to understand their users' behavior. Several snapshots of a significant monitoring activity of the duration of 90 min are shown in Fig. 13, 14 and 15.

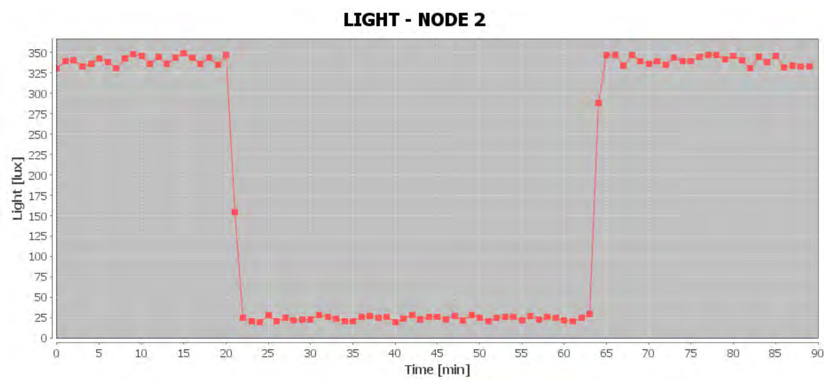
In particular, Fig. 13 shows the real-time data of the ambient temperature and the ambient light. It is clear that while the temperature in the laboratory is almost constant, the light was switched off when the room was empty.

Fig. 14 shows the activity of the worker at the Desk 1. While in the first 15 minutes he was doing some word processing, before living the workstation, he started an hard processing task to his PC that ended at the minute 80.

Fig. 15 illustrates the activity of the worker at the Desk 2. He was doing some word processing till the minute 20 and after the minute 65. In the meanwhile he was not to his desk, but his PC was left on (and with no processing task executing). Desk 2 monitoring shows how a waste of energy could be detected using the A-BMF. In particular, the waste detection can be done only after a setup phase useful to understand the signature of a particular PC activity. An example of signature extrapolated for the PC at desk 2, on the basis of 50 runs, is shown in Fig. 16 where four different working activities are displayed. In particular, subsequent the activities are: (i) active doing word processing (ii) active doing word processing and downloading stuff, (iii) inactive with the screen switched off and downloading stuff, and (iv) inactive with the screen switched

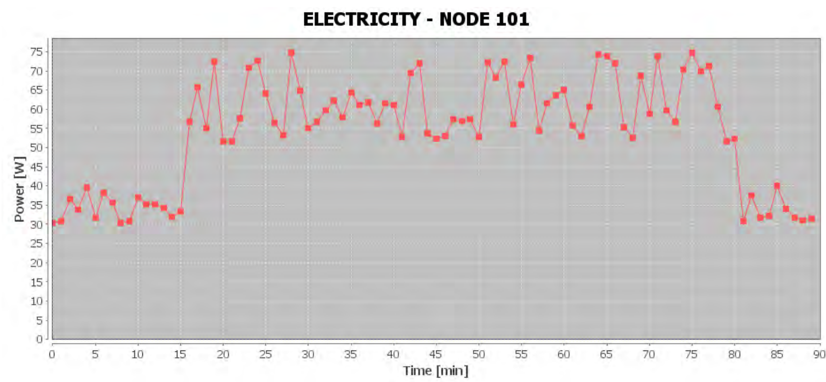


(a)

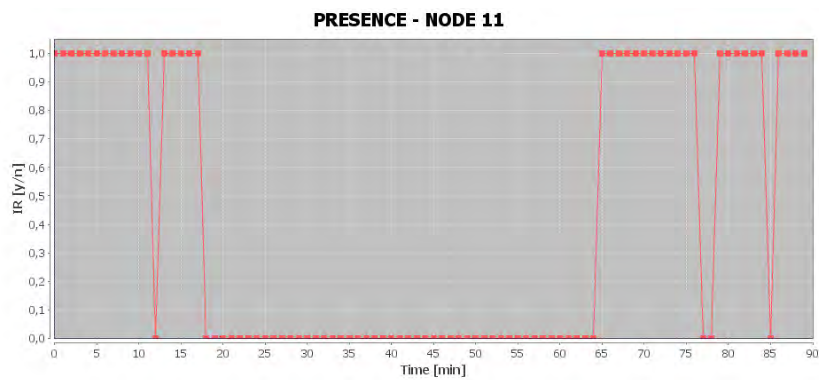


(b)

Fig. 13. Real-time data of the (a)ambient temperature and (b) ambient light.

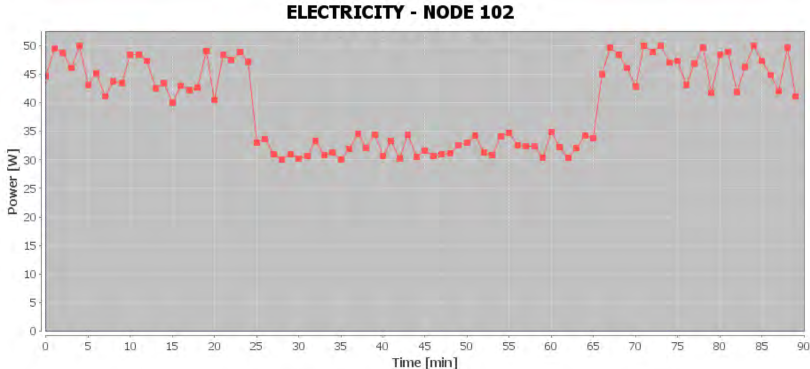


(a)

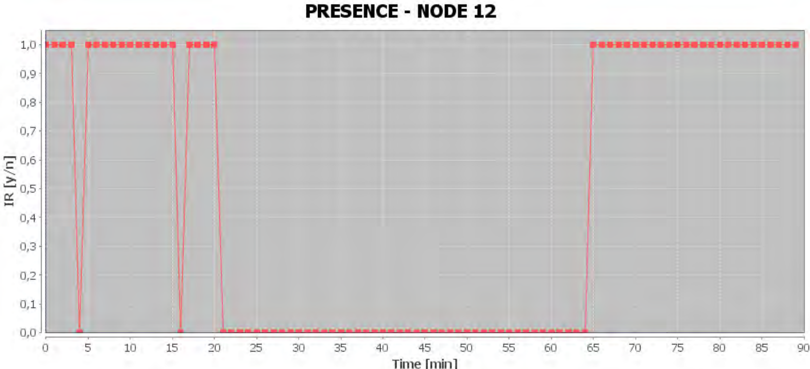


(b)

Fig. 14. Real-time data of the Desk 1. (a) workstation consumed power and (b) user presence.



(a)



(b)

Fig. 15. Real-time data of the Desk 2. (a) workstation consumed power and (b) user presence.

off. Table 3 shows the mean and the standard deviation of the power consumed for the activities above.

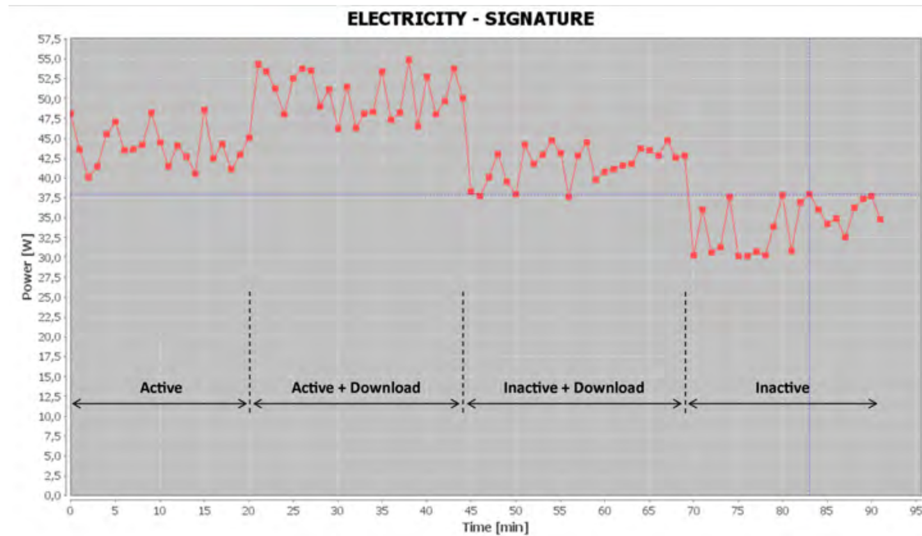


Fig. 16. The signature of the PC at desk 2.

Table 3. Signature characteristics (Mean and Standard Deviation) per activity.

	Mean [W]	Standard Deviation [W]
Active	43,96	2,48
Active + Download	50,51	2,84
Inactive + Download	41,74	2,24
Inactive	34,02	3,01

6. Conclusions and Future Work

In this paper we have proposed A-BMF, an agent-based architecture for flexible, efficient and embedded sensing and actuation in buildings. Specifically, the distributed software architecture is embedded into both WSANs and more capable computing devices (e.g. PCs, smartphones, plug computers). The proposed architecture can be seen as basic middleware for developing intelligent building management systems to achieve the Smart Building concept. Currently the

proposed architecture is exploited to monitor the space occupation and energy expenditure in computer laboratories for students to analyze energy consumption patterns with respect to users' behavior so as to semi-automatically implement behavior policies. In the current implementation, BMA and CA are merged into a component-based application implemented through OSGi [24]. Moreover, only one cluster can be deployed. On-going work is aimed at completing the JADE-based implementation of the multi-cluster architecture founded on the BMA and on multiple coordinated CAs. Future work will be devoted to: (i) the design of a higher-level agent-based architecture for Smart Buildings atop the proposed architecture to trade off inhabitants' personal comfort and building energy expenditure; (ii) the support of user mobility in buildings based on the interoperation between body sensor network worn by users and the intelligent agent based building infrastructure; (iii) the formalization of the A-BMF system through communicating real time state machine-based formalisms [14] for verification of A-BMF-based application scenarios; (iv) the exploitation of streaming techniques [13] to enhance sensor data collecting at application and network level.

Acknowledgments. This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053, and by TETRis - TETRA Innovative Open Source Services, funded by the Italian Government (PON 01-00451).

References

1. Aiello, F., Bellifemine, F.L., Fortino, G., Galzarano, S., Gravina, R.: An agent-based signal processing in-node environment for real-time human activity monitoring based on wireless body sensor networks. Journal of Engineering Applications of Artificial Intelligence 24, 1147–1161 (October 2011), http://dx.doi.org/10.1016/j.engappai.2011.06.007
2. Aiello, F., Fortino, G., Galzarano, S., Gravina, R., Guerrieri, A.: An analysis of Java-based mobile agent platforms for Wireless Sensor Networks. Multi-Agent and GRID Systems 7(6), 243–267 (2011)
3. Aiello, F., Fortino, G., Gravina, R., Guerrieri, A.: A Java-Based Agent Platform for Programming Wireless Sensor Networks. The Computer Journal 54(3), 439–454 (2010)
4. Bellifemine, F., Fortino, G., Giannantonio, R., Gravina, R., Guerrieri, A., Sgroi, M.: SPINE: a domain-specific framework for rapid prototyping of WBSN applications. Software Practice & Experience 41, 237–265 (03 2011), http://dx.doi.org/10.1002/spe.998
5. Bellifemine, F., Rimassa, G.: Developing multi-agent systems with a FIPA-compliant agent framework. Softw. Pract. Exper. 31, 103–128 (February 2001), http://dx.doi.org/10.1002/1097-024X(200102)31:2¡103::AID-SPE358¿3.0.CO;2-O
6. Bölöni, L., Jun, K., Palacz, K., Sion, R., Marinescu, D.C.: The Bond Agent System and Applications. In: Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents. pp. 99–112. ASA/MA 2000, Springer-Verlag, London, UK, UK (2000), http://dl.acm.org/citation.cfm?id=647629.732585

7. Davidsson, P., Boman, M.: A Multi-Agent System for Controlling Intelligent Buildings. In: Proceedings of the Fourth International Conference on MultiAgent Systems (ICMAS-2000). pp. 377–. IEEE Computer Society, Boston, MA, USA (2000), http://dl.acm.org/citation.cfm?id=518904.878902
8. Davidsson, P., Boman, M.: Distributed monitoring and control of office buildings by embedded agents. Information Sciences-Informatics and Computer Science: An International Journal - Special issue: Intelligent embedded agents 171, 293–307 (05 2005), http://dl.acm.org/citation.cfm?id=1077829.1077831
9. Domanski, J., Dziadkiewicz, R., Ganzha, M., Gab, A., M.M., M.: Implementing GliderAgent - an agent-based decision support system for glider pilots. In: NATO ASI Book, vol. to appear. IOS press (2012)
10. EasySen LLC: WiEye - Sensor board for wireless surveillance and security (2011), [Online]. Available: http://www.easysen.com/WiEye.htm (current December 2011)
11. Essaaidi, M., Fortino, G.: Wireless Sensor Networks and Software Agents. In Software Agents, Agent Systems and their Applications (M. Essaidi, M. Paprizicky and M. Ganzha, Eds.), Information and Communication Security Vol. 32., Chapter 3. IOS press., vol. 32 (2012)
12. Fortino, G., Galzarano, S.: On the development of mobile agent systems for wireless sensor networks: issues and solutions. In Multiagent Systems and Applications: Practice and Experience. Maria Ganzha and Lakhmi Jain Eds. Studies in Computational Intelligence, Springer-Verlag (2012)
13. Fortino, G., Nigro, L.: Development of virtual data acquisition systems based on multimedia internetworking. Computer Standards & Interfaces 21, 429–440 (1999)
14. Fortino, G., Nigro, L.: A toolset in Java2 for modelling, prototyping and implementing communicating real-time state machines. Microprocessors and Microsystems 23, 573–586 (2000)
15. Guerrieri, A., Fortino, G., Ruzzelli, A., O'Hare, G.: A WSN-based Building Management Framework to Support Energy-Saving Applications in Buildings. In Advancements in Distributed Computing and Internet Technologies: Trends and Issues, Chapter 12. Hershey, PA, USA: IGI Global (2011)
16. Hagras, H., Callaghan, V., Colley, M., Clarke, G.: A hierarchical fuzzy-genetic multi-agent architecture for intelligent buildings online learning, adaptation and control. Inf. Sci. Inf. Comput. Sci. 150, 33–57 (3 2003), http://dl.acm.org/citation.cfm?id=763284.763288
17. Huberman, B.A., Clearwater, S.H.: A Multi-Agent System for Controlling Building Environments. In: Lesser, V.R., Gasser, L. (eds.) Proceedings of the International Conference on Multiagent Systems (ICMAS-95). pp. 171–176. The MIT Press (1995)
18. Jiang, X., Dawson-Haggerty, S., Dutta, P., Culler, D.: Design and implementation of a high-fidelity AC metering network. In: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks. pp. 253–264. IPSN '09, IEEE Computer Society, Washington, DC, USA (2009), http://dl.acm.org/citation.cfm?id=1602165.1602189
19. Lopes, R., Assis, F., Montez, C.: MASPOT: A Mobile Agent System for Sun SPOT. In: Proceedings of the 2011 Tenth International Symposium on Autonomous Decentralized Systems. pp. 25–31. ISADS '11, IEEE Computer Society, Washington, DC, USA (2011), http://dx.doi.org/10.1109/ISADS.2011.10
20. Luck, M., McBurney, P., Preist, C.: A Manifesto for Agent Technology: Towards Next Generation Computing. Autonomous Agents and Multi-Agent Systems 9, 203–252 (11 2004)
21. Mobile Agent Platform for Sun SPOT: MAPS (2011), [Online]. Available: http://maps.deis.unical.it (current December 2011)

22. Muldoon, C., O'Hare, G.M.P., Collier, R., O'Grady, M.J.: Agent Factory Micro Edition: A Framework for Ambient Applications. In: Proceedings of Intelligent Agents in Computing Systems Workshop (held in Conjunction with International Conference on Computational Science (ICCS)) Reading, UK. Lecture Notes in Computer Science (LNCS). pp. 727–734. Springer-Verlag Publishers (2006)
23. Naji, H., Meybodi, M., Falatouri, T.: Intelligent building management systems using multi agents: Fuzzy approach. International Journal of Computer Applications 14(6), 9–14 (02 2011), published by Foundation of Computer Science
24. OSGi Alliance: Open System Gateway Initiative (OSGi), documents and software (2011), [Online]. Available: http://www.osgi.org (current December 2011)
25. Qiao, B., Liu, K., Guy, C.: A Multi-Agent System for Building Control. In: Proceedings of the IEEE/WIC/ACM international conference on Intelligent Agent Technology. pp. 653–659. IAT '06, IEEE Computer Society, Hong Kong (2006), http://dx.doi.org/10.1109/IAT.2006.17
26. Stankovic, J.: When sensor and actuator cover the world. ETRI Journal 30(5), 627–633 (2008)
27. Tynan, R., Muldoon, C., O'Grady, M.J., O'Hare, G.M.P.: A mobile agent approach to opportunistic harvesting in wireless sensor networks. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems: demo papers. pp. 1691–1692. AAMAS '08, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2008), http://dl.acm.org/citation.cfm?id=1402744.1402768
28. Zhao, P., Simoes, M., Suryanarayanan, S.: A conceptual scheme for cyber-physical systems based energy management in building structures. In: Proceedings of the 9th IEEE/IAS International Conference on Industry Applications (INDUSCON). pp. 1–6. Sao Paulo, Brazil (11 2010)

Appendix

A. Building Management Events

Table 4. Defined building management events.

Event Name	***Standard Parameters***	***Additional Parameters*** ***<KEY, VALUE>***
BM_SA_ ADVERTISEMENT	ID_SA; ID_CA; Event.MSG_TO_BASESTATION; Event.NOW	<MSG_TYPE, BM_SA_ADVERTISEMENT> <SENSOR_TYPE, VALUE>* <ACTUATOR_TYPE, VALUE>* if exists(<SENSOR_TYPE, VALUE>*) <FUNCTION, VALUE>*
BM_SENSOR_ SCHEDULE	ID_CA; ID_SA; Event.MSG; Event.NOW	<MSG_TYPE, BM_SENSOR_SCHEDULE> <ADDRESSEE_TYPE, VALUE> <ADDRESSEE, VALUE> <REQUEST_ID, VALUE> <PERIOD_TIMESCALE, VALUE> <PERIOD_VALUE, VALUE> <LIFETIME_TIMESCALE, VALUE> <LIFETIME_VALUE, VALUE> <SENSOR_TYPE, VALUE> <DATA_TYPE, VALUE> <SYNTHETIC_DATA_TYPE, VALUE> if DATA_TYPE.VALUE == THRESHOLD_NOTIFICATION <THRESHOLD_TYPE, VALUE> <THRESHOLD_VALUE, VALUE>
BM_ACTUATOR_ SCHEDULE	ID_CA; ID_SA; Event.MSG; Event.NOW	<MSG_TYPE, BM_ACTUATOR_SCHEDULE> <ADDRESSEE_TYPE, VALUE> <ADDRESSEE, VALUE> <REQUEST_ID, VALUE> <PERIOD_TIMESCALE, VALUE> <PERIOD_VALUE, VALUE> <LIFETIME_TIMESCALE, VALUE> <LIFETIME_VALUE, VALUE> < ACTUATOR_TYPE, VALUE> <ACTUATOR_PARAM, VALUE>*
BM_ UNSCHEDULE	ID_CA; ID_SA; Event.MSG; Event.NOW	<MSG_TYPE, BM_UNSCHEDULE> <ADDRESSEE_TYPE, VALUE> <ADDRESSEE, VALUE > <REQUEST_ID, VALUE>
BM_ GROUP_ MANAGEMENT	ID_CA; ID_SA; Event.MSG; Event.NOW	<MSG_TYPE, BM_GROUP_MANAGEMENT> <ADDRESSEE_TYPE, VALUE > <ADDRESSEE, VALUE> <MEMBERSHIP_TYPE, VALUE> <MEMBERSHIP_COUNT, VALUE> If MEMBERSHIP_TYPE.VALUE != RESET <MEMBERSHIP_GROUPS, VALUE>
BM_ SA_RESET	ID_CA; ID_SA; Event.MSG; Event.NOW	<MSG_TYPE, BM_SA_RESET> <ADDRESSEE_TYPE, VALUE > <ADDRESSEE, VALUE >
BM_DATA	ID_SA; ID_CA; Event.MSG_TO_BASESTATION; Event.NOW	<MSG_TYPE, BM_DATA> <TIMESTAMP, VALUE> <REQUEST_ID, VALUE> <RESULT, VALUE>
BM_ACK	ID_SA; ID_CA; Event.MSG_TO_BASESTATION; Event.NOW	<MSG_TYPE, BM_ACK> <MSG_TYPE_TO_ACK, VALUE> <ACK_PARAM, VALUE>

Table 5. Additional parameters of the building management events.

Additional Parameter	*Description*	*PREDEFINED VALUEs*
ADDRESSEE_TYPE	The type of event target	SA, List of SAs, GROUP, GROUP_COMPOSITION
ADDRESSEE	The event target	SA+ | ([NOT] G [STO [NOT] G]*)
REQUEST_ID	The unique identifier of a request	*no predefined int value*
PERIOD_VALUE	The period of the request execution	*no predefined int value*
PERIOD_TIMESCALE	The timescale of the period	MSEC, SEC, MIN, HOUR, DAY
LIFETIME_TIMESCALE	The lifetime of the request	MSEC, SEC, MIN, HOUR, DAY
LIFETIME_VALUE	The timescale of the request	*no predefined int value*
SENSOR_TYPE	The specific sensor type	ACC_X, ACC_Y, ACC_Z, HUMIDITY, IR, LIGHT, MAGNETIC_X, MAGNETIC_Y, SOUND, TEMPERATURE, ELECTRICITY, INTERNAL_VOLTAGE
ACTUATOR_TYPE	The specific actuator type	LED
ACTUATOR_PARAM	An actuator parameter	If ACTUATOR_TYPE == LED LED_0_TOGGLE, LED_1_TOGGLE, LED_2_TOGGLE
DATA_TYPE	The data type of sensor readings	SENSED_DATA, THRESHOLD_NOTIFICATION
SYNTHETIC_DATA_ TYPE	The synthetic data type of sensor readings. Data aggregation can be set.	NO_SYNTHETIC (RAW DATA), AVERAGE, MIN, MAX
THRESHOLD_TYPE	The threshold type applied on sensor reading	LOWER, BIGGER, TRANSITION
MEMBERSHIP_TYPE	The type of membership operation	UPDATE, ADD, DELETE, RESET
MEMBERSHIP_COUNT	The counter of the membership configuration sent	*no predefined int value*
FUNCTION	The type of in-node function computed on the sampled data	ELABORATION_AND_THRESHOLD_STANDARD, ELABORATION_STANDARD, THRESHOLD_STANDARD, AVERAGE, MIN, MAX, THRESHOLD_TYPE_LOWER, THRESHOLD_TYPE_BIGGER, THRESHOLD_TYPE_TRANSITION
TIMESTAMP	Timestamp of the transmitted data	*no predefined int value*
RESULT	Transmitted data	*no predefined int value*
MSG_TYPE_TO_ACK	The message type to ack	BM_SENSOR_SCHEDULE, BM_ACTUATOR_SCHEDULE, BM_UNSCHEDULE, BM_GROUP_MANAGEMENT
ACK_PARAM	Type of ack	if MSG_TYPE_TO_ACK == BM_SENSOR_SCHEDULE || BM_ACTUATOR_SCHEDULE || BM_UNSCHEDULE REQUEST_ID.VALUE if MSG_TYPE_TO_ACK == BM_GROUP_MANAGEMENT MEMBERSHIP_COUNT.VALUE

B. SA's MAPS actions

```
A0: addDispatcherRule(msgTypeList());
    firstProcessedEvent=FALSE;
    Event timer = new Event(agent.getId(), agent.getId(), Event.TMR_EXPIRED,
        Event.NOW );
    timerID = agent.setTimer(true, advertisementTime(), timer);
    addDispatcherRule(timer);
A1: Event msg = new Event(agent.getId(), agent.getCAId(), Event.MSG_TO_BASESTATION,
        Event.NOW);
    msg.setParam(ParamsLabel.MSG_TYPE, BM_SA_ADVERTISEMENT);
    setAdvertisementParams(msg);
    agent.send(agent.getId(), agent.getCAId(), msg, true);
A2: if(isMsgForCurrSA(msgEvent.getParam(ParamsLabel.ADDRESSEE),
        msgEvent.getParam(ParamsLabel.ADDRESSEE_TYPE)){
    firstProcessedEvent=TRUE;
    removeDispatcherRule(timer);
    agent.resetTimer(agent.getId(), timerID);
}
A3: msgType = msgEvent.getParam(ParamsLabel.MSG_TYPE);
A4: plane = createSensorPlane(msgEvent.getParam(ParamsLabel.REQUEST_ID),
        msgEvent);
A5: plane = createActuatorPlane(msgEvent.getParam(ParamsLabel.REQUEST_ID),
        msgEvent);
A6: agent.removePlane(msgEvent.getParam(ParamsLabel.REQUEST_ID));
A7: updateMembership(msgEvent);
A8: Iterator i = agent.getPlaneList();
    while(i.hasNext()){
        plane = (Plane)i.next();
        if(plane.getID() != this.getID())
            agent.removePlane(plane.getID());
    }
    firstProcessedEvent=FALSE;
    timer = new Event(agent.getId(), agent.getId(), Event.TMR_EXPIRED, Event.NOW );
    timerID = agent.setTimer(true, advertisementTime(), timer);
    addDispatcherRule(timer);
A9: Event msg = new Event(agent.getId(), agent.getCAId(), Event.MSG_TO_BASESTATION,
        Event.NOW);
    msg.setParam(ParamsLabel.MSG_TYPE, BM_ACK);
    setAckParams(msg);
    agent.send(agent.getId(), agent.getCAId(), msg, true);
```

Fig. 17. The MAPS actions of the SA's Manager plane.

```
A0: storedDataCounter = 0;
    isOneShotRequest = isOneShot(request);
A1: Event sensing = new Event(agent.getId(), agent.getId(),
        request.getParam(ParamsLabel.SENSOR_TYPE), Event.NOW);
    agent.sense(sensing);
    addDispatcherRule(sensing);
A2: Event timer = new Event(agent.getId(), agent.getId(), Event.TMR_EXPIRED,
        Event.NOW );
    period = getPeriodTimer(request);
    lifetime = getLifetimeTimer(request);
    timer.setParam(ParamsLabel.LIFETIME_ELAPSED, "false");
    timerID = agent.setTimer(true, period, lifetime, timer);
    addDispatcherRule(timer);
    dataToStore = getDataToStore(request);
A3: storeData(event.getParam(SENSED_DATA));
    storedDataCounter++;
A4: if(request.getParam(ParamsLabel.DATA_TYPE) != "THRESHOLD_NOTIFICATION" ||
        isThresholdChecked(request, getStoredData())){
        Event msg = new Event(agent.getId(), agent.getCAId(),
            Event.MSG_TO_BASESTATION, Event.NOW);
        msg.setParam(ParamsLabel.MSG_TYPE, BM_DATA);
        setDataParams(msg, getStoredData());
        agent.send(agent.getId(), agent.getCAId(), msg, true);
    }
    storedDataCounter = 0;
A5: syntheticData = calculateSyntheticData(getStoredData(),
        request.getParam(ParamsLabel.SYNTHETIC_DATA_TYPE));
    if(request.getParam(ParamsLabel.DATA_TYPE) != "THRESHOLD_NOTIFICATION" ||
        isThresholdChecked (request.getParams(), syntheticData)){
        Event msg = new Event(agent.getId(), agent.getCAId(),
            Event.MSG_TO_BASESTATION, Event.NOW);
        msg.setParam(ParamsLabel.MSG_TYPE, BM_DATA);
        setDataParams(msg, syntheticData);
        agent.send(agent.getId(), agent.getCAId(), msg, true);
    }
    storedDataCounter = 0;
```

Fig. 18. The MAPS actions of the SA's Sensing Request plane.

Giancarlo Fortino is an Associate Professor of computer engineering at the Department of Electronics, Informatics, and Systems of the University of Calabria, Italy. His research interests include distributed computing, wireless sensor networks, agent-based computing, and real-time systems. He is author of more than 170 papers in international journal, books and conference proceedings. He received a Laurea degree and a PhD in Computer Engineering from the University of Calabria in 1995 and 2000, respectively.

Antonio Guerrieri is a research fellow in Computer Engineering at the University of Calabria. His research interests include high-level programming methods for wireless sensor networks with specific focus on methodologies and frameworks for building sensor networks. He is author of several papers in international journal, books and conference proceedings. He received his Bachelor, Master and PhD in Computer Engineering from the University of Calabria in 2003, 2008, and 2012 respectively.

Received: January 1, 2012; Accepted: May 4, 2012

CIP – Каталогизација у публикацији
Народна библиотека Србије, Београд

004

COMPUTER Science and Information Systems : the International journal / Editor-in-Chief Mirjana Ivanović. – Vol. 9, No 3 (2012) - . – Novi Sad (Trg D. Obradovića 3): ComSIS Consortium, 2012 - (Belgrade : Sigra star). –30 cm

Polugodišnje. – Tekst na engleskom jeziku

ISSN 1820-0214 = Computer Science and Information Systems
COBISS.SR-ID 112261644

Cover design: V. Štavljanin
Printed by: Sigra star, Belgrade

ComSIS Computer Science and Information Systems

Volume 9, Number 3, Special Issue, September 2012

Contents

Editorial

Papers

983 A Systematic Approach to the Implementation of Attribute Grammars with Conventional Compiler Construction Tools
Daniel Rodríguez-Cerezo, Antonio Sarasa-Cabezuelo, José-Luis Sierra

1019 Implementation of EasyTime Formal Semantics using a LISA Compiler Generator
Iztok Fister Jr., Marjan Mernik, Iztok Fister, Dejan Hrnčič

1045 Using Aspect-Oriented State Machines for Detecting and Resolving Feature Interactions
Tom Dinkelaker, Mohammed Erradi, Meryeme Ayache

1075 A MOF based Meta-Model and a Concrete DSL Syntax of IIS*Case PIM Concepts
Milan Čeliković, Ivan Luković, Slavica Aleksić, Vladimir Ivančević

1105 LL Conflict Resolution using the Embedded Left LR Parser
Boštjan Slivnik

1125 Indexing Ordered Trees for (Nonlinear) Tree Pattern Matching by Pushdown Automata
Jan Trávníček, Jan Janoušek, Borivoj Melichar

1155 A Programming Language Independent Framework for Metrics-based Software Evolution and Analysis
Črt Gerlec, Gordana Rakić, Zoran Budimac, Marjan Heričko

1187 High-level Multicore Programming with C++11
Zalán Szűgyi, Márk Török, Norbert Pataki, Tamás Kozsik

1203 Supporting Heterogeneous Agent Mobility with ALAS
Dejan Mitrović, Mirjana Ivanović, Zoran Budimac, Milan Vidaković

1231 Language Engineering for Syntactic Knowledge Transfer
Mihaela Colhon

1249 Implementing an eXAT-based Distributed Monitoring System Prototype
Gleb Peregud, Julian Zubek, Maria Ganzha, Marcin Paprzycki

1287 Modeling a Holonic Agent based Solution by Petri Nets
Carlos Pascal, Doru Panescu

1307 Information Resource Management in an Agent-based Virtual Organization — Initial Implementation
Maria Ganzha, Adam Omelczuk, Marcin Paprzycki, Mateusz Wypysiak

1331 Decentralized Management of Building Indoors through Embedded Software Agents
Giancarlo Fortino, Antonio Guerrieri

ISSN: 1820-0214

Computer Science and Information Systems

Vol 9, No 3, September 2012