Com
SIS

Computer Science and Information Systems

Vol 8, No 2, May 2011

Computer Science and Information Systems

Published by ComSIS Consortium

Special Issue on Advances in Formal Languages, Modeling and Applications

Volume 8, Number 2
May 2011

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 8, Number 2, 2011
Novi Sad

Computer Science and Information Systems

Special Issue on Advances in Formal Languages, Modeling and Applications

ISSN: 1820-0214

ComSIS Journal is sponsored by:

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

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 surveys 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 Expanded (also known as SciSearch) and Journal Citation Reports / Science Edition by Thomson Reuters,
- 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,
- Center for Evaluation in Education and Science and Ministry of Science of Republic of Serbia (CEON) in cooperation with the National Library of Serbia,
- 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 one of the Editors.

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 30 (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 8, Number 2, Special Issue, May 2011

CONTENTS

Editorial

Invited Papers

225 **Challenges and Directions in Formalizing the Semantics of Modeling Languages**
Barrett R. Bryant, Jeff Gray, Marjan Mernik, Peter J. Clarke, Robert B. France, Gabor Karsai

255 **Software Agents: Languages, Tools, Platforms**
Costin Bădică, Zoran Budimac, Hans-Dieter Burkhard, Mirjana Ivanović

Regular Papers

299 **SPEM Ontology as the Semantic Notation for Method and Process Definition in the Context of SWEBOK**
Miroslav Líška, Pavol Navrat

317 **Ontology Driven Development of Domain-Specific Languages**
Ines Čeh, Matej Črepinšek, Tomaž Kosar, Marjan Mernik

343 **Domain-Specific Language for Coordination Patterns**
Nuno Oliveira, Nuno Rodrigues, Pedro Rangel Henriques

361 **From DCOM Interfaces to Domain-Specific Modeling Language: A Case Study on the Sequencer**
Tomaž Kos, Tomaž Kosar, Jure Knez, Marjan Mernik

379 **A DSL for PIM Specifications: Design and Attribute Grammar based Implementation**
Ivan Luković, Maria João Varanda Pereira, Nuno Oliveira, Daniela da Cruz, Pedro Rangel Henriques

405 **UML Profile for Specifying User Interfaces of Business Applications**
Branko Perišić, Gordana Milosavljević, Igor Dejanović, Branko Milosavljević

427 **Formalizing Business Process Specifications**
Andreas Speck, Sven Feja, Sören Witt, Elke Pulvermüller, Marcel Schulz

447 **An Approach to Assess and Compare Quality of Security Models**
Raimundas Matulevičius, Henri Lakk, Marion Lepmets

477 **GammaPolarSlicer**
Sérgio Areias, Daniela da Cruz, Pedro Rangel Henriques, Jorge Sousa Pinto

501 **Animation of Tile-Based Games Automatically Derived from Simulation Specifications**
Jan Wolter, Bastian Cramer, Uwe Kastens

517 **Solving Difficult LR Parsing Conflicts by Postponing Them**
C. Rodriguez-Leon, L. Garcia-Forte

533 **Detecting Concurrency Anomalies in Transactional Memory Programs**
João Lourenço, Diogo Sousa, Bruno Teixeira, Ricardo Dias

EDITORIAL

Everyone recognizes the importance of formal modeling activities as support for software development. These models must use an appropriate abstraction level in order to describe systems without unnecessary details. The main objective is to describe the domain (concepts and relations) and the actions using a formal representation. For that, ontologies have been used and domain specific languages (DSL) have been created. The implementation of new DSLs implies the construction of new language-based tools. This can be done automatically using traditional language processing techniques based on grammars.

This special issue contains revised and expanded versions of selected high quality papers presented either at the Conference on Compilers, Programming Languages, Related Technologies and Applications (CoRTA'2010) or at the Fourteenth East-European Conference on Advances in Databases and Information Systems (ADBIS 2010) or at the workshop integrated in this last one called MDASD - Model Driven Approaches in System Development. One of the invited papers comes from the International Workshop on Formalization of Modeling Languages (FML 2010).

The first conference, organized by the University of Minho, took place on September 09-10, 2010, in Braga, Portugal. The latter was organized in co-operation with Faculty of Sciences and Faculty of Technical Sciences from University of Novi Sad and took place on September 20 - 24, 2010, Novi Sad, Serbia. FML 2010 was collocated with ECOOP 2010 and took place on June 21, 2010, in Maribor, Slovenia.

CoRTA2010 (4th edition) is a forum where researchers, developers, educators present their work under development and exchange new ideas and information about language design, processing, assessment, and application (http://corta2010.di.uminho.pt/). CoRTA2010 was a track under INForum -- Simpósio de Informática: http://inforum.org.pt/INForum2010.

ADBIS 2010 provided a forum for the dissemination of research work and it promoted the interaction and collaboration between the database and information systems research communities. MDASD workshop (http://perun.pmf.uns.ac.rs/adbis2010/workshop-mdasd.php) gathered researchers working on Model-Driven (MD) languages, techniques and tools, as well as DSLs. These techniques were applied in information system and application development, databases and related areas. So, MDASD participants could exchange their experience, discuss new ideas and evaluate and improve MD approaches.

The purpose of FML 2010 was to provide a forum to discuss challenge areas in formalizing modeling languages and constructing automated tools from such formalizations (http://www.cis.uab.edu/FML2010/).

There are several common topics between CoRTA, MDASD, ADBIS and FML like:

- Modeling languages and model-driven development;
- Automatic generation of modeling tools;
- Specification of domain specific languages;
- Ontologies as a formal model to develop languages and tools;
- Paradigms, concepts, methodologies, and novel constructs on programming languages;
- Language quality assessment and grammar metrics.

At the end of all these events, some of the high quality submitted papers were selected for possible publication in a special issue of ComSIS and their authors were invited to prepare extended versions of their papers. These extended versions were then reviewed by experts in the field. In order to ensure the best possible quality, the improved papers were then submitted to a second round of reviewing.

This special issue includes two invited papers. The first one “Challenges and Directions in Formalizing the Semantics of Modeling Languages” by Barrett Bryant, Jeff Gray, Marjan Mernik, Peter Clarke, Robert France and Gabor Karsai, discuss semantics-based approaches for the formalization of modeling languages and the automatic construction of modeling tools. The tools for model-driven engineering are usually based on manual processes. In order to automatize their development it is necessary to formalize the modeling language underlying. A semantics-based approach for that purpose is described in this paper.

“Software Agents: Languages, Tools, Platforms” by Costin Badica, Zoran Budimac, Hans-Dieter Burkhard and Mirjana Ivanović is the second invited paper. It presents a very complete survey about agent-based software development. Significant languages, tools, platforms and development activities are described.

The first regular paper has the title “SPEM Ontology as the Semantic Notation for Method and Process Definition in the Context of SWEBOK” and the authors are Miroslav Liška and Pavol Návrat. This paper presents a meta-model ontology that adds a semantic notation for software process definition. This approach provides concepts for knowledge based software process engineering and it was applied to several case studies.

“Ontology Driven Development of Domain-Specific Languages” by Ines Čeh, Matej Črepinšek, Tomaž Kosar and Marjan Mernik is the second regular paper and it explains a new approach to develop DSLs where the design phase is based on ontologies. The main idea is to present a set of rules that allow the automatic translation of ontology concepts into grammar symbols. Preliminary results of the Ontology2DSL framework are presented.

The next paper “Domain-Specific Language for Coordination Patterns” by Nuno Oliveira, Nuno Rodrigues, Pedro Rangel Henriques presents the development of a new DSL to specify coordination patterns of system components. The main objective is to separate the architecture decision features from other code in order to improve the system comprehension process.

In “From DCOM Interfaces to Domain-Specific Modeling Language: A Case Study on the Sequencer” by Tomaž Kos, Tomaž Kosar, Jure Knez and Marjan Mernik a domain-specific modeling language developed to enable domain experts to program or model their own measurement procedures is described. This approach turns them independent from the measurement system producers. Some experiences were performed to prove the effectiveness of this new domain-specific modeling language.

The paper “A DSL for PIM Specifications: Design and Attribute Grammar based Implementation” by Ivan Luković, Maria João Varanda Pereira, Nuno Oliveira, Daniela da Cruz and Pedro Rangel Henriques, presents the design and implementation of a new DSL that will allow to specify easily Platform Independent Model (PIM) specifications. These specifications are used by a model driven software tool (IIS*Case) for system modeling and prototype generation.

The next paper “UML Profile for Specifying User Interfaces of Business Applications” by Branko Perišić, Gordana Milosavljević, Igor Dejanović and Branko Milosavljević, presents a DSL-based approach to perform automatic generation of user interfaces. Since the new DSL created is based on UML, it can be easily integrated with other application UML models.

“Formalizing Business Process Specifications” by Andreas Speck, Sven Feja, Sören Witt, Elke Pulvermüller, Marcel Schulz addresses formalisms to specify business processes in particular the use of temporal logic. In order to avoid the complexity of temporal logic, a graphical notation is proposed and semantic specializers are used to identify and check element types.

Raimundas Matulevicius, Henri Lakk and Marion Lepmets in “An Approach to Assess and Compare Quality of Security Models” describe an approach to perform a comparative study of security models in order to assess their quality.

In the paper “GrammaPolarSlicer”, the authors SérgioAreias, Daniela da Cruz, Pedro Rangel Henriques and Jorge Sousa Pinto propose an approach to ensure the software quality on reusing processes. The paper introduces the concept of “caller-based slicing” as a way to certify the integration of components into legacy systems and proposes the use of visualization techniques to improve the approach.

“Animation of Tile-Based Games Automatically Derived from Simulation Specifications” by Bastian Cramer, Jan Wolter and Uwe Kastensis is concerned with the implementation of visual languages using Devil. When the language has an execution semantics the gap between program depiction and program execution disappears. This tool allows the rapid development of simulations and animations based on several kinds of visual languages: diagrammatic, iconic and graph based ones.

Concerning the Language Processing subject, a new technique to solve LR parsing conflicts is presented by Luis Garcia-Forte and Casiano Rodriguez-Leonin “Solving Difficult LR Parsing Conflicts by Postponing Them”. The main idea is to avoid grammar modifications when reduce-reduce conflicts are detected. The solution is based on a postponing approach.

The paper “Detecting Concurrency Anomalies in Transactional Memory Programs” by João Lourenço, Diogo Sousa and Bruno Teixeira presents a framework for detection of both low and high-level anomalies in transaction memory programs. The tool was applied to a set of programs and the authors proved the effectiveness of using static analysis in the process of anomalies identification.

On behalf of the Editorial Board and the ComSIS Consortium, we would like to thank the authors for their high-quality contributions, and also the reviewers for the effort and time invested into the preparation of this issue of Computer Science and Information Systems journal.

Ivan Luković,
Mirjana Ivanović and
Maria João Varanda Pereira
Guest Editors

DOI:10.2298/CSIS110114012B

Challenges and Directions in Formalizing the Semantics of Modeling Languages

Barrett R. Bryant[1], Jeff Gray[2], Marjan Mernik[3], Peter J. Clarke[4], Robert B. France[5], and Gabor Karsai[6]

[1] Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, Alabama 35294-1170, USA
bryant@cis.uab.edu
[2] Department of Computer Science, University of Alabama
Tuscaloosa, Alabama 35487-0290, USA
gray@cs.ua.edu
[3] Faculty of Electrical Engineering and Computer Science, University of Maribor
2000 Maribor, Slovenia
marjan.mernik@uni-mb.si
[4] School of Computing and Information Sciences, Florida International University
Miami, Florida 33199, USA
clarkep@cis.fiu.edu
[5] Computer Science Department, Colorado State University
Fort Collins, Colorado 80523-1873, USA
france@cs.colostate.edu
[6] Institute for Software-Integrated Systems, Vanderbilt University
Nashville, Tennessee 37235, USA
gabor.karsai@vanderbilt.edu

Abstract. Developing software from models is a growing practice and there exist many model-based tools (e.g., editors, interpreters, debuggers, and simulators) for supporting model-driven engineering. Even though these tools facilitate the automation of software engineering tasks and activities, such tools are typically engineered manually. However, many of these tools have a common semantic foundation centered around an underlying modeling language, which would make it possible to automate their development if the modeling language specification were formalized. Even though there has been much work in formalizing programming languages, with many successful tools constructed using such formalisms, there has been little work in formalizing modeling languages for the purpose of automation. This paper discusses possible semantics-based approaches for the formalization of modeling languages and describes how this formalism may be used to automate the construction of modeling tools.

Keywords: model-based tools, modeling languages, semantics.

1. Introduction

With increasing frequency, scientists and engineers in diverse areas of focus, as well as end-users with specific domain expertise, are requiring computational processes to allow them to complete some task (e.g., avionics engineers who seek input on a modeled design from verification tools, or geneticists who need to describe computational queries to process a gene expression). A challenge emerges from the lack of knowledge of such users in terms of expressing their computational desire (i.e., such users typically are not familiar with programming languages). Model-driven engineering (MDE) is an approach that provides higher levels of abstraction to allow such users to focus on the problem, rather than the specific solution or manner of realizing that solution through lower level technology platforms [46][52]. However, the potential impact of modeling is reduced due to the imprecise nature in which modeling languages are defined [26]. The large majority of modeling languages are defined in an ad hoc manner that lacks precision and a common reference definition for understanding the meaning of language concepts. In current practice, the meaning of a modeling language is often contained only in a model translator (we will use the term *model interpreter* in this paper to refer to such translators) that converts a model representation into some other form (e.g., source code). The current situation in MDE is not unlike the early period of computing when the definition of a programming language was delegated to "what the compiler says it means." Such an approach not only promotes misunderstanding of the meaning of a modeling language, but also limits opportunities for automating the generation of various language tools (much like the adoption of grammars provided a reference point for compiler and other tool generation for a programming language).

The advantages of formal specification of programming language semantics are well-known. First, the meaning of a program is precisely and unambiguously defined; second, it offers a unique possibility for automatic generation of language-based tools (e.g., [27]). Unfortunately, formal specifications, syntax and semantics, of modeling languages have not been developed to this level yet. Although the syntax of modeling languages is commonly specified by metamodels, an appropriate and standard formalism for specifying the (behavioral) semantics of modeling languages does not yet exist. Hence, there is no automatic generation of model interpreters, debuggers, simulators and verification tools.

In this paper, we describe challenges and directions in formalizing the semantics of modeling languages. The ideas developed in this paper were derived from the *Workshop on Formalization of Modeling Languages* held in conjunction with the *European Conference on Object-Oriented Programming (ECOOP)* in Maribor, Slovenia, on June 21, 2010. The paper is organized as follows. Section 2 motivates the need for semantics in modeling languages and reviews existing work in this area. In Section 3, we describe an approach based on state machine models. Section 4 describes a metamodel-based approach to semantics. In Sections 5 and 6, we discuss our experiences with semantics-based modeling tools for verification. Finally, we conclude in Section 7.

2. The Need for Semantics in Modeling Languages

Much of the success of MDE is dependent on the descriptive power of domain-specific modeling languages (DSMLs) [24][29][50]. One of the current challenges of adopting a DSML is the lack of a precise description of the semantics of the DSML. Initial attempts are described in [9], [10] and [16]. The typical technique for specifying the syntax and static semantics of a DSML is to use a metamodel, which describes concepts in a problem domain and their relationships. A standard known as MOF (Meta-Object Facility) has been proposed for defining the syntax of modeling languages by following a similar role as BNF and its variants (e.g., EBNF) for programming languages. Metamodels are currently even used for specifying the syntax of domain-specific programming languages [42]. However, the situation concerning syntactical description of languages is completely different from semantics. It is often easier to describe the structure of a DSML using a metamodel than it is to specify the syntax of a programming language using BNF. However, specifying detailed behavior (semantics) is much harder with DSMLs. In our opinion, this is why only the syntax of current DSMLs are formally described, but the semantics are left toward other less than desirable means. For example, as will be discussed further in Section 5, the semantics of the UML (Unified Modeling Language) metamodel is defined using a mixture of OCL (Object Constraint Language) and informal text, which is clearly unacceptable for formal analysis. Hence, the meaning (semantics) of models are often not formally described. For this purpose, general-purpose programming languages (e.g., C++) are often used to define model interpreters that have an internal representation of the semantics of a DSML. The lack of a formal definition of DSML semantics contributes to several problems, as highlighted in the following paragraphs.

Tool Generation Challenges: The semantics of DSMLs are not defined formally. Hence, proving properties about concepts and relationships in the domain is not possible. Moreover, a model interpreter cannot be automatically generated in most cases. A further consequence is that various other model-based tools (e.g., debuggers, test engines, simulators, verifiers) also cannot be generated automatically.

Tool Analysis Challenges: Model interpreters are often implemented with general-purpose programming languages (GPLs). This has several consequences. Verifying a model interpreter is a very difficult, if not impossible task. As such, verification, optimization, and parallelization of models can be expressed only through GPLs.

Formal Language Design: DSMLs are also languages that need to be designed properly. This leads to several key questions: What are the design principles for modeling languages? How are the results of domain analysis used in modeling language design?

Modeling Language Composition: In practice, multiple domains might be involved to describe different perspectives of a modeled system. In such a case, there is a need for composing DSMLs together. Presently, there is little support for formal composition and evolution of DSMLs.

2.1. Related Work in Modeling Language Definition

Some work on the generation of various modeling tools has already been investigated. Different approaches to the issue of defining the semantics of DSMLs have been proposed; these differ in their applicability and potential of leveraging automatic or at least semi-automatic language tool generation.

2.2. Mapping the DSML into Existing Formal Languages

A common way of defining the semantics of a modeling language is through *translation semantics*, where the abstract syntax of the main DSML is mapped into the abstract syntax of an existing formal language, with well-defined and understood semantics. The mapping is achieved through model transformations. An advantage of this approach is that the DSML can convey existing tools of the language into which it is translated. A common critique of this approach is that since the semantics definition is not defined in the metamodel of the DSML, it is very challenging to correctly map the constructs of the DSML into the constructs of the target language. The underlying cause for this is that the mappings are not at the same level of abstraction and the target language may not have a simple mapping from the constructs in the source language. Another issue of the translation semantics approach is the mapping of execution results (e.g., error messages, debugging traces) back into the DSML in a meaningful manner, such that the domain expert using the modeling language understands the result.

One concrete approach that uses translation semantics is called *semantic anchoring* [9], which uses the well-known Abstract State Machines (ASM) formalism [7] to define the semantics. We will discuss the technique in detail below. This solution maps the abstract syntax of the DSML, which was defined in the GME (Generic Modeling Environment) metamodeling tool [33], into well-established semantic domains, called semantic units (e.g., timed automata, and discrete event systems) that have been defined in the ASML (Abstract State Machine Language) tool. The initial work on semantic anchoring did not show any application of tool generation from the semantics specification, although the usage of ASML enables compilation, simulation, test case generation and verification of ASML specifications, as will be discussed further in Section 3. A similar concrete approach was proposed by Di Ruscio et al. [16], which also did not demonstrate any tool generation based on the semantics definition. Gargantini, Riccobene and Scandurra [22] introduce a semantic framework based on ASM, which also includes three translational semantics techniques: semantic mapping, semantic hooking and semantic meta-hooking. The authors do not demonstrate any tool generation from their semantics specifications. The Moses tool suite [21], which defines the syntactical aspects (e.g., vertex edge/types, syntactical predicates) of the language with a Graph Type Definition Language (GTDL), uses ASM for prototyping model interpreters to achieve the definition of semantics. Based on this kind of formal specification, the Moses tool suite generates animation and debugging tools for visual models. The work presented in [43] describes a translation semantics definition with Maude, which is a rewriting logic-based language. Based on such a semantics definition simulation, reachability and model-checking analysis tools can be generated. Sadilek and Wachsmuth [44] present a semantics definition based on a transition system, where the states are defined by metamodel instances and the transitions are defined by model transformations. The work of Hahn [25] uses the Object-Z language [48] as the means of defining the translation semantics.

2.3. Weaving the Semantics into the Metamodel

Another approach is to *weave behavior* into the abstract syntax (i.e., the metamodel) by a meta-language (also called action language), which can be used to specify the bodies of operations that occur in the metamodel. This permits the model to be executable, because the semantics are defined inside the operation bodies. The significant drawback of this approach is the fact that some meta-languages are very similar to 3rd generation programming languages; therefore, they have to be used in an operative way. The advantage of this approach is the fact that this kind of semantics specification can be mastered by most domain experts.

A well-known representative of this approach is the Kermeta tool [40], which extends an abstract meta-layer with an imperative action language to weave a semantic definition within the metamodel. Kermeta constructs contain specification of operations over metamodel elements. The built-in support for specification of operational semantics enables the automatic generation of simulation and testing tools. Another example is the approach proposed by Scheidgen and Fischer [45], where an operation is specified through the use of OCL statements and an activity diagram. The graphical format of this meta-language is particularly familiar to users with a strong modeling background. The authors mentioned that in the future they will work on automatic debugger generation. Soden and Eichler [49] propose a similar approach based on the usage of activity diagrams as the meta-language. Their future work will be implemented in a framework known as the Model Execution Framework (MXF) and should take an important place in the Eclipse environment. Based on the semantics definition, various tools like trace analysis and runtime verification will be automatically generated. The Mosaic XMF framework [3], which uses an extended OCL language to provide semantics, is another representative of the semantics definition approach. Initial work that corresponds to the behavior weaving approach was also undertaken in UML [51], where action semantics were proposed to achieve the goal of executable UML models. To define the semantics of a new language, no notation was enforced, but the authors "suggest activities with action semantics for language modelling." Ducasse et al. [17] use Smalltalk as a meta-language in their DSML semantics definition.

2.4. Defining the Semantics with Rewrite Rules

Semantics also can be specified through *rewriting systems,* where the system typically consists of rewrite rules. Each rewrite rule consists of a left- and a right-hand side. The execution of a rewrite system is based on the repeated application of the rewrite rules to an existing configuration (e.g., model). A rule is applied when the left-hand side of the rule is found in the configuration, in such a way that this occurrence will be replaced by the right-hand side of the rule. The execution is complete when there is no rule that can be applied to the configuration. Typically, the existing approaches employ graph rewriting where the semantics can be specified in an operational fashion through the graphical definition given by graph grammars. Graph rewriting provides a mathematically precise and visual specification technique by combining the advantages of graphs and rules into a single computational paradigm [53].

Graph rewriting specification was employed in the AToM3 tool [32], which uses triple graph grammars as rewriting rules. One of the interesting features of AToM3 is that the definition of rewriting rules is given through concrete syntax, which makes semantic specification especially amenable for domain-experts. AToM3 can use graph grammar definitions to generate visual model simulators and implement model optimizations and code generation. The dynamic metamodeling [19] approach describes the semantics of UML behavior diagrams with collaboration diagrams, which are used in graph transformations. The authors mention future work on the generation of model simulators. Ermel et al. [20] enable translation of UML behavior diagrams into graph transformations, which are the basis for semantics that are used to generate a visual simulator of UML models.

2.5. Other Approaches to the Definition of Semantics

There also exist other examples of generating tools from semantic definitions that are described in GPLs. Perhaps a valuable lesson can be learned even from these examples. One of the most well-known approaches is Ptolemy [18], which is a tool that enables animated interpretation of hierarchically composed domain-specific models. Models in Ptolemy consist of heterogeneous domains (models of computation) that can have different semantics. Adding a new DSML to Ptolemy is cumbersome, because the syntax and semantics have to be defined manually (i.e., hand-coded) in Java by implementing a "director" that assigns executable semantics to the DSML constructs.

3. Defining the Semantics of Modeling Languages

We view semantics as a mapping from the abstract syntax (A) of the DSML to some semantic domain (D). The abstract syntax defines the fundamental modeling concepts, their relationships, and attributes used in the DSML, and the semantic domain is some mathematical framework whose meaning is well-defined. The abstract syntax defines the data structures that represent the modeling constructs, and, as such, it can be considered as a schema for the models. For instance, in a modeling language representing Finite State Machines (FSMs), we will need data structures for states and transitions, which need to be related to each other such that one can find the source and target states of transitions. Instances of such data structures do represent FSMs, and algorithms are available to analyze them. The concrete syntax (S) is the human-readable manifestation of the abstract syntax. In our FSM example, the concrete syntax can be textual (e.g., a simple language where an FSM is represented as a set of names for states, and a set of transitions represented in the form 'state1 → state2', where state1 and state2 are names of states), or it can be graphical (e.g., a graphical notation with bubbles representing states and arrows connecting bubbles representing the transitions). There is always a well-defined mapping between A and S. We use the concrete syntax to create and modify the models, with the assistance of a customized metamodeling tool, such as the GME. Note that changes on the models performed using the concrete syntax must eventually be reflected as changes in the abstract syntax form of the models.

An example for the visual depiction of abstract syntax is shown in Figure 1, which uses the UML class diagram graphical formalism. The abstract syntax is that of a Stateflow-style [36] hierarchical state machine, with `States` and `Transitions` being the main elements. The top-level model element `Stateflow` is a `Folder` that acts as a container for models. This container will contain `States` that contain other `States` and `Transitions`. The recursive containment of states within states allows the composition of hierarchical state machines. Transitions connect `TransConnectors` that are abstract (only their derived classes can be instantiated), and that could be `States`, `Junctions`, initial transitions (`TransStart`), history junctions (`History`), or references (`ConnectorRef`) that point to other `TransConnectors`. States may also contain Data or Event elements, as well as an optional reference to a data type (`TypeBaseRef`). Note that this composition expressed as abstract syntax follows the legal composition of model elements available in the Stateflow language. For example, a `Transition` cannot connect a `Data` element to a `State` – there is no legal association between them.

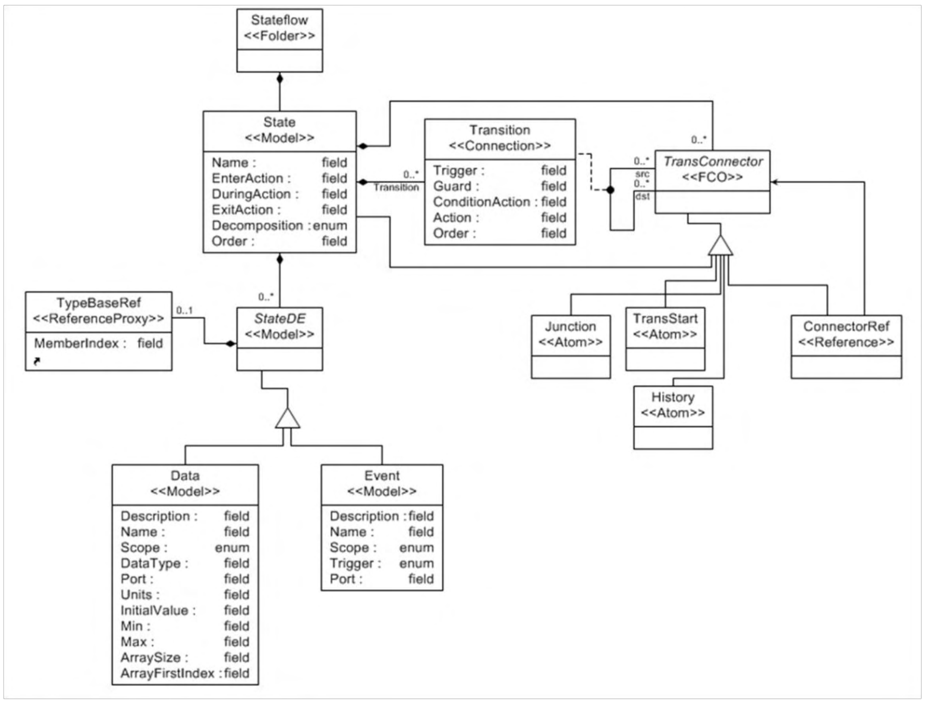


Fig. 1. Abstract syntax for a DSML representing a Hierarchical FSM

One can also define well-formedness constraints (*C*) over the abstract syntax. In our example, a well-formedness constraint could specify that there must be precisely one state marked as “initial” among the states contained in a Stateflow model, and the sub-states of a state. Such constraints delineate what models are considered ‘correct’ with respect to a static notion of semantics; the constraints can be checked on the models directly, without referring to a semantic domain.

The semantic domain for such a DSML could be a finite state *machine (M)* (implemented in hardware or software), with a finite set of *states* (with precisely one, distinguished state called the *initial state*), a finite set of triggering *events*, and state *transitions* between states. Transitions are labeled with triggering events and Boolean guard expressions over some variables of the system. A model expressed in the DSML compliant with the abstract syntax will map to a specific machine that operates as follows: The machine is always in a specific state, called the *current* state. When the execution starts, the current state is the initial state. When an event arrives, it is matched against the event labels attached to transitions emanating from the current state, and if a matching label is found the transition is selected. The guard for the selected transition is evaluated, and if it is true then the current state becomes the target state the transition points to. If the event does not match any event on an outgoing transition (or if it does match, but the guard is false), the current state does not change. It is required that if multiple transitions are selected, at most one guard can be true, otherwise the behavior is non-deterministic and the model is incorrect. Note that this machine does not have hierarchical states.

The semantics of the models can be defined by a mapping $m : A \rightarrow D$ that instantiates a specific (non-hierarchical) finite state machine from a model. After the machine is created, it operates in some environment according to the algorithm described above. Note that the semantics is *ultimately* defined by our understanding of how the machine works: although it can be formalized, it is still dependent on our (possibly inaccurate) understanding of the operation of the machine. After this understanding is refined, we can 'build' it as a digital circuit or as a software simulator. Note that the semantic domain defines the meaning of a model with respect to a dynamic notion of semantics; one needs a "machine" to execute the computation denoted by the model.

Note that not all DSMLs have an executable (or 'operational') semantics. For instance, UML class diagrams are not 'executable,' however, they can be expressed in various forms (e.g., C++ code consisting of classes with data members and member functions). Some DSMLs have very weak opportunity for semantics definition; for instance, UML use-case diagrams can only be paraphrased in a natural language, without any formal mapping. Below, we restrict the discussion to DSMLs that do have executable semantics.

Drawing from the example, we can observe that the specification of semantics may be accomplished in two steps: (1) defining the 'semantic domain,' and (2) defining the mapping between the abstract syntax and the constructs of the semantic domain. For a pragmatic approach one can envision a translator for (2), and a simulator (or interpreter) for (1) that interprets the result of (2) with some input. Below we describe two variations on how these steps can be accomplished.

3.1. Definition via a Semantic Unit

Assume we have well-defined, accepted, and well-understood modeling languages whose semantics are simple and defined in a non-ambiguous, preferably executable way. Let's call these core modeling languages *semantic units*. An example of a semantic unit could be the domain of simple finite state machines, as described in the previous section. If a new DSML needs to be defined, one has to specify the semantics of this new language by showing how the models built in the new language could be reduced to (or transformed into) the well-defined semantic units. The principle is illustrated in Figure 2.

In this method, the semantics are mainly defined by the transformation $M_{DSMLi,SU}$ that maps the abstract syntax of the DSML (A of DSML-i) to the abstract syntax of a semantic unit (A of SU). The concrete syntax (C) of the DSML is related to the abstract syntax of the DSML (A) via a mapping (M_{Ci}). The semantic domain of the DSML is some S, and the notional semantics of the DSML is defined via the mapping M_{Si}. The key idea here is that we define the M_{Si} mapping in two steps: (1) the transformation ($M_{DSMLi,SU}$), and (2) the semantic mapping of the semantic unit M_{SU}. Note that semantic units also have a DSML: a concrete syntax (C of SU), a semantic domain (S of SU), an abstract syntax (A of SU), and the mappings: M_{C2} for the syntax and M_{SU} for the semantics. The base semantic domain is much simpler than a higher level DSML. The transformation can be specified formally, for instance in the form of graph transformation rules [2], which represent how to rewrite a higher level DSML into the lower level DSML; hence, establishing a formal, yet executable mapping between the two languages. For the example described above, the transformation rewrites the hierarchical, Statechart-like state machine into a flat, non-hierarchical state machine.

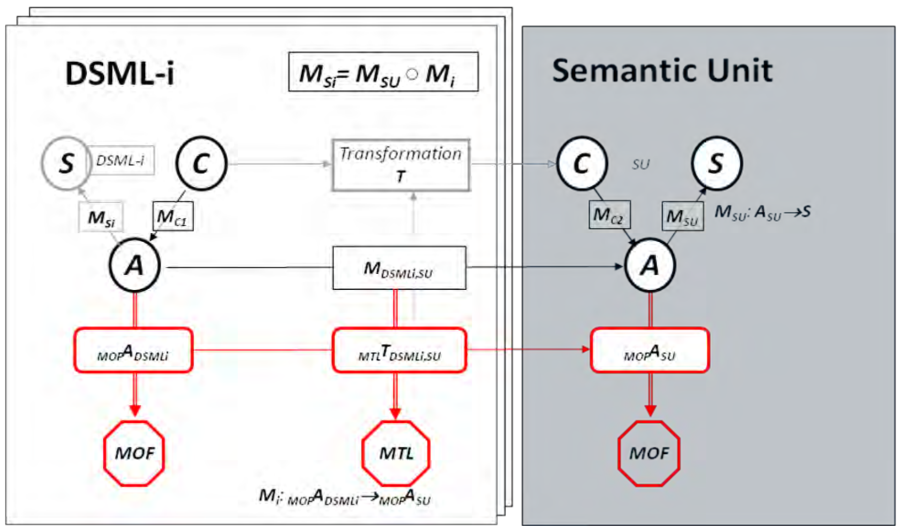


Fig. 2. Defining semantics via a transformation and a semantic unit

For specifying the semantic unit, a tool has been created that uses the Abstract State Machine Language (ASML) [38] to represent semantic units. ASML allows building these semantics units using the Abstract State Machine concepts [7] (i.e., essentially as transition systems with sophisticated data structures representing the state of the system). A number of prototype model transformations have been built that show how a non-trivial DSML (e.g., a Statechart-like language) can be formally defined via the transformation [9]. These form the initial components of a tool suite where one can define the abstract syntax of a language, together with its semantics using semantic units and transformations. An interesting property of ASML is that it is executable, thus one can rapidly prototype and experiment with DSMLs by executing their models as ASML “programs.”

In this approach, the main complexity is in the model transformation process, and semantic units are typically simple. A semantic unit is a subject of reuse: it is designed to be used with different DSMLs. Because of this desired property, all of the semantic (and possibly syntactic) variations are kept in the transformation part. Note that the semantic unit can be expressed in any formalism that does not have to be executable. ASML was used in the projects described above and is suitable for execution and test generation, but formalisms better suited for model checking (e.g., nuSMV [12]) can be used as well.

3.2. Definition via an Interpreter

The approach described in the previous section is well-suited for cases when one semantic unit can serve a number of DSMLs and all the semantic variations can be captured in the transformation. However, this is not the case for many DSMLs, most notably the 20+ variants of the Statechart notation [5]. In this case, another approach is to simplify the translation part and define the semantics using an *interpreter* that directly executes the models.

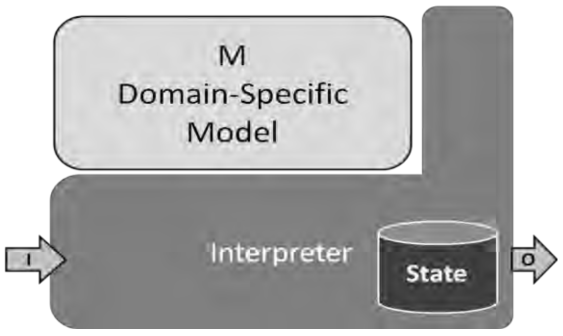


Fig. 3. Defining semantics via an interpreter

Formally, an interpreter is a mapping *i* that depends on the model M, and implements $i(M) : I \times S(M) \rightarrow O \times S(M)$, where I is the input event alphabet, O is the output event alphabet, and S is the set of the internal states of the interpreter, also dependent on the model. The concept is illustrated in Figure 3. The model is a read-only data structure that controls the interpreter's behavior, while the state is updated by the interpreter as it processes inputs. Of course, an interpreter is not different conceptually from a semantic unit, but typically much more complex.

Such interpreters can be defined in any executable language, including conventional languages. This has advantages: (1) any developer skilled in the implementation language can understand the specification of the semantics, (2) all the formal reasoning and analysis tools available for the implementation language can be used, (3) fast prototyping of semantics is feasible, (4) program verification, debugging, and testing tools available for the implementation language can be immediately used. The disadvantages of the approach are: (1) reasoning about programs is typically more difficult than reasoning about models, (2) verifying an interpreter + model assembly is inefficient, as the resulting system has many more states than strictly needed by the model, and (3) treating non-deterministic behavior as complex, because a concrete interpreter is always deterministic.

We have used this interpreter-based approach to define the semantics of two Statechart variants: (1) UML State machines, and (2) Matlab Stateflow. Each has a specification of about 100 pages in English, and for some subsets formal specifications exist, but are documented in journal papers. We have defined a common data structure (an abstract syntax) for the models, and coded the interpreter in pure Java (only the core libraries were used). The code for the abstract syntax part was about 600 lines; functional code common across the two variants required about 250 lines; the Stateflow-specific code had about 600 lines; and the UML State machine variant had about 400 lines. All the code was reviewed by 3-4 programmers and thoroughly tested and compared to existing tools using carefully chosen examples (models and input/output sequences). Our experience indicates that such interpreter-based specification is feasible, and can be quite compact.

3.3. Challenges

When defining the semantics of DSMLs, several challenges arise, some of which are listed below.

Existence of valid models. One can define an abstract syntax with very restrictive well-formedness constraints, such that no valid models can be constructed. In the case of a complex DSML, it may become a challenge to recognize such a problem.

Existence of valid models that generate an acceptable behavior. A secondary problem is to verify if a valid model exists that generates an acceptable behavior, which, for instance satisfies certain properties (e.g., deadlock freedom). It is a defect of the semantics definition if such a model cannot be constructed.

Composability. In a project, multiple DSMLs are often used. Syntactic composition can be simple, but composition of semantics needs to be investigated more thoroughly as a core research topic.

Efficiency of verification with interpreters. The interpreter-based method has a shortcoming: the system has much more states than the original model, so its verification is more complex. We need techniques to introduce abstractions over the states of an interpreter-based system to reduce the complexity.

Reusability. One goal of the semantic units was reusability, and the same applies to the interpreter-based approach. We were able to take advantage of the features of the implementation language (namely, inheritance and polymorphism) when developing the interpreters for the Statechart variants, but the question arises regarding how this can be extended to other cases.

Dissemination. Definition of the semantics for a DSML must be published in a form supporting review by the stakeholders. A key research question regards the best way of disseminating or sharing such specifications.

4. A DSML with Metamodel-Based Semantics

Recent advances in unified communication, mobile technology, and the desire for collaborators from geographically dispersed teams to coordinate their communication activities are becoming commonplace. There is a strong demand for an easy and flexible way of building user-centric communication services that effectively shields users of these systems from the heterogeneity of communication technologies, and that supports the dynamic nature of communication-based collaboration. Many existing communication service frameworks are custom-built, inflexible, costly, and technology specific. They provide little or no support for user-driven specification, adaptation and coordination of communication services performed in response to changes in highly dynamic environments (e.g., those found in disaster management and healthcare).

To address the aforementioned problems, Deng et al. [14] proposed the *Communication Virtual Machine technology* which consists of an interpreted DSML, the *Communication Modeling Language (CML)*, and a semantic rich platform to execute the communication models, the CVM. In this section we present an extension of CML called the *Workflow Communication Modeling Language (WF-CML)* that better supports the dynamic coordination of communication services. WF-CML defines communication-specific abstractions of workflow concepts found in many of the major general-purpose workflow languages, including UML activity diagrams [41], YAWL [1], and Windows Workflow Foundation [39]. The definition of WF-CML includes the metamodel and the dynamic semantics. Due to space limitations, we only present a subset of the metamodel and an overview of the dynamic semantics, yet to be completed.

4.1. Motivating Scenario

To further motivate the need for WF-CML, we present a scenario developed at the Miami Children's Hospital [8]. The following are the actors in the scenario: A Discharge Physician (DP), a Senior Clinician (SC), a Primary Care Physician (PCP), a Nurse Practitioner (NP) and the Attending Physician (AP). Patient Discharge Scenario:

(1) On the day of discharge, Dr. Burke (DP) establishes an audio communication with Dr. Monteiro (SC) to discuss the discharge of baby Jane. During the conversation, Dr. Burke composes a discharge package, `DisPkg_1`, referred to as a *form*, and sends it to Dr. Monteiro to be validated. The `DisPkg_1` form consists of a `RecSum-Jane.txt` (text file), summary of patient's condition; `xRay-Jane.jpg`, an x-Ray of the patient's heart, (non-streamfile); and a `HeartEcho-Jane.mpg` (video clip), an echocardiogram (echo) of the patient's heart. After `DisPkg_1` is sent, Dr. Burke contacts Dr. Sanchez (PCP) to join the conversation with Dr. Monteiro to discuss the

patient's condition. During the conversation, Dr. Monteiro validates `DisPkg_1` and sends it to Dr. Burke.

(2) Since the form `DisPkg_1` is received within 24 hours and is validated, Dr. Burke then sends it to Nurse Smith (NP) and Dr. Wang (AP) (If the form had not been validated and received within 24 hours, the workflow requires that Dr. Burke send out an interim discharge note (`InterimNote_1`)). At the same time, Dr. Burke continues his conference with Drs. Monteiro and Sanchez.

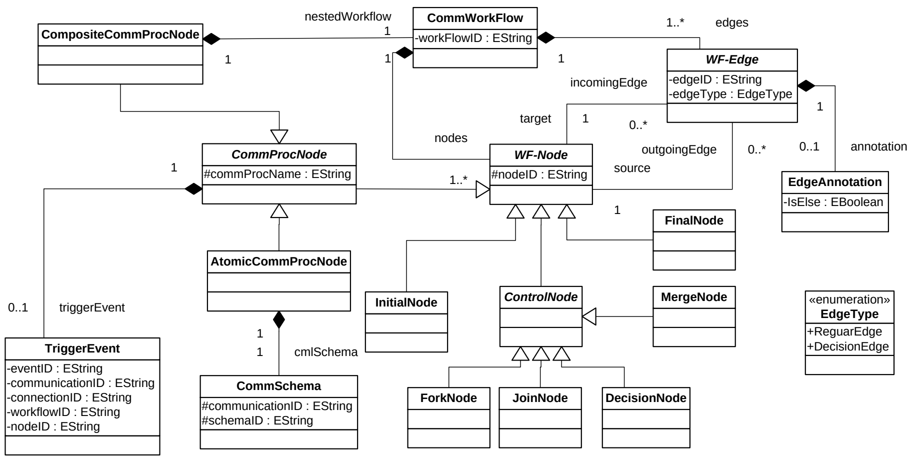


Fig. 4. Partial Abstract Syntax for WF-CML

4.2. Metamodel

The metamodel for WF-CML consists of the abstract syntax, represented as a UML class diagram, and the static semantics defined using OCL. Figure 4 shows a partial class diagram of the abstract syntax for WF-CML. The complete class diagram and static semantics can be found on the project's web page[1].

A WF-CML model is a graph (`CommWorkFlow`) consisting of nodes (`WF-Node`), edges (`WF-Edge`), and trigger events (`TriggerEvent`) as shown in Figure 4. The nodes are described as follows: `InitialNode` and `FinalNode` – signify the beginning and ending of a model representing the coordination of communication processes. `CommProcNode` (communication process node) - is either an atomic communication model (`AtomicCommProcNode`) or a nested workflow model (`Composite-CommProcNode`) and has zero or one trigger event associated with the node. The atomic communication model represents a model created using pre-

[1] http://cml.cs.fiu.edu/

workflow CML. `DecisionNode`, `ForkNode`, `JoinNode` and `MergeNode` - express control flow between communication processes. There are two types of edges (decision and regular). A decision edge is annotated with zero or more atomic events. If there is no event annotation on the decision edge, it is considered an *else* edge.

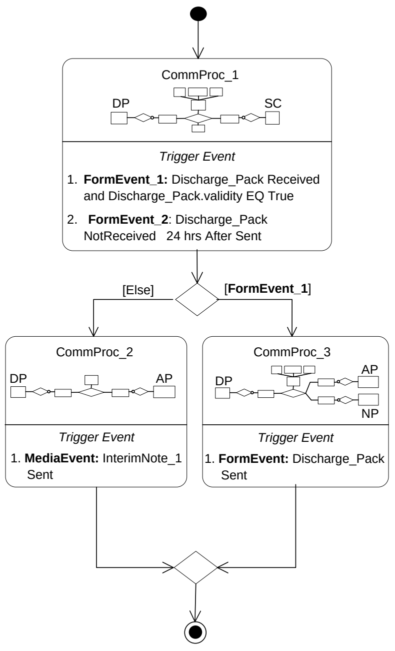

CommProc_1
DP
SC
Trigger Event
1. FormEvent_1: Discharge_Pack Received and Discharge_Pack.validity EQ True
2. FormEvent_2: Discharge_Pack NotReceived 24 hrs After Sent
[Else]
[FormEvent_1]
CommProc_2
DP
AP
Trigger Event
1. MediaEvent: InterimNote_1 Sent
CommProc_3
DP
AP
NP
Trigger Event
1. FormEvent: Discharge_Pack Sent

Fig. 5. WF-CML Model for Scenario

Figure 5 shows the WF-CML model for the scenario described in the previous section. The CML model in `CommProc_1`, top node in the figure, specifies the communication between the DP and the SC, the user ids and names are instantiated when the WF-CML model is executed by Dr. Burke, and he loads the contact information for the SC. There are two types defined for this communication, a form type (`Discharge_Pack`) and a built-in media type (`LiveAudio`). The trigger event in `CommProc_1` states that this node is exited when a validated patient form of type `Discharge_Pack` is received, in this case `DisPkg_1`, and it is validated; or the patient form is not received 24 hours after being sent.

4.3. Dynamic Semantics

The semantic rules of WF-CML extend the semantic rules for CML [54]. We first provide an overview of the semantic rules for realizing CML models followed by the semantics rules for WF-CML models.

CML:

$((CI_{in}, DI_{in}), CSP\_Env_i) \Rightarrow ((CI_{out}, DI_{out}), Script_{out}, Event_{out}, CSP\_Env_{i+1})$

where:

(CI_{in}, DI_{in}) - input control and data instances capturing a user's communication needs to be realized by the communication service.

CSP_Env_i - state of the CS process including the state of the executing control and data instances, (CI_i, DI_i), negotiation state, Neg_i, and media transfer state, MT_i.

(CI_{out}, DI_{out}) - updated control and data instances generated during the transition.

$Script_{out}$ - communication control script generated, including (re)negotiation and media transfer scripts, executed by the CVM middleware.

$Event_{out}$ - output event generated during the execution of the CS process, including media events or negotiation events.

CSP_Env_{i+1} - updated environment of the CS process. The structure is similar to CSP_Env_i stated above.

WF-CML:

$(Event_{in}, WF\_Env_i) \Rightarrow ((CI_{out}, DI_{out}), WF\_Env_{i+1})$

where:

$Event_{in}$ - an input event that may trigger the execution of the next node in the WF-CML model. These events include negotiation events, data transfer events and exception events.

WF_Env_i - the current configuration of a process executing the WF-CML model (WF_Proc). Its state is defined as (WF_{exec}, CS_Procs, Curr_CS),

where:

WF_{exec} - the currently executing WF-CML model in the WF_Proc process.

CS_Procs - a list of executing CS processes in the executing WF_Proc process.

Curr_CS - currently active CS processes with respect to the WF Proc process.

WF Env_{i+1} - the updated configuration of the WF Proc process.

The rules describing the semantics for CML and WF-CML models may be applied to the motivating scenario presented in Section 4.1 as follows. The WF-CML model is processed using the semantics rule for WF-CML and shown in Tables 1 and 2. Table 1 shows the left-hand side of the rule and Table 2 the right-hand side of the rule. The input to the rule, shown in the third row in Table 1 (i.e., when i = 0), includes: (1) the null event, and (2) the workflow environment (WF_{exe}). The current workflow environment includes: (a) the WF-CML model shown in Figure 5, (b) the list of executing processes (`CS_Procs`), which is empty, and (c) the currently active CS processes in the workflow (`Curr_CS`), which is null. The output of the rule includes: (1) the control instance and data instance pair (CI, DI) to be processed by the CML semantic model, (2) the currently executing WF-CML model (Figure 5), (3) the list of executing processes which is `Comm_Proc_1`, the top node in Figure 5, and (4) the currently active node in the WF-CML model, `Comm_Proc_1`.

Table 1. Left-hand side of the semantic rule used for WF-CML.

i	$Event_{in}$	WF_Env_i		
		WF_{exec}	CS_Procs	Curr_CS
0	null	WF-CML model (see Figure 5)	empty	null
…				
k	FormEvent_1	WF-CML model (see Figure 5)	CommProc_1 (see Figure 5, top node)	CommProc_1 (see Figure 5, top node)
…				

Table 2. Right-hand side of the semantic rule used for WF-CML.

i	(CI_{out}, DI_{out})	WF_Env_{i+1}		
		WF_{exec}	CS_Procs	Curr_CS
0	$(M_1$, null)	WF-CML model (see Figure 5)	CommProc_1 (see Figure 5, top node)	CommProc_1 (see Figure 5, top node)
…				
k	$(M_p$, null)	WF-CML model (see Figure 5)	CommProc_1, CommProc_3 (see Figure 5)	CommProc_3 (see Figure 5, bottom right)
…				

The (CI, DI) model pair extracted from the WF-CML model is processed by the semantic rule for CML. The left-hand and right-hand sides of the rule are shown in Tables 4 and 5, respectively. Table 3 shows some of the CML models used during the realization of the communication. The third row of Table 4, where j = 0, shows the input model pair of $(M_1$, null). Table 3 shows that M_1 is a model representing the communication between two persons and the connection (C1) supports the transmission of live audio and a patient discharge form. The media and form types on the connection are not labeled. We use the pair (null, null) in Table 4 to represent the initial models in the system. The initial states for the negotiation and media transfer state machines are the negotiation ready state (`Neg_Ready`) and the media transfer ready state (`MT_Ready`), respectively. After applying the rule, Table 5 shows the output generated and the updated state of the system. The models generated are the same as the input models because these models are used during negotiation; the script generated creates a connection with the remote party in the connection, Dr. Monteiro, and sends the control model (M_1); the event generated (`Neg_Initiated`) reflects that negotiation has started. The entries in the table for j=1 and j=2 represents the negotiation process. The application of the rule shown in Tables 4 and 5 with the row labeled j=2 shows the application of the rule to enable live audio.

Table 3. Some of the CML models used in the motivating scenario.

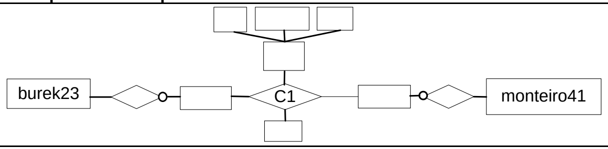
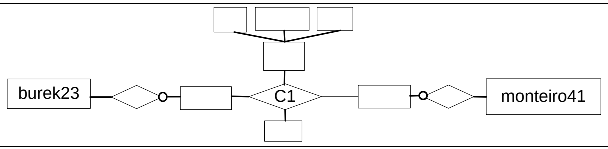
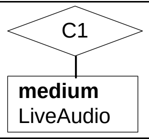
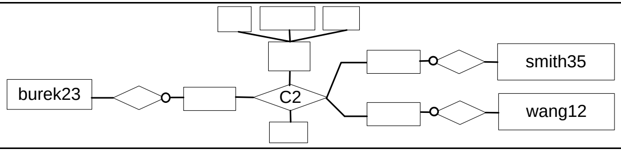
Model ID	Graphical Representation of the CML model
M_1 (control instance)	
M_2 (control instance)	
M_3 (data instance)	
…	
M_p	

Table 4. Left-hand side of the semantic rule used for CML.

j	(CI_{in}, DI_{in})	CSP_Env_i			Comments
		(CI_i, DI_i)	Neg_i	MT_i	
0	(M_1, null)	(null, null)	Neg_Ready	MT_Ready	M_1 is the control instance model created by the local participant, Dr. Burke
1	(M_2, null)	(M_1, null)	WaitingSameCI	MT_Ready	M_2 is the control instance model received by Dr. Burke's CVM from the remote participant, Dr. Monteiro
2	(M_2, M_3)	(M_2, null)	Neg_Ready	MT_Ready	M_3 is the model that represents the activation of live audio
…					

During the communication for `CommProc_1` an event will eventually be triggered that moves the workflow onto the next node. In the scenario, the `FormEvent_1` is triggered, as shown in Table 1 row labeled i = k. The right-hand side of the WF-CML rule in Table 2 shows that both `CommProc_1` and `CommProc_3` are now active and the currently active node with respect to workflow is `CommProc_3`. Two communication processes are active since our semantics do not force the termination of a communication after the workflow model moved on to the next node. Note that the control model (M_p) is now processed by the CML semantic rule which establishes a new connection with two participants, Nurse Smith and Dr. Wang.

Table 5. Right-hand side of the semantics rule used for CML.

j	(CI_{out}, DI_{out})	$Script_{out}$	$Event_{out}$	CSP_Env_{i+1}		
				(CI_{i+1},DI_{i+1})	Neg_{i+1}	MT_{i+1}
0	(M_1, null)	createConnection (“C1”); sendSchema (“C1”, “burke23”, “monteiro41”, “M_1, null”)	Neg_ Initiated	(M_1, null)	Neg_ Initiated	MT_ Ready
1	(M_2, null)	sendSchema (“C1”, “burke23”, “monteiro41”, “M_2, null”); addParticipant (“C1”, “monteiro41”)	Neg_ Complete	(M_2, null)	Neg_ Complete	MT_ Ready
2	(M_2, M_3)	enableInitiator (“C1”, “LiveAudio”); sendSchema (“C1”, “burke23”, “monteiro41”, “M_2, M_3”)	Enable_ Stream	(M_2, M_3)	Neg_ Ready	Stream_ Enabled
…						

4.4. Challenges

WF-CML supports the execution of communication models in a distributed environment, where participants in the communication are allowed to change the currently executing communication process. The complexity of executing WF-CML models directly provide us with the following challenges: (1) What notation should be used to define the dynamic semantics (e.g., operational, denotational, or axiomatic)? (2) How to define the environments for a communication process and workflow process? (3) How can the semantics be extended to support dynamic adaptation of the WF-CML?

5. Model-Based Verification Tools

MDE provides a context in which formal specification and verification techniques can be applied. There is evidence that this is already taking place (e.g., see [11], [23], [30], [34], [47]). With respect to the UML, in the late nineties the precise UML (pUML) group helped raise awareness of the need for more formal descriptions of UML semantics to enable rigorous analysis of structural and functional properties of systems captured in UML models. Over the last decade, we have seen a significant number of papers on using relatively mature formal verification techniques to analyze properties described in particular UML models (e.g., there has been significant work on using model checking techniques to analyze UML state machine models, and Petri net variants to analyze activity models).

Despite the focused attempts, there are very few UML-based verification tools that can be described as usable by practitioners. In the following, we discuss some of the opportunities for applying verification techniques in MDE and discuss some of the challenges. For the most part, the opportunities and challenges are presented in terms of UML modeling issues, primarily because this is one of the more widely used (and misused) MDE languages, and there is a dire need for practical UML-based verification tools.

5.1. Towards Usable UML-based Verification Tools

The UML has reached a level of maturity that now allows us to reach for some of the lower hanging fruit (not necessarily the same as low-hanging fruit!) where application of rigorous verification techniques are concerned. One of the frustrating experiences that a modeling student or practitioner learning a language such as the UML goes through is determining if his/her model is, in some sense, a valid description. In the case of students, the only feedback that they often receive is the instructor’s grade of their work. There is a need to provide modelers, in particular, UML modelers, with some means of checking the validity of their model.

An obvious approach is to provide some support for executing or animating models. The Colorado State University (CSU) UMLAnT (UML Animation and Testing) tool provides a means for dynamically analyzing (testing) UML design models. A UMLAnT design model consists of class diagrams with operations specified in a Java-like action language called JAL [15]. UMLAnT is an Eclipse plug-in that provides support for (1) generating test inputs that satisfy criteria based on coverage of elements in a sequence diagram that describes the scenarios that will be exercised in a test, (2) executing the design model using test inputs (a test input is an operation with parameter values), and (3) showing execution progress in terms of sequence diagrams and changes to object configurations. We are currently updating the tool to the latest version of Eclipse and improving its robustness.

We are also developing lightweight scenario-based analysis techniques that allow developers to check whether a scenario describing a desired or undesired behavior is supported by a model [56]. The technique provides a less expensive way of analyzing a system in the cases where exhaustive formal analysis is not possible or cost-effective. In the approach we are developing, a behavior is described as a sequence of snapshots, where a snapshot is an object configuration that conforms to a class diagram. A class model with operations specified in the OCL is transformed to a class model, called a Snapshot Model, that characterizes all possible behaviors (sequences of snapshots). A verifier then provides scenarios (expressed as sequence diagrams) and the analysis tool we are developing checks whether these scenarios conform to the Snapshot Model.

One of the problems that our analysis approaches and those developed by other researchers face is that they do not handle incomplete models well. This is one of the challenges that we are currently tackling in our analysis work. Another aspect that requires attention is ensuring consistency of behavioral and structural concepts across different modeling views. This is a particularly challenging problem in the UML, and is sometimes one of the reasons practitioners limit their use to one or two UML diagram types (typically class diagrams, sequence diagrams or state machine diagrams). One of the problems that hinders research in this area is the size of the UML language (as reflected in its metamodel) - this makes it very difficult to determine precisely the consistency relationships that must hold across elements in different diagrams. Furthermore, it has not been verified that the UML metamodel is a valid description that can be relied upon correctly to define these relationships. A good usability challenge problem for verification tools is finding an answer to the question “is the UML metamodel correct?”

5.2. Formal Verification Challenges: Transformations, Semantic Variations, and Models@Run.Time

The previous subsection identified some obvious opportunities for applying verification techniques in the MDE context. That was just the tip of the iceberg; there are other more challenging verification problems that should be tackled in MDE. A challenging problem concerns verification of model transformations [35]. In a recently published paper on testing model transformations, we highlighted some of these challenges [4]. One of the major problems concerns generating an adequate set of test models. Generating test inputs for programs that use inputs with simple structures is challenging in itself; when the inputs are models with complex structures the challenges are greater.

Another problem that must be considered is the variety of semantics that can be associated with languages such as the UML. In the UML, some parts of the semantics are intentionally left undefined to allow users to tailor semantics to their needs. While formal methods purists may argue for defining a single semantics for the UML, the practical reality is that different groups use the UML differently, and this need must be supported. It is highly unlikely that a single verification approach would meet all structural, functional and behavioral analysis needs. To tackle this problem we have started a research initiative called GeMoC (Generic Model of Computation) with the goal of developing a verification framework that can be used in a modeling environment that supports a variety of semantics (or models of computation).

An emerging MDE research area that attempts to extend the use of models to runtime management is models@run.time [6]. There has been significant work on using models to support runtime adaptation of software. Verifying adaptations at runtime is a particularly challenging problem that groups working in this area are currently addressing.

6. Semantics-Based Tools in Domain-Specific Modeling

As mentioned in Section 2, a formal description of a modeling language allows for the automatic generation of supporting tools that are based on the modeling language semantics. This section motivates the need for such tool generation by summarizing our previous work in generating debuggers and testing engines for domain-specific languages (DSLs) [31], [37]. Our framework (Figure 6) for automatic generation of DSL debuggers and test engines reuses existing GPL tools [55]. The framework consists of a mapping process that records the correspondence between the DSL program and the generated GPL code, a tool methods mapping that specifies how DSL tool actions are mapped to GPL tool actions (e.g., a DSL debugging command might request execution of several GPL debugging commands), and a tool results mapping, which specifies how obtained results should be displayed to the end-user using only DSL abstractions.

Existing approaches for defining the formal semantics of programming languages can be used to specify the semantics of DSMLs. However, a critical point of this work is that a semantics definition should be model-based. To fulfill this objective and accomplish transparency of low-level formalisms, three steps are followed. The first step focuses on the methodology to specify state transitions to show dynamic behavior of meta-elements. The second step concerns the visual language to control the sequence of the defined state transitions and runtime configurations. The third step includes transformation of specifications into the different language-based tools. The combination of all outcomes of these steps will form the semantic framework. Figure 7 shows an outline of the approach. The first part of the figure demonstrates abstract syntax and static semantic definitions; current platforms provide a means for specifying these definitions. The second part depicts the dynamic semantics specification technique based on activity diagrams and graph grammars. These tools are used to define a sequence of state transitions. The last part shows specification of verification properties within domain boundaries. Finally, all these specifications can be transformed into the different language-based tools (e.g., interpreter, code generator, simulator, verifier).

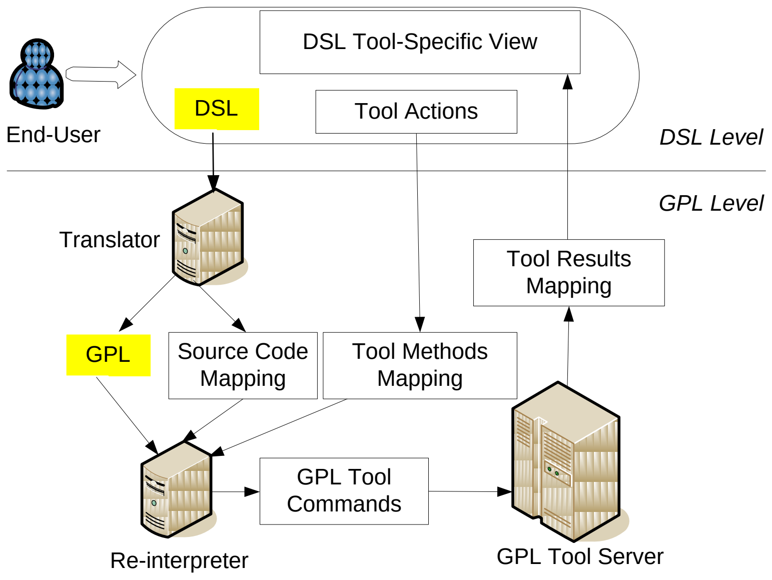


Fig. 6. The framework for automatic generation of DSL tools

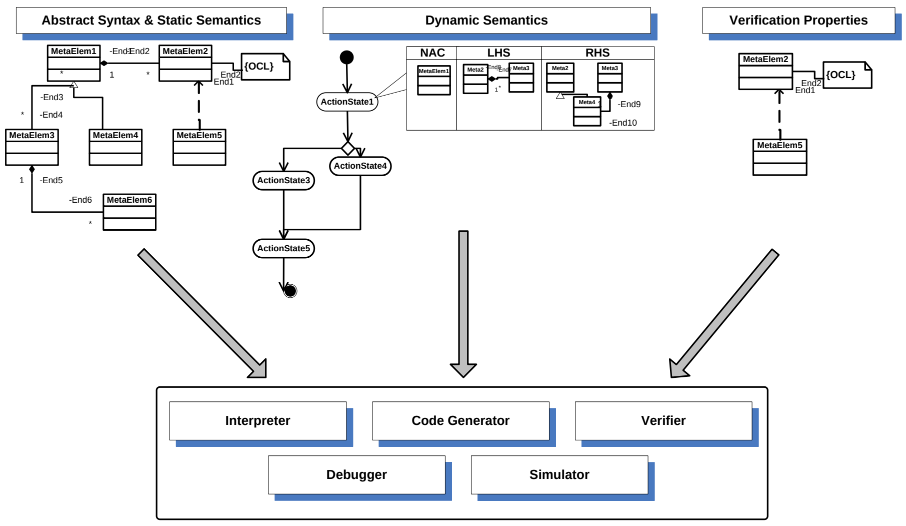


Fig. 7. Semantics-Based Tool Generation

In [13], we performed some experiments on semi-automatic generation of tools for modeling languages and focused on how to specify the behavioral semantics of a DSML by a sequence of graph transformation rules, enabling transformation of a modeling language specification into the model checking tool Alloy [28]. In our initial study, we demonstrated specification of sequential system semantics that connects the initial model to possible result models. First, we focused on how to specify the behavioral semantics of a DSML by a sequence of graph transformation rules. While each graph transformation rule represents a state change of a sequential system, a sequence of state changes is defined by an activity diagram. Sequence definitions control what state transition is to be fired, in what order, and what condition. All these steps are mapped into a transition system that is used to generate a state space. We provided an example to demonstrate semantics definition of a DSML and verification of an assertion in one of the model checking tools (i.e., Alloy). The activities investigated in our initial work can be summarized by the following items:

1. mapping metamodel elements to Alloy abstract signatures,
2. mapping model elements to Alloy concrete signatures,
3. mapping graph transformation rules to Alloy predicates, and
4. mapping verification tasks to Alloy asserts.

Abstract signatures are used to define the meta-layer of the models. To define a model layer in Alloy, these abstract signature definitions are extended into concrete signatures. Each model element is mapped into an appropriate concrete signature in Alloy. Behavioral specifications, which we define by means of graph transformation rules, are mapped into Alloy predicates. Each task defined in a semantics definition is transformed into an Alloy predicate having two parameters, g and g', representing the current state and the next state. Finally, the assertions that would be satisfied at the final states are transformed into Alloy assert definitions.

Although our current investigation was performed manually, it demonstrated how DSML designers can define semantic and verification specifications using visual models. We are currently investigating how to generalize and automate this process.

7. Conclusions

DSMLs allow end-users and domain experts to specify the core essence of a problem using visual abstractions that are close to the problem space of a specific domain. A key research challenge in the adoption of such modeling languages concerns the manner in which the semantics of each DSML is specified. Typically, the behavioral semantics of a DSML is described within individual hard-coded model interpreters. Such a representation of the semantics is not specified in a manner that is ameliorable to formal analysis and generation of model-based tools. As such, the utility of a DSML is hampered due to the lack of a single representation that formally denotes the semantics of the language. This paper has described several research projects that investigate and develop a formal, yet widely usable, means to specify DSML semantics. Our future work is automatic generation of model interpreters, simulators, debuggers and verifiers from such semantic specifications, which would have significant impact on the current practice of model-driven engineering in terms of automating many tasks that are currently done ad hoc in a manual hand-crafted manner.

Acknowledgments. This work was supported in part by United States National Science Foundation awards CAREER CPA-1052616, CCF-0811630, CCF-1018711, HRD-0833093, OISE-0730065, and OISE-0968596, and programs by the Defense Advanced Research Project Agency: the Evolutionary Design of Complex Systems and Model-based Integration of Embedded Systems, by the National Science Foundation's ITR program, by Boeing and General Motors through the ESCHER initiative, and NASA's Robust Software Engineering program.

References

1. van der Aalst, W. M. P., ter Hofstede, A. H. M.: YAWL: Yet Another Workflow Language. Information Systems, Vol. 30, No. 4, 245–275. (2003)
2. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The Design of a Language for Model Transformations. Journal on Software and System Modeling, Vol. 5, No. 3, 261–288. (2006)
3. Álvarez, J. M., Evans, A., Sammut, P.: Mapping between Levels in the Metamodel Architecture. In: Gogolla, M., Kobryn, C. (eds.): The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, Vol. 2185. Springer-Verlag, New York, 34-46. (2001)
4. Baudry, B., Ghosh, S., Fleury, M., France, R., La Traon, Y., Mottu, J.-M.: Barriers to Systematic Model Transformation Testing. Communications of the ACM, Vol. 53, No. 6, 139-143. (2010)
5. von der Beeck, M.: A Comparison of Statecharts Variants. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.): Formal Techniques in Real-Time and Fault-Tolerant Systems. Lecture Notes in Computer Science, Vol. 863. Springer-Verlag, Berlin, 128–148. (1994)
6. Blair, G., Bencomo, N., and France, R. R.: Models @ run.time. Computer, Vol. 42. No. 10, 22–27. (2009).
7. Börger, E.: The Origins and Development of the ASM Method for High-Level System Design and Analysis. Journal of Universal Computer Science, Vol. 8, No. 1, 2-74. (2002)
8. Burke, R. P., White, J. A.: Internet Rounds: A Congenital Heart Surgeon's Web Log. Seminars in Thoracic and Cardiovascular Surgery, Vol. 16, No. 3, 283–292. (2004)
9. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic Anchoring with Model Transformations. In: Hartman, A., Kreische, D. (eds.): Model Driven Architecture - Foundations and Applications. Lecture Notes in Computer Science, Vol. 3748. Springer-Verlag, Berlin, 115-129. (2005)
10. Chen, K., Sztipanovits, J., Neema, S.: Compositional Specification of Behavioral Semantics. In Proceedings of DATE '07, Design, Automation and Test in Europe, IEEE, Nice, France, 906-911. (2007)
11. Chiorean, D., Pasca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML Models Consistency Using the OCL Environment. Electronic Notes in Theoretical Computer Science, Vol. 102, 99 – 110. (2004)
12. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A New Symbolic Model Checker. International Journal on Software Tools for Technology Transfer, Vol. 2, 2000. (2000)
13. Demirezen, Z., Mernik, M., Gray, J., Bryant, B. R.: Verification of DSMLs Using Graph Transformation: A Case Study with Alloy. In Proceedings of the 6th

International Workshop on Model-Driven Engineering, Verification and Validation, ACM, Denver, Colorado. (2009)

14. Deng, Y., Sadjadi, S.M., Clarke, P. J., Hristidis, V., Rangaswamy, R., Wang, Y.: CVM - A Communication Virtual Machine. Journal of Systems Software, Vol. 81, No. 10, 1640–1662. (2008)
15. Dinh-Trong, T T., Ghosh, S., France, R. B.: A Systematic Approach to Generate Inputs to Test UML Design Models. In Proceedings of ISSRE '06, the 17th International Symposium on Software Reliability Engineering, IEEE, Raleigh, North Carolina, 95–104. (2006)
16. Di Ruscio, D., Jouault, F., Kurtev, I., Bezivin, J., Pierantonio, A.: Extending AMMA for Supporting Dynamic Semantics Specifications of DSLs. Technical Report, INRIA/LINA, http://hal.archives-ouvertes.fr/ccsd-00023008/en. (2006)
17. Ducasse, S., Girba, T., Kuhn, A., Renggli, L.: Meta-Environment and Executable Meta-Language using Smalltalk: An Experience Report. Software and Systems Modeling, Vol. 8, No. 1, 5-19. (2009)
18. Eker, J., Janneck, J. W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming Heterogeneity - The Ptolemy Approach, Proceedings of the IEEE, Vol. 91, No. 1, 127-144. (2003)
19. Engels, G., Hausmann, J., Heckel, R., Sauer, S.: Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Evans, A., Kent, S., Selic, B. (eds.): The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, Vol. 1939. Springer-Verlag, New York, 323-337. (2000)
20. Ermel, C., Holscher, K., Kuske, S., Ziemann, P.: Animated Simulation of Integrated UML Behavioral Models Based on Graph Transformation. In Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE Computer Society, Dallas, Texas, 125-133. (2005)
21. Esser, R., Janneck, J.W.: Moses - A Tool Suite for Visual Modeling of Discrete-Event Systems. In Proceedings of the IEEE 2001 Symposia on Human Centric Computing Languages and Environments (HCC'01), IEEE Computer Society, Stresa, Italy, 272-279. (2001)
22. Gargantini, A., Riccobene, E., Scandurra, P.: A Semantic Framework for Metamodel-Based Languages. Automated Software Engineering, Vol. 16, No. 3, 415-454. (2009)
23. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling, Vol. 4, No. 4, 386-398. (2005)
24. Gray, J., Tolvanen, J.P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-Specific Modeling. In Fishwick, P. A. (ed.): Handbook of Dynamic System Modeling, CRC Press, Boca Raton, Florida. (2007)
25. Hahn, C.: A Domain Specific Modeling Language for Multiagent Systems. In Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, International Foundation for Autonomous Agents and Multiagent Systems, Estoril, Portugal, 233-240. (2008)
26. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"?, Computer, Vol. 37, No. 10, 64-72. (2004)
27. Henriques, P. R., Pereira, M. J. V., Mernik, M., Lenič, M., Gray, J., Wu, H.: Automatic Generation of Language-Based Tools using the LISA System. IEE Proceedings Software, Vol. 152, No. 2, 54-69. (2005)
28. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 2, 256-290. (2002)

29. Kelly, S., Tolvanen, J-P.: Domain-Specific Modeling: Enabling Full-Code Generation, John Wiley and Sons. (2008)
30. Knapp. A.: A Formal Semantics for UML Interactions. In: France, R. B., Rumpe, B. (eds.): The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, Vol. 1723. Springer-Verlag, New York, 116–130. (1999)
31. Kosar, T., Oliveira, N., Mernik, M., Varanda Pereira, M. J., Črepinšek, M., da Cruz, D., Henriques, P. R.: Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. Computer Science and Information Systems, Vol. 7, No. 2, 247-264. (2010)
32. de Lara, J., Vangheluwe, H., Alfonseca, M.: Metamodelling and Graph Grammars for Multi-Paradigm Modelling in AToM 3. Software and Systems Modeling, Vol. 3, No. 3, 194-209. (2004)
33. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments, Computer, Vol. 34, No. 11, 44-51. (2001)
34. Lilius, J. and Porres Paltor, I. Formalising UML State Machines for Model Checking. In: France, R. B., Rumpe, B. (eds.): The Unified Modeling Language. Modeling Languages, Concepts, and Tools, Lecture Notes in Computer Science, Vol. 1723. Springer-Verlag, New York, 430–445. (1999)
35. Lin, Y., Zhang, J., Gray, J.: A Testing Framework for Model Transformations. In Beydeda, S., Book, M., Gruhn, V. (eds.), Model-driven Software Development, Springer, Heidelberg, Germany, 219-236. (2005)
36. Mathworks: Matlab Simulink/Stateflow Tools, http://www.mathworks.com (2010)
37. Mernik, M., Heering, J., Sloane, A. M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys, Vol. 37, No. 4, 316-344. (2005)
38. Microsoft Corporation: The Abstract State Machine Language, http://research.microsoft.com/en-us/projects/asml. (2010)
39. Microsoft Corporation: Windows Workflow Foundation, http://msdn.microsoft.com/en-us/vbasic/cc506054 (2010)
40. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving Executability into Object-Oriented Meta-Languages. In: Briand, L. C., Williams, C. (eds.): Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science, Vol. 3713. Springer-Verlag, Heidelberg, Germany, 264-278. (2005)
41. Object Management Group. Unified Modeling Language: Superstructure, Version 2, http://www.omg.org/spec/UML/2.3. (2010)
42. Porubän, J., Forgáč, M., Sabo, M., Běhálek, M.: Annotation Based Parser Generator. Computer Science and Information Systems, Vol. 7, No. 2, 291-307. (2010)
43. Romero, J.R., Rivera, J.E., Durán, F., Vallecillo, A.: Formal and Tool Support for Model Driven Engineering with Maude, Journal of Object Technology, Vol. 6, No. 9, 187-207. (2007)
44. Sadilek, D. A., Wachsmuth, G.: Using Grammarware Languages to Define Operational Semantics of Modelled Languages. In: Oriol, M., Meyer, B. (eds.): Objects, Components, Models and Patterns, Lecture Notes in Business Information Processing, Vol. 33. Springer-Verlag, Heidelberg, Germany, 348-356. (2009)
45. Scheidgen, M., Fischer, J.: Human Comprehensible and Machine Processable Specifications of Operational Semantics. In: Akehurst, D. H., Vogel, R., Paige, R. F. (eds.): Model Driven Architecture - Foundations and Applications, Lecture Notes in Computer Science, Vol. 4530. Springer-Verlag, Heidelberg, Germany, 157-171. (2007)

46. Schmidt, D. C.: Guest Editor's Introduction: Model-Driven Engineering. Computer, Vol. 39, No. 2, 25-31. (2006)
47. Shah, S., Anastasakis, K., Bordbar, B.: From UML to Alloy and Back Again. In Proceedings of MoDeVVa '09, the 6th International Workshop on Model-Driven Engineering, Verification and Validation, ACM, Denver, Colorado, USA, 1–10. (2009)
48. Smith, G.: The Object-Z Specification Language, Kluwer Academic Publishers. (2000)
49. Soden, M., Eichler, H.: Towards a Model Execution Framework for Eclipse. In Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture, ACM, Enschede, Netherlands, 1-7. (2009)
50. Sprinkle, J., Mernik, M., Tolvanen, J.-P., Spinellis, D.: Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? IEEE Software, Vol. 26, No. 4, 15-18. (2009)
51. Sunyé, G., Pennaneac'h, F., Ho, W.-M., Le Guennec, A. and Jézéquel, J.-M.: Using UML Action Semantics for Executable Modeling and Beyond. In: Dittrich, K. R., Geppert, A., Norrie, M. C. (eds.): Advanced Information Systems Engineering, Lecture Notes in Computer Science, Vol. 2068. Springer-Verlag, Heidelberg, Germany, 433-447. (2001)
52. Sztipanovits, J., Karsai, G.: Model-Integrated Computing. Computer, Vol. 30, No. 4, 110-111. (1997)
53. Varró, D.: A Formal Semantics of UML Statecharts by Model Transition Systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.): Graph Transformation, Lecture Notes in Computer Science, Vol. 2505. Springer-Verlag, Heidelberg, Germany, 378-392. (2002)
54. Wang, Y., Wu, Y., Allen, A., Espinoza, B., Clarke, P.J., Deng, Y.: Towards the Operational Semantics of User-Centric Communication Models. In Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference, vol.1, pp.254-262. (2009)
55. Wu, H., Gray, J., Mernik, M.: Grammar-Driven Generation of Domain-Specific Language Debuggers. Software: Practice and Experience, Vol. 38, No. 10, 1073-1103. (2008)
56. Yu, L., France, R. B., Ray, I.: Scenario-Based Static Analysis of UML Class Models. Model-Driven Engineering Languages and Systems. Lecture Notes in Computer Science, Vol. 5301. Springer-Verlag, New York, 234–248. (2008)

Barrett R. Bryant is Professor and Associate Chair of Computer and Information Sciences at the University of Alabama at Birmingham (UAB). He received his B. S. in computer science from the University of Arkansas at Little Rock in 1979 and his Ph. D. in computer science from Northwestern University in 1983, after which he joined UAB. His research interests include theory and implementation of programming languages, formal specification of software systems, and component-based software engineering. He is a member of EAPLS, and a senior member of ACM and IEEE.

Jeff Gray received the BSc and MSc degrees in Computer Science from West Virginia University in 1991 and 1993, and the Ph.D. in Computer Science from Vanderbilt University in 2002. He is currently Associate Professor of Computer Science at the University of Alabama. Jeff's research interests are in the general areas of software engineering and programming languages, and in the specific areas of model-driven engineering, aspect orientation, and software evolution. He is a member of the IEEE and ACM.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998, respectively. He is currently Professor of Computer Science at the University of Maribor. He is also Visiting Professor of Computer and Information Sciences at the University of Alabama at Birmingham, 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.

Peter J. Clarke received his BSc. degree in Computer Science and Mathematics from the University of the West Indies (Cave Hill) in 1987, MS degree from SUNY Binghamton University in 1996 and PhD in Computer Science from Clemson University in 2003. His research interests are in the areas of software testing, software metrics, model-based testing, model-driven software development and domain-specific modeling languages. He is currently Associate Professor of Computing and Information Sciences at Florida International University. He is a member of the ACM (SIGSOFT, SIGCSE, and SIGAPP); IEEE Computer Society; and a member of the Association for Software Testing (AST).

Robert France is Professor of Computer Science at Colorado State University. His research interests are in the area of Software Engineering, in particular formal specification techniques, software modeling techniques, design patterns, and domain-specific modeling languages. He is an editor-in-chief of the Springer journal on Software and System Modeling (SoSyM), a Software Area Editor for IEEE Computer, and is a past Steering Committee Chair of the MoDELS/UML conference series. He was also a member of the revision task forces for the UML 1.x standards. He was awarded the Ten Year Most Influential Paper award at MODELS in 2008.

Gabor Karsai is Professor of Electrical Engineering and Computer Science at Vanderbilt University and a senior research scientist in the Institute for Software-Integrated Systems at Vanderbilt. He conducts research in model-integrated computing. Karsai received a Technical Doctorate degree in Electrical Engineering from the Technical University of Budapest, Hungary, and a PhD in Electrical Engineering from Vanderbilt University. He is a member of the IEEE Computer Society.

Received: January 14, 2011; Accepted: March 11, 2011.

DOI:10.2298/CSIS110214013B

Software Agents: Languages, Tools, Platforms

Costin Bădică[1], Zoran Budimac[2], Hans-Dieter Burkhard[3], and Mirjana Ivanović[2]

[1]Software Engineering Department, Faculty of Automatics, Computers and Electronics,
Bvd.Decebal, Nr.107, Craiova, RO-200440, Romania
badica_costin@software.ucv.ro
[2] Faculty of Sciences, Department of Mathematics and Informatics
Trg Dositeja Obradovica 4, 21000 Novi Sad, Serbia
{zjb, mira}@dmi.uns.ac.rs
[3]Humboldt University, Institute of Informatics,
Rudower Chaussee 25, D-12489 Berlin, Germany
hdb@informatik.hu-berlin.de

Abstract: The main goal of this paper is to provide an overview of the rapidly developing area of software agents serving as a reference point to a large body of literature and to present the key concepts of software agent technology, especially agent languages, tools and platforms. Special attention is paid to significant languages designed and developed in order to support implementation of agent-based systems and their applications in different domains. Afterwards, a number of useful and practically used tools and platforms available are presented, as well as support activities or phases of the process of agent-oriented software development.

Keywords: agent technologies, agent programming languages, agent platforms.

1. Introduction

The metaphor of “intelligent software agents” as basic building blocks for the development of new generation intelligent software systems triggered both theoretical and experimental computer science research aiming to develop new programming languages for agent systems. Fifteen years ago [64] software agent technology has been recognized as a rapidly developing area of research and one of the fastest growing areas of information technology.

In our opinion, the main achievement of this trend of research was the development of new programming models that address both the basic features of agenthood (autonomy, reactivity, proactivity and social abilities) as well as more advanced, human-like features usually collectively coined in the agent literature as “mental attitudes” (beliefs, desires, intentions, commitments), following the model of “intentional systems” introduced by the philosopher Daniel Dennett in 1971 to explain behavior of rational agents.

Agent oriented technologies, engineering of agent systems, agent languages, development tools and methodologies are an active and emergent research area and agent development is getting more and more interesting. There are many approaches, theories, languages, toolkits, and platforms of different quality and maturity which could be applied in different domains.

Our motivation and the main goal of the paper are to bring a survey in the field of agent technology and to cover different aspects of agents. Agents, agent-oriented programming (AOP), and multi-agent systems (MAS) introduce new and unconventional concepts and ideas. Still, there is a number of definitions of the term ‘agent’ that include a property common to all agents: *agent acts on behalf of its user,* as well as a lot of additional properties: agent communicates with other agents in a multi-agent system; acts autonomously; is intelligent; learns from experience; acts proactively as well as reactively; is modeled and/or programmed using human-like features (beliefs, intentions, goals, actions, etc.); is mobile, and so on.

After more than two decades of scientific work in the field, the challenge is to include agents in real software environments and widely use the agent paradigm in mainstream programming. One way to facilitate this is to provide agent-oriented programming languages, tools and platforms.

Pioneering work is done by the *Foundation for Intelligent Physical Agents (FIPA).* “FIPA was originally formed as a Swiss based organization in 1996. Since its foundations, FIPA has played a crucial role in the development of agents standards and has promoted a number of initiatives and events that contributed to the development and uptake of agent technology. Furthermore, many of the ideas originated and developed in FIPA are now coming into sharp focus in new generations of Web/Internet technology and related specifications." (cf. [116]). Since 2005, FIPA is the standards organization for agents and multi-agent systems of the IEEE Computer Society standards organization.

Our recent overview of the agent programming literature revealed a number of trends in the development of agent programming languages. These trends follow the main achievements of computer science disciplines that are traditionally directly connected to multi-agent systems, i.e. formal methods, object-oriented programming, concurrent programming, distributed systems, discrete simulation, and artificial intelligence. Adding on top of that the metaphor of “humanized agents” with roots in psychology research, by regarding them as intentional systems that are endowed with mental states, we can get a panoramic view of the current status of the world of agent programming languages, tools and platforms. The whole paper or some sections of it could be extremely useful and give more insights into the domain for a wide range of readers. PhD students and young researchers can find plenty of useful information and state-of-the-art in the domain of available languages and platforms for programming software agents. Professionals in different companies who are willing to apply this new, promising technology in everyday programming and implementation of real world applications based on agent technology, could find the paper very helpful. Undergraduate students who like to widen their traditional knowledge and be introduced to modern trends in programming can use it as an additional reading material. Moreover, all of them can find a valuable source of references and suggestions for further reading.

The rest of the paper is organized as follows. The second section attempts to give an overview of all essential notions, issues and concepts related to agents and agent technology which are used in other chapters of the paper. Thereinafter, it makes the distinction between single agent and multi-agent systems, goes through the broad spectrum of agent properties, discusses the most acknowledged classifications of software agents, presents the most well-known agent architectures, and explores the two most important agent communication approaches. Section three lists and discusses standard languages and several prototype languages that have been proposed for constructing and implementing agent-based systems. Afterwards, section four presents a number of tools and platforms that are available to support activities or phases of the process of agent-oriented software development. The last chapter gives some concluding remarks.

2. What is a software agent?

2.1. Introduction

Over the last years, many researchers in different fields have proposed a large variety of definitions for the term “agent”. The common understanding is that it is an entity which “acts autonomously on behalf of others”. Even if we restrict ourselves to computer science, there are a lot of different definitions and a lot of different fields where agents are used. It started 30 or more years ago in (Distributed) Artificial Intelligence. With the arrival of the Internet and with the dissemination of computer games, the notion of agents has become broadly used even by non-experts, e.g. for electronic marketing, assistance systems, search engines, chatter bots etc, or as constituents of larger software projects. For the latter ones, it is useful to distribute the overall tasks to autonomous entities and to organize a framework of cooperation and interaction in a multi-agent system. Agents are typical inhabitants of open systems like the Internet. Open systems have been characterized by Hewitt [58] already in the 80’s as systems with continuous availability, extensibility, modularity, arm-length relationships, concurrency, asynchronous work, decentralized control, and inconsistent information.

Michael Coen [126] puts very small restrictions on a program to be considered as an agent: "... programs that engage in dialogs and negotiate and coordinate transfer of information." In the IBM [127], intelligent agents are defined as: "...software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires.". The Software Agents Group at MIT [128] compares software agents to conventional software and emphasizes the following differences: "Software agents differ from conventional software in that they are long-lived, semi-autonomous, proactive, and adaptive.". More detailed is the so-called “weak notion of agency” by Wooldridge and Jennings [104], [105]. They define an agent as "... a hardware or (more usually) software based computer system that enjoys the properties:

– **autonomy**: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;
– **social ability**: agents interact with other agents (and possibly humans) via some kind of agent-communication language;
– **reactivity**: agents perceive their environment (which may be the physical world, a user via a graphical user interface, a collection of other agents, the Internet, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;
– **proactiveness**: agents do not simply act in response to their environment, they are able to exhibit goal-directed behavior by taking the initiative.

A definition in the sense of “strong notion of agency” is [105]: "An agent is a computer system that, in addition to having the properties identified in the definition of weak agent, is either conceptualized or implemented using concepts that are more usually applied to humans (knowledge, obligations, beliefs, desires, intentions, emotions, human-like visual representation, etc.)."

The “strong notion of agency” corresponds to the usage in the field of artificial intelligence (AI). These systems are often specified using human mental categories: beliefs, plans, goals, intentions, desires, commitments, etc. Shoham [90] claims that the use of mental categories in agent specification is justified only under the following conditions:

– mental categories are precisely defined using some formal theory,
– agent has to obey that theory,
– every mental category used in an agent specification has to give some benefit.

A collection of various agent definitions, based on the weak notion of agency, can be found in [52]. It is not the aim of this paper to give a unique definition of an agent. Instead, the reader will find that the different tools presented in the rest of the paper correspond to different notions. Some more relevant concepts are introduced in the following sections.

2.2. Agent Classification and Architectures

General classifications in the agent community [105] distinguish between reactive architectures and deliberative architectures. Reactive architectures are considered as simple controls, while deliberative architectures implement complex behavior including mental attitudes (goals etc.) and planning, based on symbolic representations and models. Hybrid architectures are combinations of both reactive control on the “lower level” for fast responses and deliberative control on the “higher level” for long-term planning.

A technologically better classification addresses the possible “states” of an agent, similar to the approach given in [86]. A state is a snapshot of the system (e.g. the content of the memory) at a certain time point on a discrete time scale. Transitions describe the state changes between two time points. For agents, the time scale is usually chosen according to the sense-think-act-cycle. This cycle consists of

– processing incoming information (“sense”, e.g. parsing messages from other agents, analyzing human requests, possibly in natural language etc.)
– more or less complex decision procedures (“think”, e.g. by simple decision or rules, or by deliberation, planning etc.)
– sending outgoing information (“act”, sending messages to other agents, preparing answers in a human-like style etc.).

An internet agent may in one cycle get a customer request, update its database and send an answer. The state is the content of the database at the end of this cycle. An agent may have different cycles at different time scales: A search engine may answer search requests more frequently than its index machine gets updated. “This creates a need for synchronization efforts which can be facilitated e.g. by different layers.”

A behavior (architecture) is called *stimulus-response* behavior if there are no different states at all. The response of the agent to an input is always the same (if there is some probabilistic component, then the distribution is the same). It can be produced e.g. by a fixed set of rules, by an input-output table or by a neural network. It doesn’t matter if the response routine is a simple or a complex one. A search engine may perform extensive search over large databases and a lot of effort to rank the results. If nothing is stored after answering, then the same calculations yielding an identical answer will be performed every time for the same request.

If the responses of an agent to identical requests are different, then they depend on the state of the agent. The search engine may maintain profiles of its users such that the answers depend on the stored profiles. The profile is updated every time the user makes a request, i.e. the state of the agent is changed. It is useful to distinguish between different kinds of states according to their contents:

The so-called *belief* or *world model* stores internal representations about the situation in the environment for later usage. It is updated according to the sensor inputs. It is called belief because it needs not to be true knowledge (e.g. the profile of a user calculated only by the available user inputs needs not represent her or his true preferences).

Future directed states are created by “mental attitudes” related with decision processes. They are named as *goals*, *plans* etc., and they guide the future deliberation and actions (towards a formerly chosen goal). A trading agent may have the goal of an optimal transaction. For that, it may develop a plan for searching appropriate offers from databases and for negotiation with other agents.

As introduced above, agents are called “*stimulus-response*” if they do not have states. Nevertheless, their decision procedures might be very complex, e.g. in complex information systems. Agents with states may have a world model (belief) and/or future-directed state components like goals and plans. Their deliberation process considers updates of the world model and commitment procedures for selecting goals and constructing plans. Actually, it is up to the programmer to decide if mental notions are used for data constructs. Sometimes, very simple agents come with mental attitudes. There is nothing wrong with it if it helps for better understanding.

The popular *BDI architecture* is inspired by the work of the philosopher [16]. BDI stands for *belief*, *desire* and *intention*. Belief describes the world model as above, while desires and intentions are future-directed mental notions. Bratman argues that the mental notion of goals is not sufficient to express complex future-directed behavior. A rational agent should adopt only goals which it believes to be achievable, i.e. not in conflict with the belief and not in conflict with each other. However, before committing to a goal, the agent may have different desires, which may be in conflict. A human may at the same time have conflicting desires, e.g. go to home, to be at the beach, to ride a bicycle, and to drive a car. Then he has to make a commitment, to choose which desires to adopt as intentions. Rational behavior demands to select only non conflicting options, e.g. to go by bicycle to the beach.

In BDI architectures, desires are used as preliminary stages of possible intentions: first the agent collects desirable options, and then it selects some of them as intentions e.g. through ranking, while avoiding conflicts between intentions. Then it performs appropriate actions to achieve the intentions.

Not all the so-called BDI-architectures really implement Bratman's ideas. In some cases, the agent simply selects a single desire and calculates an appropriate action sequence (called intention) which fulfills this desire. In such a case, a desire is in fact a goal, and the intention is the related plan to achieve the goal.

A lot of theoretical work using multimodal temporal logics has been performed for the foundation of deliberative agent controls and some of them resulted in executable formalisms like MetateM ([48]).

2.3. Robots and Software Agents

Robots are often considered as hardware controlled by a software agent acting as the brain. The sensors and actuators of the robot provide the input and output for the agent. This works well in simple settings, but it poses problems for more complex robots in real environments. Control of such robots is more than information processing: parts of such robots coordinate not only by messages, but by physical interactions too. It is very difficult or even impossible to model the physical dependencies in terms of information processing. However, those relations can be used directly by clever design. Modern robotic approaches are inspired by biology and use local sensor-actor loops, etc. "The key observation is that the world is its own best model. It is always exactly up to date. It always contains every detail there is to be known. The trick is to sense it appropriately and often enough." [109] This paradigm is known as *behavioral robotics*, *biologically inspired robotics* etc.

Related robot controls are able to perform surprisingly complex tasks. Their behavior *emerges* from the physical *situatedness* of an *embodied* entity. An approach that exploits situated automata is described in [66]. Pattie Maes [72] has developed an agent architecture that is composed of modules organized into a network. The *Subsumption Architecture*, based on behaviors, is the best-known architecture of this kind. Brooks built many robots (based on four principles) using an AI approach [25], [26]:

– *Situatedness* - The robots are situated in the world.
– *Embodiment* - The robots have bodies and experience the world directly - their actions are part of a dynamics with the world and have immediate feedback on their own sensations.
– *Intelligence* - They are observed to be intelligent - but the source of intelligence includes: computational engine, situation in the world, the signal transformations within the sensors, and the physical coupling of the robot with the world.
– *Emergence* - The intelligence of the system emerges from the system's interactions with the world and from sometimes indirect interactions between its components.

A behavior is implemented as a simple state machine with few states (not representations of the outside world). Behaviors can overwrite each other (subsumption). Brooks' robots built from connected behaviors are capable of performing some complex tasks with relatively simple programming [24]. Maes has shown that the same ideas can also be exploited in the design of software agents [73]. The behavioral agent architectures are sometimes considered as prototypes of reactive architectures [105]). While single behaviors are simple, their hierarchical combination into a more complex behavior becomes more and more complicated. Because of that, the *hybrid* architectures combine low-level approaches with classical reasoning approaches in hierarchical architectures. Such architectures may consist of several levels, where low levels can use behavioral (and possibly subsymbolic) architectures, and higher levels are usually deliberative (symbolic) ones. Symbolic modeling becomes necessary when sensing does not give enough information, or when planning is really needed.

Because of the “physics in the loop”, the assumptions usually connected with software agents are not fulfilled: Physical components do not behave like objects (or agents). This difference is recently stressed by the investigation of so-called Cyber Physical Systems [110], which are considered as distributed systems where the components perform information processes as well as physical processes. The interaction between the components is of physical and computational nature, as well.

2.4. More Features of Agents

Depending on different usages of agents, they can have a lot of different features. Such features are often used for classification as well. We have already discussed different basic architectures. Next we describe mobility, size intelligence and ability to adapt and to learn. Relations to other agents are described later in the section on Multi-Agent Systems. A collection of agent features will be given in table 1.

Mobility - Agents can be *static* or *mobile*. Static agents are permanently located at one place, while mobile agents can change their location. When a static agent wants some action to be executed at a remote site, it will send a message to an agent at that location with the request for the action. In a similar situation, a mobile agent would transmit itself to the remote site and invoke the action execution. There are a lot of benefits from usage of mobile agents [27] but if we wanted to get all of these benefits without a mobile agent, we would need a large amount of work and it would be practically almost impossible [1]. The advocated utility of mobile agents is to support optimization of sophisticated operations that may require strong interactivity between components or special computing facilities as encountered e.g. in negotiation, network management and monitoring, and load balancing for scientific computing. Mobility of software agents is closely related to the problem of code mobility in distributed programming with applications in operating systems and computer networks. Some problems related with mobile agents concern security and safety. A good overview of code mobility paradigms can be found in the reference paper [53].

Size and Intelligence - Agents can be of various sizes and can possess various amounts of intelligence. Generally, intelligence of a software agent is proportional to its size, so we can distinguish: *big-sized*, *middle-sized* and *micro agents*. It is difficult to make clear boundaries among these categories.

1. A big-sized agent occupies and controls one or more computers. It possesses enough competence to be useful even if it acts alone, without the other agents in a MAS. A big-sized agent can be as big and as intelligent as an expert system [63] with competences for expert problem solving, e.g. distributed medical care or plane ticket reservation.
2. A middle-sized agent is the one that is not useful without the other agents in a MAS or without additional software [6], [7], [8]. However, it is able to perform some non-trivial task(s). A user-interface agent that acts without other agents and performs some simple actions can also be classified as a middle-sized agent. Mobile agents are usually middle-sized agents.
3. Micro agents (also called the Society of Mind agents) [77] do not possess any intelligence. Minsky followed the idea that the intelligence emerges as a global effect of the overall activity of many simple and unintelligent agents.

Adaptation – Adaptive agents can adapt their behavior to different situations and changes in the environment. For example, a navigation system can adapt to changes in traffic (e.g. a traffic jam) and propose alternative routes. This makes adaptive agents more robust to non-predicted changes in a dynamic environment.

Learning - Agents can use learning capabilities for better performance. Learning can be done online, e.g. by data mining from data which are constantly collected through interaction with users (e.g. for profiles). Offline learning refers to training processes (e.g. for pattern recognition) prior to productive agent usage.

Agents may possess many features in various combinations. The following table is a slightly modified collection from [54]:

Table 1. Agent's features

Adaptivity	*Agents can adapt to unpredicted changes.*
Autonomy	An agent can act without direct intervention by humans or other agents and that it has control over its own actions and internal state
Benevolence	It is the assumption that agents do not have conflicting goals and that every agent will therefore always try to do what is asked of it.
Character (personality)	An agent has a well-defined, believable "personality" and emotional state.
Competitiveness	An agent is able to coordinate with other agents except that the success of one agent may imply the failure of others.
Cooperation or collaboration	An agent is able to coordinate with other agents to achieve a common purpose; non-antagonistic agents that succeed or fail together.
Coordination	An agent is able to perform some activity in a shared environment with other agents. Activities are often coordinated via plans, workflows, or some other process management mechanism.
Credibility	An agent has a believable personality and emotional state.
Deliberation	A deliberative agent decides for its actions by reasoning processes which may involve mental categories like goals, plans etc.
Embodiment	An embodied agent can interact with its environment by physical processes. This allows for emergent controls guided by sensor data without internal representations.
Emergent behavior	More complex behavior emerges by interaction of (simple) agents with each other (swarm intelligence) or with the environment (embodied agents, situated agents).
Flexibility	The system is responsive (the agents should perceive their environment and respond in a timely fashion to changes that occur in it), pro-active and social.
Goal directed	Agent behavior is guided by mental qualities like goals, which are results of deliberation. Then the agent tries to achieve the goal by appropriate actions.

Hybrid architecture	Combination of different architectures. Often with simple (reactive, stimulus response) control for low level behavior and deliberative control for high level behavior.
Inferential capability	An agent can act on abstract task specification using prior knowledge of general goals and preferred methods to achieve flexibility; goes beyond the information given, and may have explicit models of self, user, situation, and/or other agents.
Intelligence	An agent's state is formalized by knowledge and the agent interacts with other agents using symbolic language.
Interpretation ability	An agent is interpretive if it can correctly interpret its sensor readings.
"Knowledge-level" communication ability	The ability to communicate with persons and other agents with language more resembling human-like "speech acts" than typical symbol-level program-to-program protocols.
Learning	An agent is capable of learning from its own experience, its environment, and interactions with others.
Mobility	an agent is able to transport itself from one machine to another and across different system architectures and platforms.
Prediction ability	An agent is predictive if its model of how the world works is sufficiently accurate to allow it to correctly predict how it can achieve the task.
Proxy ability	An agent can act on behalf of someone or something acting in the interest of, as a representative of, or for the benefit of, some entity.
Personality (character)	An agent has a well-defined, believable "personality" and emotional state.
Proactiveness	An agent does not simply act in response to its environment; it is able to exhibit goal-directed behavior by taking the initiative.
Rationality	It is the assumption that an agent will act in order to achieve its goals, and will not act in such a way as to prevent its goals being achieved — at least insofar as its beliefs permit.
Reactivity	An agent receives some form of (sensory) input from its environment, and it performs some action that changes its environment in some way.
Resource limitation	An agent can only act as long as it has resources at its disposal. These resources are changed by its acting and possibly also by delegating.

Reusability	Processes or subsequent instances can require keeping instances of the class ‘agent’ for an information handover or to check and to analyze them according to their results.
Ruggedization	An agent is able to deal with errors and incomplete data robustly.
Sensor-actor coupling	Agents act by (direct) connections between sensors and actors. This can be used for reactive controls.
Situatedness	An agent (robot) is situated in its environment. Its behavior can be guided by physical interactions (e.g. sensor-actor coupling). This can be an efficient alternative to control using internal representations.
Social ability	An agent interacts and this interaction is marked by friendliness or pleasant social relations; that is, the agent is affable, companionable or friendly.
Sound	An agent is sound if it is predictive, interpretive and rational.
Stimulus response	A stimulus response agent has no internal state. It means that its responses are equal for equal inputs.
Temporal continuity	An agent is a continuously running process, not a "one-shot" computation that maps a single input to a single output, then terminates.
Transparency and accountability	An agent must be transparent when required, but must provide a log of its activities upon demand.
Trustworthiness	An agent adheres to laws of robotics and is truthful.
Unpredictability	An agent is able to act in ways that are not fully predictable, even if all the initial conditions are known. It is capable of nondeterministic behavior.
Veracity	It is the assumption that an agent will not knowingly communicate false information.

2.5. Multi-Agent Systems and Agent Communication

“*Distributed Problem Solving*” is performed by agents working together towards a solution of a common problem (e.g. for expert systems) [12]. *Multi-Agent Systems* (MAS) take a more general view of agents which have contact with each other in an environment (e.g. the Internet) [13]. The rules of the environment as well as the agent controls determine the form of *coordination*. The agents may be *cooperative* or *competitive*. Relations between local and global behavior in such MAS have been studied using game theory and social theories (cf. [101]).

Communication via exchange of messages is the usual prerequisite for coordination. Nevertheless, cooperation is possible even without communication, by observing the environment. The two most important approaches to communication are using protocols and using an evolving language [28]. Both have their advantages and disadvantages. For industrial applications, communication protocols are the best practice, but in systems where homogeneous agents can work together, language evolution is the more acceptable option [28]. *Agent Communication Languages* (ACLs) provide important features like technical declarations (sender, receiver, broadcasting, peer-to-peer, …), speech act (query, inform, request, acknowledge,…) and content language (e.g. predicate logic). Together with these features, related protocols are defined to determine the expected reactions to messages (e.g. an inform message as an answer to query message). A number of languages for coordination and communication between agents was enumerated in [102]. The most prominent examples [102] are given in Table 2.

Table 2. Languages for coordination and communication between agents.

Agent communication language	***Description***
KQML ("Knowledge Query and Manipulation Language")	It is perhaps the most widely used agent communication language [102], [45]. KQML uses speech-act performatives such as reply, tell, deny, untell, etc. Every KQML message consists of a performative and additional data written in several slots. Some slots are :content, :in-reply-to, :sender, :receiver, :ontology, etc. The set of performatives in KQML and their slots should be general enough to enable agent communication in every agent application. There are claims that there might be some problems with the semantics of performatives. Various agents may interpret the same performative in various ways.
FIPA-ACL ("FIPA Agent Communication Language")	It is an agent communication language that is largely influenced by ARCOL [102]. FIPA ACL has been defined by FIPA - Foundation for Intelligent Physical Agents. Together FIPA-ACL [47], ARCOL, and KQML establish a quasi standard for agent communication languages [102]. Syntax and semantics of FIPA ACL are very similar to the syntax and semantics of KQML. Time will show which one of these two standards will prevail.
ARCOL ("ARTIMIS COmmunication Language")	ARCOL has a smaller set of communication primitives than KQML, but these can be composed. This communication language is used in the ARTIMIS system [102].

KIF ("Knowledge Interchange Format")	This logic-based comprehensive language with declarative semantics has been designed to express different kinds of knowledge and meta-knowledge [102]. KIF is a language for content communication, whereas languages like KQML, ARCOL, and FIPA-ACL are for intention communication.
COOL ("Domain independent COOrdination Language")	COOL relies on speech-act based communication, aims at explicitly representing and applying coordination knowledge for multi-agent systems and focuses on rule-based conversation management (conversation rules, error rules, continuation rules, …) [102]. Languages like COOL can be considered as supporting a coordination/communication (or "protocol-sensitive") layer above intention communication.

Contract Net Protocols and *Blackboard Systems* are well understood mechanisms for organizing MAS. Contract Net Protocols organize the distribution of tasks to other agents by announcing tasks, receiving bids from other agents and choosing one of the bidding agents for execution. Blackboard systems provide a common active database (the blackboard) for information exchange.

MAS with many agents are often used for simulations to study *Swarm Intelligence* and for social simulations in the field of *Socionics.* Social simulations include simulations of financial markets, traffic scenarios, and social relationships. Swarm intelligence can lead to complex "intelligent" behavior which emerges from the interaction of very simple agents, e.g. in ant colonies or in trade simulations. Complex problems, e.g. the well-known travelling salesman problem can be solved with swarm techniques.

3. Languages for constructing Agent-based systems

An essential component of agent-based technology and implementation of agent-based systems is a programming language. Such a language, called an *agent-oriented* programming language, should provide developers with high-level abstractions and constructs that allow direct implementation and usage of agent-related concepts: beliefs, goals, actions, plans, communication etc.

Most agent systems are still probably written in Java and C/C++ [102]. Although traditional languages are not well-suited for agent systems, it is achievable to implement them in Pascal, C, Lisp, or Prolog languages [79]. Typically, object-oriented languages (Smalltalk, Java, or C++) are easier to use for realization of agent systems as agents share some properties with objects such as encapsulation, inheritance and message passing but also differ definitely from objects vis-à-vis polymorphism [79]. Apart from these standard languages, several prototype languages for implementing agent-based systems have been proposed to support better realization of agent-specific concepts.

Devising a sound classification and analysis methodology for agent programming languages is a very difficult task because of the highly-dimensional and sometimes interdependent heterogeneous criteria that can be taken into account, e.g. computational model, programming paradigm, formal semantics, usage of mental attitudes, architectural design choices, tools availability, platform integration, application areas, etc. Therefore, here we take a more pragmatic approach by firstly proposing a light top-level classification that takes into account those aspects that we consider most relevant for agent systems, i.e. the usage of mental attitudes. According to this classification we find: agent-oriented programming (AOP) languages; belief-desire-intention (BDI) languages, hybrid languages (that combine AOP and BDI within a single model), and other (prevalently declarative) languages. Understanding the current state of affairs is an essential step for future research efforts in the area of developing agent-oriented programming languages.

Table 3 (at the end of the paper) brings summary and specific information for all agent languages presented in the paper that we managed to collect from different sources: Web Page, IDE, Implementation language, Agent platform integration, Applications, Paradigm, and Textbook.

3.1. Agent-oriented programming model

The term *Agent-oriented Programming* (AOP) was coined in [90] to define a novel programming paradigm. It represents a computational framework whose central compositional notion is an agent, viewed as a software component with *mental qualities*, *communicative skills* and a *notion of time*. AOP is considered to be a specialization of object-oriented programming (OOP), but there are some important differences between these concepts ([107], [90]). Objects and agents differ in their *degree of autonomy*. Unlike objects, which directly *invoke* actions of other objects, agents *express their desire* for an action to be executed. In other words, in OOP the decision lies within the requesting entity, while in AOP the receiving entity has the control over its own behavior, by deciding whether an action is executed or not. Also, agents can often have *conflicting* interests, so it might be harmful for an agent to execute an action request from another agent. An additional difference is *flexibility*. Agents often exhibit pro-active and adaptive behavior and use learning to improve their performance over time. *The thread of control* is the final major difference. While multi-agent systems are multi-threaded by default, there is usually a single thread of control in OOP.

An important part of the AOP framework, as described in [90], is a programming language. *Agent-orient Programming Language* (APL) is a tool that provides a high-level of abstraction directed towards developing agents and incorporates constructs for representing all the features defined by the framework. Most of all, it should allow developers to define agents and bind them to specific behaviors [87]; represent an agent’s knowledge base, containing its mental state; and allow agents to communicate with each other.

The AOP paradigm was very influential for the further development of agent programming languages, resulting in a number of languages.

AGENT0

AGENT0 (see Table 3) [90], [107] and [9], was the first agent-oriented programming language that has been developed, providing a direct implementation of the agent-oriented paradigm. Although being more of a prototype than a “real” programming language, it gives a feel of how a large-scale system could be built using the AOP concept.

In AGENT0, an agent definition consists of four parts: a set of capabilities (describing what the agent can do), a set of beliefs, a set of commitments or intentions, and a set of commitment rules containing a message condition, a mental condition and an action [107]. Agents communicate with each other through an exchange of messages which can be one of three different types: (1) a request for performing an action, (2) an “unrequest”, for refraining from an action, and (3) an informative message, used for passing information. Usually, *requests* and *unrequests* result in agent’s commitments being modified, while an *inform* message results in a change in agent’s beliefs. Furthermore, a message can be private, corresponding to an internally executed subroutine, or public, for communication with other agents in the environment. These messages can alter agent’s beliefs and commitments, i.e. its mental state. A crucial task is, therefore, to maintain the agent’s mental state in a consistent form. As proposed in [90], there are three different ways of achieving this: 1) Using formal methods and mathematical logic; 2) Heuristic methods; 3) Making the language for mental space description as simple as possible, thus enabling trivial verification (the solution applied in AGENT0). Shoham [90] proposes the model for agent execution using a simple loop, which every agent regularly iterates: 1) Read the current messages, and, if needed, update set of beliefs and commitments; 2) Execute all commitments for the current cycle. This can result in further modifications of beliefs.

PLACA

Planning Communicating Agents - PLACA (see Table 3) is an improvement of the AGENT0 language, extending it with *planning facilities*, which significantly reduce the intensity of communication ([95], [9]). In PLACA, an agent doesn’t need to send a separate message each time it requests another agent to perform some action. Instead, it can provide another agent with a description of the desired final state. After checking the rule conditions are satisfied, as well as by using its planning abilities, the receiving agent presents to the sender a plan of actions to execute in order to reach the desired state. This means that the agents communicate requests for actions via high-level goals.

The logical component of PLACA is similar to that of AGENT0, but it includes operators for planning. Due to introduction of plans, mental categories and syntax in PLACA are a bit different than those in AGENT0. If a received message satisfies the message condition and if the current mental state of the receiving agent satisfies the mental condition, then the agent's mental state will be changed and the agent will send appropriate messages.

PLACA, as AGENT0, is an experimental language, not designed for practical use.

Agent-K

Agent-K (see Table 3) is another extension of the AGENT0 [35]. It replaces custom communication messages (i.e. *request*, *unrequest* and *inform*) with the standardized KQML. This improves the general interoperability of agents and enables them to communicate with different types of agents (that employ KQML as well). It doesn't, however, include the improvements brought by PLACA. Merging of the two concepts is achieved by modifying the AGENT0 interpreter to handle KQML messages. Since the interpreter is implemented in Prolog, an intermediate level was introduced to convert the Lisp-style format of KQML messages into an unordered Prolog list of unary predicates. In addition, this layer transforms textual parts of a KQML message into tokens that can be handled by the interpreter. The interpreter has been modified to include these changes and to allow multiple actions to occur at the same time, i.e. when there is a match between an incoming message and multiple commitment rules. Because of that, each Agent-K agent is a separate process with its own instance of the interpreter.

As an addition, Agent-K uses the KAPI[1] library for agent communication, which can transport KQML messages over TCP/IP and e-mail to remote systems. Although the integration with KQML should improve the interoperability of agents, this was not fully achieved [35] because Agent-K uses Prolog to encode agents' beliefs and commitments (thus restricting the communication to other Prolog-based agents only). Authors of the language propose another language for knowledge representation, (e.g. KIF), to be used.

MetateM

The Concurrent MetateM, currently called simply MetateM (see Table 3), [14], [15] is probably one of the oldest programming languages for multi-agent systems. It was based on the direct execution of logical formulae [49], [106].

[1] The KAPI library is provided by Jay Weber, EIT

MetateM has its roots in formal specification using temporal logic by bringing in the idea of executable temporal specifications. Therefore, it can be equally well described as a temporal logic programming language that is based on temporal rather than on first-order logic.

A MetateM agent program consists of a set of temporal rules that are built according to the paradigm: *declarative past and imperative future*. Intuitively this means that: (i) the conditional part of the rules is interpreted declaratively by matching it with the history of the agent execution, i.e. what is true in the current state of the agent and what was true in the past states of the agent, and (ii) the execution part of the rules represents the choices that the agent is facing in the next state, as well as in future states. So, intuitively, the execution of a MetateM program is in fact the process of building a concrete model of the program specification using a forward chaining algorithm.

The current implementation of MetateM [123] is based on Java and it supports asynchronous and concurrent execution of multiple agents that are able to exchange messages such that each message sent is guaranteed to arrive at a future moment in time. Moreover, MetateM supports a dynamic structuring of agents based on two sets of agents that are associated with each agent in the system: (i) the *content* set representing those agents that the current agent can control, and (ii) the *context* set representing those agents that can influence the current agent. This style of grouping allows efficient agent communication using multicast messages [50].

April and MAIL

Agent PRocess Interaction Language - April (see Table 3) [74] is a process-oriented symbolic language that was not designed specifically for agent-oriented programming, but rather as a general-purpose multi-process development tool. Nevertheless, it provides the necessary infrastructure for developing and employing simple agents. The main entity in an April system is a *process*, which represents an agent in the multi-agent paradigm. An agent is identified by its private or public handle. Private handles are accessible within the system only, while the public handles are available to agents in other systems as well. Public handles are registered in the system's *name server* and as such can be found from other systems connected to it. These inter-connected name servers allow one to build a global April application.

April has a simple communication infrastructure that uses TCP/IP and permits access to non-April based applications. Agents communicate by exchanging messages identified by their handles. If two agents send a message to a third agent, the April system cannot guarantee that they will arrive in the order of transmission, since there is no global synchronization clock. What can be assured is that if one agent sends *n* messages to another, they will arrive in the order they were sent, but it is not always possible to determine how much time an operation will take to execute. Therefore, "April is not particularly suitable for time-critical real-time applications" [74].

A powerful feature offered by April is *macro*, which gives developers the ability to define new language constructs, based on the existing ones. One of the main purposes of macros was to serve as tools for developing new, richer and more agent-oriented languages on top of April. Concepts as messages based on a particular speech-act, agent mobility, knowledge handling etc. can also be simulated [74]. Authors of April intended to include these extensions in a more developer-friendly manner and to create a new agent-oriented programming language called MAIL. MAIL as a high level language was intended for realization of many common MAS. First version of April and MAIL specification was the subject of an ESPRIT project and April had to serve as implementation language for MAIL, in fact as the intermediary between C and MAIL. MAIL was prototyped using IC-Prolog II (distributed logic programming system). Funding for the project was cancelled before it was implemented.

VIVA

VIVA (see Table 3) [98], an agent-oriented declarative programming language, was based on theory of VIVid agents introduced by same author. A VIVid agent is a software-controlled system with state expressed in a form of beliefs and intentions (as mental categories) and with behavior represented by action and reaction rules. Its basic functionality covered possibility to represent and perform actions in order to generate and execute plans. VIVA was in accordance with agent-oriented programming paradigm, but it was slightly conservative as it adopted as many concepts as possible from Prolog and SQL. The basic design principles of VIVA apart from conservativeness were scalability and versatility [98].

An agent specified in VIVA could run on a number of hosts with the same or different hardware/software architectures. The composition of MAS and the locations of participating agents had to be specified before a VIVA application could run.

The language was intended for general-purpose software agent programming, embedded systems and robots but has not fulfilled expectations of the authors to be widely used in MAS.

GO!

Multi-paradigm programming language GO! (see Table 3) [32] is conceptually similar to April. It combines OOP, concurrent, logic and functional paradigms into a single framework. Based on April, GO! brings following extensions: knowledge representation features of logic programming, yielding a multi-threaded, strongly typed and higher order language (in the functional-programming aspect) [21]. In inheritance from April, threads primarily communicate through asynchronous message passing. Threads, as executing action rules, react to received messages using pattern matching and pattern-based message reaction rules. A communication daemon enables threads in different GO! processes to communicate transparently over a network. Each agent usually can encompass several threads directly communicating with threads in other agents. Threads within a single GO! process can also communicate by manipulating a shared cell or dynamic relation objects.

As a strongly typed language it can improve code safety as well as reduce the programmer's burden. New types and new data constructors can be easily added. The designers of the language have had in mind critical issues like security, transparency and integrity, in regards to adoption of the logic programming essence. Features of Prolog like the cut ('!') have been left out for obvious reasons. In Prolog, the same clause syntax is used for defining relations (declarative semantics), and for defining procedures (operational semantics). In GO!, however, behavior is described using action rules expressed in a specialized syntax.

3.2. BDI based languages

A significant and influential trend in designing agent programming languages stemmed from the success of the practical reasoning agent architectures, among which the most notable is probably the PRS – Procedural Reasoning System [55]. PRS became the first system embodying a belief, desire, and intention (BDI) architecture. Based on that, approximately around the same time with Shoham, Rao proposed the AgentSpeak(L) language [83]. AgentSpeak(L) employs the metaphors of belief, desire, and intention of the BDI architecture to shape the design of an innovative agent programming language. However, AgentSpeak(L) was only a proposal, while Jason programming language became in 2004 the first implementation of an interpreter for an extended version of AgentSpeak(L) [22], [121]. AgentSpeak(L) is often described as a BDI agent programming language, as it is assumed to convey the most important ideas of BDI agent architectures (including the PRS).

In this section we will present several important agent programming languages which support BDI architecture and belong to the hybrid paradigm.

AgentSpeak

The language was originally called AgentSpeak(L) (see Table 3), but became more popular as AgentSpeak. This term is also used to refer to the variants of the original language. The primary goal of the authors of AgentSpeak [100] was to join BDI architectures for agents and for object-based concurrent programming and to develop a language that would capture the essential features of both. They identified the primary characteristics of agents: complex internal mental state, proactive or goal-directed behavior, communication through structured messages or speech acts, distribution over a wide-area network, adequate reaction to changes in the environment, concurrent execution of plans and reflective or meta-level reasoning capabilities.

The basic construct in AgentSpeak is an *agent family* and its purpose is analogous to a class in object-oriented languages. Each agent (an instance of an agent family) contains a *public* and a *private* area which are, respectively, offered to other agents or used for agent's internal purposes. An agent's behavior is described using three different concepts: *database relations*, *services* and *plans*. Agents execute actions in order to meet their own, or desires of other agents. For fulfilling its own desires, an agent uses a set of private services (inner *goals)*, while other agents can invoke its public services (corresponding to messages from other agents). In AgentSpeak there are three distinct types of services for different purposes:

– *Achieve-service*: used to achieve a certain state of the world
– *Query-service*: used to check whether something is true, considering the associated database
– *Told-service*: used to share some information with another agent.

Once a service has been invoked, an agent proceeds to execute it by the means of plans. Once a plan has been activated, its *goal statements* are executed. Upon successful execution of all goal statements, the reached state is assessed in order to make sure that the desired state of affairs has been achieved.

Agents communicate in AgentSpeak by exchanging messages, either asynchronously (default) or synchronously. A message can be sent to a specific agent or to an agent family, in which case it is forwarded to all instances of that family. If a message sent to another agent contains some information, but puts no obligation upon the receiving agent, it is called an *inform* speech-act. Otherwise, it's a *request*. In addition, a message can have a *priority* assigned to it, thus giving it an overall importance.

In recent times, a work on *Coo-AgentSpeak* has been published in [2]. It incorporates ideas presented in *Coo-BDI* [3] into AgentSpeak. Coo-BDI extends the standard BDI model with cooperation, allowing agents to exchange their plans for satisfying intentions.

Jason

Jason (see Table 3) is probably the first implementation of AgentSpeak(L) using the Java programming language and belongs to the hybrid agent paradigm [22]. The syntax of Jason exhibits some similarities with Prolog. However, the semantics of the Jason language is different and it is based on AgentSpeak(L). One strength of Jason is that it is tightly integrated with Java with the following immediate consequences: (i) the behavior of the Jason interpreter can be tailored using Java; (ii) Jason can be used to build situated agents by providing a Java API for integration with an environment model that is developed with Java; (iii) Jason has been integrated with some existing agent frameworks, including JADE [18], AgentScape [97], and Agent Factory [113].

AF-APL

Agent Factory Agent Programming Language - AF-APL (see Table 3) is the core of Agent Factory agent development environment. AF-APL is originally based on Agent-Oriented Programming [90], but was revised and extended with BDI concepts (hybrid paradigm). AF-APL is described as a “practical rule-based language” based on *commitment rules*. A commitment rule joins together three types of mental attitudes: *beliefs*, *plans*, and *commitments*. The syntax and semantics of the AF-APL language have been derived from a logical model of how an agent commits itself to a course of action [33], [85]. The semantics of AF-APL was formalized in Rem Collier’s Ph.D. thesis using multi-modal first-order branching-time logic [33].

An AF-APL programmer can declare explicitly, for each agent, a set of sensors (*situated* agents) referred to as perceptors and a set of effectors (actuators). Perceptors are in fact instances of Java classes which define how to convert raw sensor data into beliefs. An actuator is realized as an instance of a Java class with responsibilities: 1) to define the action identifier that should be used when referring to the action (realized by the actuator); 2) to contain code that implements the action. These declarations, specified within the agent program, are termed the embodiment configuration of the agent.

The AF-APL programming language is strongly related to the Agent Factory framework for the development and deployment of agent systems (see [76] for a recent overview and applications of Agent Factory framework).

3APL

3APL (see Table 3) [60] is not explicitly declared a descendant of either AgentSpeak(L), or Agent0. However, in our opinion it was clearly influenced by both AOP and BDI families of languages, and more important, both families of languages were clearly influenced by the general settings of the *intentional stance* towards understanding and development of a software system [38]. It is interesting to note that 3APL was theoretically shown to be at least as expressive as AgentSpeak(L) [59]. However, although it’s Web page is still alive [111], we have noticed that 3APL language and supporting tools do not seem to be further developed. Rather, one of its authors, Koen V. Hindriks switched to the development of a new language GOAL. Nevertheless, 3APL is still relevant as it has opened the new direction of *goal-oriented agent-programming languages* and in some sense it has unified ideas from AOP, BDI and logic within a single programming model with declarative goals (hybrid paradigm). Moreover, there is an explicitly declared successor of 3APL called 2APL that is currently being developed [133]. 3APL has been applied to robot control using an API called ARIA (provided by ActivMedia Robotics[2]).

[2] http://www.activmedia.com/

2APL

2APL (see Table 3) is the successor of 3APL, enhancing it in many aspects. Probably the most important aspect is the clear separation of multi-agent and individual agent concerns. The multi-agent part is addressing the specification of a set of agents, a set of external environments and the relations between them, i.e. agent – agent and agent – external environment relations. The individual agent concepts in 2APL cover beliefs, goals, plans, events, messages, and rules, so it has many similarities with the programming notions that are available in other BDI and AOP languages. 2APL amalgamates declarative and imperative programming styles, so it can be described as hybrid (in the sense of the classification from [21]). Probably this is the most notable difference between 2APL and GOAL, as GOAL is clearly a declarative programming language, while 2APL is described by its authors as a "practical agent programming language". 2APL has been designed to work with JADE and, in comparison with 3APL, it provides practical extensions that allow better testing and debugging [34].

JACK Agent Language

JACK^{TM} Intelligent Agents, or simply JACK (see Table 3), is a commercial agent platform provided by Autonomous Decision Making Software – AOS [130]. The main JACK components are: *JACK Agent Language* (also known as JAL), *JACK compiler*, *JACK kernel*, and *JACK Development Environment*. JAL is a superset of Java that incorporates the full Java language and provides the necessary constructs for building agent-oriented programs according to the BDI model. JAL is translated into JAVA source code using the *JACK compiler*, and the resulting Java code can be run on top of the JACK runtime engine, also known as *JACK kernel*. JACK Development Environment is an integrated graphical environment for the development of JACK multi-agent applications.

JACK supports the development of distributed agent applications by allowing agents to be deployed in separate processes, possibly running on different networked machines. JACK agents are able to exchange messages in a peer-to-peer fashion, as well as they are able to find each other using name servers. JACK and supporting tools are reviewed in [103].

JADEX

JADEX (see Table 3) is a Java-based agent platform that tries to respond to three categories of requirements: openness, middleware, and reasoning, thus bridging the gap between middleware-centered and reasoning-centered systems [82], [120]. The architecture of a JADEX agent follows the Procedural Reasoning System (PRS, [55]) computational model of practical reasoning. Agents in JADEX communicate by exchanging messages. Internally, an agent reacts to events in its execution cycle that combines reaction and deliberation processes.

A JADEX agent uses the concepts of BDI agents: beliefs, desires (goals in JADEX), and intentions (plans in JADEX). JADEX employs an object-oriented representation of beliefs. Additionally, beliefs have an active role, i.e. their update can trigger generation of events or adoption/dismissing of goals. JADEX uses four types of goals: (i) *perform* goal designating action execution; (ii) *achieve* goal designating a point-wise condition in the lifecycle of an agent that must be reached; (iii) *query* goal that is an introspection mechanism by which an agent is inspecting its own internal state; (iv) *maintain* goal designating a process-wise condition that must be maintained during the agent's execution. JADEX plans represent the behavioral aspect of an agent and they have a procedural flavor. A plan consists of a *head* and a *body*, similarly to a procedure in a procedural language.

The JADEX language combines the declarative specification of an agent containing its set of beliefs, goals and plans using an Agent Definition File (ADF) and the procedural specification of the plan bodies using the Java programming language. The plan body accesses the internals of an agent through a specialized API. JADEX agents are able to run on the JADE middleware platform, thus enabling the development of distributed intelligent systems using the BDI metaphor.

3.3. Other Agent Languages

Within the generic class of "other languages" we include all those agent programming languages that do not explicitly employ mental attitudes for shaping the language, but rather use other constructs that are very useful for building intelligent software agents by supporting reasoning tasks based on formal logic, methods and calculi set on top of the main characteristics attributed to agents. Compared to AOP and BDI, this category can be characterized as a more traditional to agent programming from the point of view of computer science practices.

During the period of developing different agent-oriented programming languages, some authors and research groups proposed and implemented languages essentially based on and characterized as the declarative paradigm. In this section we will present several important agent programming languages which support the declarative paradigm.

GOAL[3]

The main motivation behind the development of Goal-Oriented Agent Language, i.e. GOAL (see Table 3) was to bridge the gap between agent logics and agent programming models (BDI and AOP) [61]. This new language introduces a declarative perspective of goals in agent programming languages by unifying the concepts of commitments from Agent0, intentions from AgentSpeak(L) and goals from 3APL. An interesting feature of GOAL is that it sets on a clean and unified theoretical basis the concepts of reasoning and knowledge representation from AI with the mentalist notions that are more specific to agent programming. The GOAL agent programming language was recently overviewed in [62]. According to this reference, GOAL has been tested on top of JADE. However, we were not able to find any references to such an experiment. The current implementation of GOAL [132] is just a prototype that is currently mainly used for educational purposes. However, it can be also useful in planning applications, for example in the transportation and logistics domain.

Golog

”alGOl in LOGic” – GOLOG (see Table 3), is a family of logic languages (declarative paradigm) based on the formalism of *situation calculus* that was developed in AI by John McCarthy for the specification of dynamic systems [75]. Situation calculus is a first-order logic language with some second-order extensions that utilizes the following concepts: (i) action; (ii) situation; and (iii) fluent. Changes in the world are modeled using the *action* concept. Histories of the world are modeled using the *situation* concept; a situation is in fact a sequence of actions. *Fluents* represent relations and functions that depend on the situation, thus we have *relational fluents* and *functional fluents*.

According to [118], the GOLOG family comprises the following languages: (i) GOLOG, the core language, initially introduced in [84]; (ii) ConGOLOG, i.e. Concurrent GOLOG, an extension of GOLOG for handling concurrency [37]; (iii) IndiGOLOG: Incremental deterministic GOLOG [37].

Recently, it was shown that the BDI-style of agent programming can be achieved with GOLOG [88], thus bridging the gap between BDI and action logic styles of agent programming.

In our literature review we have found that GOLOG was quite influential in the area of programming physical robots endowed with cognitive capabilities. This trend spawned a number of extensions of GOLOG. ICPGOLOG is an

[3] Note that the GOAL agent programming language developed by Koen V. Hindriks is not the same thing as the GOAL agent programming language proposed by (Byrne and Edwards, 1996) in Byrne, C. ; Edwards, P.: Refinement in Agent Groups. In: Weiß, G. ; Sen, S. (Eds.): Proceedings of the IJCAI’95 Workshop on Adaption and Learning in Multi-Agent Systems, Lecture Notes in Computer Science 1042, Springer, 1996, pp. 22–39. Byrne’s GOAL is a direct descendant of Agent-0 and it was proposed earlier than Hindriks’s GOAL.

extension of GOLOG with actions to describe continuous change, support for noisy sensors and effectors, and probabilistic actions [39]. Implementation of ICPGOLOG was based on the existing implementation of IndiGOLOG in Prolog. READYLOG is a robot programming and planning language that adds to GOLOG the logic specification of MDP theories for decision-theoretic planning [43]. A novel prototype implementation of the GOLOG interpreter using the Lua scripting language for the bi-ped robot platform Nao was recently reported in [44].

Although one can notice that the main focus of GOLOG was to model single robotic agents, there were also works that propose GOLOG extensions for multi-agent systems in a game-theoretic setting, namely GTGOLOG [46].

FLUX

Fluent executor – FLUX (see Table 3) is a logic programming language (declarative paradigm) based on *fluent calculus* [93]. Fluent calculus is an axiomatic theory of actions that represents an improvement of situation calculus [75], since in fluent calculus situations represent state descriptions, while in situation calculus they represent histories of action occurrences. Thus, FLUX has a declarative semantics. The language is extensively described in the textbook [94]. An important difference between FLUX and many other agent programming languages is that the main focus of FLUX is on programming single agents that act logically in a dynamic environment, rather than developing complex multi-agent systems. In this respect, FLUX is similar to GOLOG. There are works, however, that describe a practical multi-agent system that contains a set of agents, each one equipped with a FLUX interpreter, that cooperate to solve a complex problem [89]. The current implementation of FLUX is based on constraint logic programming systems (Eclipse Prolog and Sicstus Prolog) for efficient handling of the axioms of fluent calculus.

CLAIM

Computational Language for Autonomous, Intelligent and Mobile agents – CLAIM (see Table 3) is a high-level agent programming language that combines the basic functionalities required for the agent model with higher-level support specific to intelligent and cognitive abilities (belongs to the hybrid paradigm). An important characteristic of CLAIM is its built-in support for agent mobility that is based on the abstract computation model of *ambient calculus* [30]. CLAIM agents are hierarchically structured (according to the formal model of ambients), goal-directed, knowledge-based, able to communicate at knowledge level, and mobile. CLAIM agents are not entirely declarative, as they mix declarative characteristics required for the specification of the knowledge component with imperative capabilities, required for the specification of the capabilities component. CLAIM is part of

Hymalaya unified framework and it is supported by SyMPA distributed multi-agent platform. Unfortunately, there is not much information about any of them, excepting the research papers [91] and [41].

4. Tools and Platforms

Multi-agent systems are deployed and run over specialized software infrastructures that provide the set of functionalities vital for the existence of a realistic multi-agent application. Seen from the perspective of distributed systems technologies, such infrastructures are placed at the middleware level and they include a collection of software functionalities and services that assure: agent lifetime management, agent communication and message transport, agent naming and discovery, mobility, security, etc. An *agent framework* is a software infrastructure available as a software library, a language environment, or both, which provides the core software artifacts needed for creating the skeleton of a multi-agent system. A software package that provides the core functionalities for deploying and running multi-agent applications is traditionally known as an *agent platform* [96]. An *agent toolkit* is a more complex software infrastructure that allows both the development and deployment of a multi-agent system [69]. It is sometimes known as an *agent development environment* [96], because of its expected support for all engineering stages of a multi-agent application from requirements to deployment, maintenance and evolution.

Most often, a multi-agent system is deployed and runs on top of an agent platform. If an agent platform is not available, at least an agent framework is usually utilized to create the multi-agent system which is then run on a general purpose middleware platform. Agent code can be programmed either using a general-purpose programming language linking with software libraries available in the agent framework via the framework API, or using one of the agent programming languages (see the previous section).

Agent platforms can be extremely useful because they considerably simplify the development and deployment of a multi-agent system. There is the option to choose between standardized or not-standardized agent platforms. A standard agent platform is compliant with available standards for software agents. Compliance to standards is important for open systems, i.e. systems that might need to interoperate in the future with other systems that are either not available at the moment when the open systems are being developed or that, even if they are available at the moment, still might change in the future.

According to our literature review, more than 100 agent platforms and toolkits were developed (or started to be developed) [69] of different quality and maturity. Most of them are built on top of and are integrated with Java [102]. Despite this fact that clearly shows that software agent technologies triggered a significant initial interest and hope, only few of them are still currently available, while the rest either became obsolete or are not being developed anymore. In the rest of the section, we provide a brief review of some of them. Although our selection might look quite subjective, we have done our best to consider those agent platforms and toolkits that we think are most influential, currently active and also well supported by the open source and/or business communities. Note that some of the platforms considered in this paper are also overviewed in more detail by [96].

4.1. ZEUS

ZEUS [129], [4], [81], [80] developed by British Telecommunications Labs, is a collaborative agent building environment that has excellent GUI and debugging, provides library of predefined coordination strategies, general purpose planning and scheduling mechanism, self-executing behavior scripts, etc. ZEUS is one of the most complete and the most powerful agent tools which are used to design, to develop and to organize agent systems. The aim of ZEUS project was to facilitate the rapid development of multi-agent applications by abstracting into a toolkit the common principles and components underlying some existing multi-agent systems [78]. It enables applications with additional assistant tools, e.g. reports and statistics tools, agents and society viewer, etc. ZEUS documentation is very weak, which leads to difficulties in creating new applications. The three main functional components of ZEUS are (adapted from [129]): The Agent Component Library; The Agent Building Tools; The Visualization Tools.

Some characteristics of ZEUS are: it implements FIPA standards, supports KQML and ACL communication and security policy supports ASCII-encoded, Safe-Tcl scripts or MIME-compatible e-mail messages for transportation; it uses public-key and private-key digital signature technology for authentication, cash and secrecy.

4.2. JADE

Java Agent DEvelopment Framework - JADE is probably one of the most popular agent platforms that are currently available to the open source community. JADE is FIPA-compliant and it is well supported by documentation [119], a textbook [18] and an enthusiastic community of users.

A JADE agent platform can be distributed on multiple machines that run the Java virtual machine, while multiple platforms can interoperate via FIPA standards. A platform consists of multiple containers, while each container can contain zero or more JADE agents. There is exactly one *Main* container and, optionally, zero or more ordinary containers, linked to the *Main* container. The JADE containers can be distributed onto the nodes of a local area network. Each node can host several containers. Each JADE agent contains its own execution thread. Unfortunately, this design choice is one of the main limitations for the number of agents that can be created and executed on a single machine. JADE agents use a specialized execution model based on non-preemptive scheduling of dynamically loadable JAVA plugins called *behaviors*. The agent execution model combined with JADE's intuitive programming interface allows the programmer to relatively easily develop software agents that are capable of flexible reactive and/or proactive behaviors. JADE agents can interact by asynchronously exchanging FIPA ACL messages, optionally following FIPA interaction protocols [116]. Telecom Italia is currently used JADE as reference framework for Network Neutral Element Manager – NNEM project [19].

4.3. agenTool

agenTool is a Java-based graphical development environment/tool that supports the Multi-agent Systems Engineering (MaSE) methodology [39] originally developed at the Artificial Intelligence Lab of the Air Force Institute of Technology, Ohio. It implements all MaSE steps including conversation verification and code generation. One of its most interesting abilities is the possibility to work on different pieces of the system and at various levels of abstraction interchangeably, which mirrors the ability of MaSE to incrementally add detail [39]. During each step of system development it is possible to use various analysis and design diagrams. . Moreover, it is possible to transform a set of analysis models into appropriate design models using semi-automatic transformations. Some efforts have been done in order to support modeling of mobile agents.

4.4. RETSINA

Reusable Environment for Task-Structured Intelligent Networked Agents – RETSINA is a multi-agent system toolkit that has been developed since 1995 at the Intelligent Software Agents laboratory of Carnegie Mellon University's Robotic Institute [125].

RETSINA is probably one of the earliest, most influential software infrastructures for developing multi-agent systems. It supports the development of communities of heterogeneous agents that can engage in peer-to-peer relations without imposing any centralized control for agent management. A RETSINA-based multi-agent system is platform independent, being able to run on various operating systems, while its agents can be implemented using different general-purpose programming languages. RETSINA is using a multi-agent software infrastructure based on Agent Foundation Classes – AFC. A very good overview of the distributed software infrastructure of RETSINA is provided by [92].

RETSINA was utilized for developing an impressive number of applications in various areas: military operations, critical decision making, supply chain management, financial portfolio management, text mining, etc [125], [92].

4.5. JATLite

‘Java Agent Template, Lite’ - JATLite [65] has been developed at the Stanford Center for Design. The intention was to allow creating software *typed-message agents* communicating over the Internet. Agents communicated using typed messages in an agent communication language like KQML, in which some semantics are defined before runtime. Two additional requirements had to be fulfilled: *Reliable message delivery and Migrating agent communication*.

JATLite added basic infrastructure functionality that earlier systems missed, supporting buffered-message exchanges and file transfers with other agents on the Internet, as well as connection, disconnection, and reconnection in the joint computation [65]. Security aspects of JATLite message relied on current open standards for encryption and authentication. The one simple feature that JATLite added was a password associated with the agent name.

JATLite featured modular construction consisting of increasingly specialized layers: protocol, Router, KOMC, Base and Abstract layer. Developers could select the appropriate layer to start building their systems. Each layer could be exchanged with other technologies without affecting the operation of the rest of the package.

4.6. FIPA-OS

FIPA-OS [117] is a component-based toolkit enabling rapid development of FIPA compliant agents. It was first released in August 1999 supporting the majority of the FIPA specifications. It has been continuously improved until 2003 and was publicly available as an ideal choice for FIPA compliant agent development. There have been two versions of FIPA-OS:

– **Standard FIPA-OS** - Two alternative distributions were provided: Java 2 (JDK1.2) compatible version (containing code developed directly from the FIPA-OS codebase) and Java 1.1 compatible version (containing code, which has undergone automated code-refactoring to enable the JDK1.2 compatible code of FIPA-OS to be used with JDK1.1).
– **MicroFIPA-OS** - This is an extension to the JDK 1.1 version of FIPA-OS and has been designed to execute on PDAs (that can execute a PersonalJava compatible virtual machine).

Both FIPA-OS versions use tasks and conversations as the basis for support to agents’ functionalities. Developers using FIPA-OS have been encouraged to provide extensions, bug fixes and feedback to help improve different releases.

4.7. MADKIT

Multi-agent development kit - MadKit [122] [56] is an open source modular and scalable multi-agent platform which has been developed at LIRMM (France), built upon the AGR (Agent/Group/Role) organizational model (Aalaadin [42]). MadKit is written in Java and MadKit agents play roles in groups and thus create artificial societies. In addition to AGR concepts, the platform adds three design principles: Micro-kernel architecture; Agentification of services; Graphic component model.

The last version was released in November 2010. MadKit is a set of packages of Java classes that implements the agent kernel, various libraries of messages, probes and agents. This platform is not a classical agent platform as any service, besides those assured by micro-kernel, is handled by agents. Micro-kernel and existence of a range of modular services managed by agents enable a range of multiple and scalable platforms. Communication is achieved through asynchronous message passing: 1) by primitives used to send a message directly to another agent represented by its *AgentAddress*, or 2) by higher-level functions that send or broadcast to one or all agents having a given role in a specific group. MadKit uses agents to achieve distributed message passing, migration control, dynamic security, and other aspect of system management.

MadKit has been used in various projects covering a wide range of applications [67], from simulation of hybrid architectures for control of submarine robots to evaluation of social networks or study of multi-agent control in a production line.

4.8. JAFMAS

Java-based Agent Framework for Multi-Agent Systems - JAFMAS [36], is a framework for representing and developing cooperation knowledge and protocols in a multi-agent system (coordinating their knowledge, plans, and goals so that they can take actions which result in coherent joint problem solution). This framework provides a generic methodology for developing speech-act based multi-agent systems and follows several stages: agent identification, definition of each agent's conversations, determining the rules governing each agent's conversations, analyzing the coherency between all the conversations in the system, and implementation. JAFMAS provides communication (directed and subject-based broadcast), linguistics for speech-acts (e.g. KQML) and coordination support. Such functionality is based on COOL (coordination Lisp-based language for explicitly representing, applying and capturing cooperation knowledge for multi-agent systems). In COOL and JAFMAS, an agent is a programmable entity that can exchange messages within structured "conversations" with other agents, change state and perform actions. JAFMAS agents support conversation based on message exchange according to mutually agreed conventions, change state and perform local actions. Different researchers still use JAFMAS framework for developing multi-agent systems [108] [99].

4.9. Agent Building Shell

Agent Building Shell - ABS was developed at University of Toronto [10]. ABS provides several reusable layers of languages and services for building agent systems. The layers of the architecture achieve a range of functionalities [112]. The shell supports KQML/KIF based communication. COOL is provided and built on top of the agent communication language. The language supports definition, execution and validation of complex speech-act based cooperation protocols. Multiple, parallel conversations are possible and their management can be programmed through specific control mechanisms. Interaction between users (using web browsers) and agents is conversation based, using the same conversational infrastructure that supports interactions among agents. Agents negotiate by exchanging constraints about the performance of activities. In the negotiation process, agents send their requests to other agents and receive either confirmations or explanations why their requests cannot be satisfied. Agents employ a unified behavior description language that specifies behaviors as consisting of sequential, parallel and choice compositions of actions. Specific constraint propagation mechanisms are used to determine which actions will be executed. At the organization level, agents acquire authority to make requests and impose violation costs from the roles they play in the organization. Concerning knowledge management, there is a representational substrate that provides services for carrying out the various reasoning tasks outlined.

According to several authors [51], [11], [71], ABS has been considered appropriate for developing agents in supply chain management systems.

4.10. OAA

Open Agent Architecture – OAA [124] was developed in Artificial Intelligence Center, California and its last version was released in 2007. It is a framework for integrating a community of heterogeneous software agents in a distributed environment. OAA facilitates flexible, adaptable interactions among distributed components through delegation of tasks, data requests and triggers; and enables natural, mobile, multimodal user interfaces to distributed services. OAA is structured to minimize the effort in creating agents and "wrapping" legacy applications, written in various languages and platforms; to encourage the reuse of existing agents; and to allow for dynamism and flexibility in the makeup of agent communities. Unique features of OAA include great flexibility in using facilitator-based delegation of complex goals, triggers, and data management requests; agent-based provision of multimodal user interfaces; and built-in support for including the user as a privileged member of the agent community. The system has been used in different applications and some of them are:

– framework of transformer condition assessment system employing data warehouse, data mining, and Open Agent Architecture [68].
– multi-agent architecture with distributed coordination for an autonomous robot [5].

4.11. Cougaar

Cognitive Agent Architecture – Cougaar [115] is an open-source Java-based agent platform developed as result of a multi-year project of DARPA research. Cougaar is not FIPA-compliant, and more important, it was not designed for standards compliance. Cougaar agents are composed of plugins that communicate sharing common and distributed data space - *blackboard architecture*. Agents can subscribe for automatically receiving blackboard updates. The plugins communicate by publishing (adding) new objects to the blackboard, making changes to objects already published or removing objects from the blackboard. When special objects called *relays* are published onto the blackboard they are automatically forwarded by the blackboard system to other agents, thus achieving the communication between agents.

The main focus of its development was scalability [57] and as a consequence it was mostly utilized for the development of applications in military logistics [31].

4.12. AgentScape

AgentScape was developed at Delft University of Technology as a middleware platform that provides a minimal set of concepts and functionalities for the development of large-scale distributed multi-agent systems. The focus in AgentScape was set on: (i) scalability; (ii) heterogeneity through multiple code bases, programming languages and operating systems; (iii) interoperability [114]. Although AgentScape is a very interesting platform, it currently suffers from the problem that the documentation is not mature enough and is rather incomplete. Nevertheless, AgentScape has been applied in a number of interesting research and commercial projects related to the electricity market [17] and e-commerce [40].

4.13. Cybele

Cybele^{TM} is a commercial agent platform provided by Intelligent Automation Inc. for the development and deployment of large-scale distributed intelligent systems [131]. Cybele^{TM} is built on top of Java platform. Agents are programmed in Java using a standard style of programming called Activity

Centric Programming (ACP). This means that the basic building blocks of an agent are *activities*, while accesses to the basic functionalities of Cybele^{TM} are provided via an Activity Oriented Programming Interface (AOPI). Cybele^{TM} allows the development of distributed applications by installing it on several (at least 2) network nodes that together define a Cybele^{TM} community. Exactly one node is designated as a *master node*, while the rest of them are *slave nodes*. A Cybele^{TM} node can host several specialized Java applications known as Cybele^{TM} *containers*. A container provides the runtime environment for a set of Cybele^{TM} agents. It is not difficult to observe that an activity in Cybele^{TM} has similarities with behavior in JADE, as well as with a plugin in Cougaar. Moreover, the method of structuring a distributed agent application into nodes, containers, agents and activities / plugins / behaviors is also used by JADE and Cougaar.

Cybele^{TM} can be utilized as a platform for distributed robotics. The Distributed Control Framework (DCF) is a framework for building robotics applications for robot team coordination and management. Cybele^{TM} is used as a core for DCF which supports two types of robotic agents: (i) *Robot Agent* that embodies a real or a simulated robot; (ii) *Remote Control Agent* that provides the control interface for a human operator with a robot team. Additionally, DCF includes a suite of components for sensing, estimation and control of several commercial robotic platforms.

5. Conclusion

Software agents are an emergent and rapidly developing field of research. In the last decade, a number of essential advances have been made in the design and development of software agent languages and the implementation of multi-agent systems. In this brief survey, we have tried to bring some of the key concepts, languages, tools and platforms and make a reference point to a large body of literature. Our intention was to enumerate and present essential features and functionalities of selected languages, tools and platforms, instead of judging them.

We consider an orthogonal classification by looking at the way agent programming languages are used during the systems development process. On one side, we can find agent languages useful for building software agents that can be used as building blocks for the development and deployment of complex distributed applications, usually based on agent or other suitable middleware platforms. On the other hand, we can find agent programming languages used for designing and running complex simulation models that employ the agent metaphor for modeling and simulation of complex systems. However, these languages are not immediately useful for developing real systems, but are rather mostly employed for research in understanding complex systems using agent-based modeling and simulation tools, as agent simulation languages. Note that this class of languages is very often forgotten by the existing works that overview advances in agent programming.

Nevertheless, between the two extremes we can find agent languages that are useful for both systems simulation, as well as for systems development and deployment.

In Table 3 we give a brief summary of agent programming languages. It can be noted that almost all of them, particularly the recently developed ones, have appropriate web-sites and IDEs. Despite the fact that there are representatives of different programming paradigms (imperative, declarative, BDI, hybrid), almost all of them are implemented in Java and a significant number of them are implemented in Prolog. Most of the recently developed languages find their place in real environments and have been used in developing different kinds of applications. Unfortunately, for majority of them there are no appropriate textbooks.

Note that we were able to find in the literature other overview works that provide classifications and comparisons of agent programming languages. Authors of [21] propose a classification of agent programming languages based on a lightweight interpretation of the programming paradigm as imperative, declarative, and hybrid (i.e. between declarative and imperative).

For the development and deployment of a multi-agent system in real environments it is necessary that appropriate software infrastructures (frameworks, tools, platforms) exist.

According to our literature survey, more than 100 agent infrastructures have been developed in the previous two decades. For portability and usability reasons most of them are built on top of and are integrated with Java [102]. Unfortunately, only few of them are still currently available, others either becoming obsolete or not being developed anymore.

Futhermore, this prominent technology inspired some authors to go a step further. In [70] authors extrapolated future trends in multi-agent systems and presented a thorough and outstanding approach to the future of multi-agent systems. Finally, it is important to mention that in order to be accepted by the industrial community, MAS applications need to be successfully demonstrated in complex real world pilot systems [29].

Acknowledgment. Research was partially supported by the Ministry of Education and Science, Republic of Serbia, through project no. OI-174023 'Intelligent techniques and their integration into wide-spectrum decision support'.

References

1. Agents mailing list, agents@cs.umbc.edu.
2. Ancona, D., Mascardi, V., Hübner, J.F., Bordini, R.H.: Coo-AgentSpeak: Cooperation in AgentSpeak through Plan Exchange, In Third International Joint Conference on Autonomous Agents and Multiagent Systems, Vol. 2, pp. 696 – 705 (2004)
3. Ancona, D., Mascardi, V.: Coo-BDI: Extending the BDI Model with Cooperativity, In Declarative Agent Languages and Technologies, Vol. 2990, pp.109-134 (2004)
4. Azarmi, N., Thompson, S.: "ZEUS: A Toolkit for Building Multi-Agent Systems", Proceedings of fifth annual Embracing Complexity Conference, Paris, (2000)

5. Badano B.M.I., A multi-agent architecture with distributed coordination for an autonomous robot, PhD theses, University of Girona, (October 2008)
6. Badjonski, M., Ivanović, M.: "Multi-agent System for Determination of Optimal Hybrid for Seeding", Proceedings of EFITA '97 - First European Conference for Information Technology in Agriculture, Copenhagen, Denmark, June 15-18, pp. 401-404. (1997)
7. Badjonski, M., Ivanović, M., Budimac, Z.: "Possibility of using Multi-Agent System in Education", Proceedings of IEEE International Conference on Systems, Man, and Cybernetics, Orlando, Florida, USA, October 12-15, pp. 588-593. (1997)
8. Badjonski, M., Ivanović, M., Budimac, Z.: "Software Specification Using LASS", Proceedings of Asian'97, Lecture Notes in Computer Science Vol 1345, Springer-Verlag, Kathmandu, Nepal, pp. 375-376. (1997)
9. Badjonski, M.: Adaptable Java Agents – a Tool for Programming of Multi-Agent Systems, PhD thesis, Department of Mathematics and Informatics, Faculty of Natural Science, University of Novi Sad (2003)
10. Barbuceanu, M., Fox, M.S., The architecture of an agent building shell. Intelligent Agents II, LNAI 1037, Spinger-Verlag, pp. 235-250 (1996)
11. Fox M.S., Barbuceanu M., Teigen R.: Agent-Oriented Supply Chain Management, International Journal of Flexible Manufacturing System, vol 12, pp. 165-188. (2000)
12. Bădică, C., Manufacturing and Control: Putting Agents to Work, IEEE Distributed Systems Online, vol. 8, no. 6, pp. 5, (2007)
13. Bădică, C., Ganzha, M., Paprzycki, M.: Developing a Model Agent-based E-Commerce System. In: Jie Lu, Guangquan Zhang, and Da Ruan (eds.): E-service Intelligence, Studies in Computational Intelligence, Volume 37, Springer, 555-578 (2007)
14. Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R.: METATEM: A Framework for Programming in Temporal Logic, In: Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness, REX workshop, LNCS Volume 430, pp. 94-129 (1990)
15. Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R.: METATEM: An introduction. Formal Aspects of Computing 7(5), pp. 533–549 (1995)
16. Bratman, M.E.: Intention, Plans and Practical Reason. Harvard University Press, 1987.
17. Brazier, F., Cornelissen, F., Gustavsson, R., Jonker, C.M., Lindeberg, O., Polak, B., Treur, J.: A Multi-Agent System Performing One-to-Many Negotiation for Load Balancing of Electricity Use. In: Electronic Commerce Research and Applications Journal, vol.1, no.2, pp. 208-224, Elsevier, (2002)
18. Bellifemine, F., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE, John Wiley & Sons (2007)
19. Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: JADE: A software framework for developing multi-agent applications. Lessons learned, Information and Software Technology, Volume 50, Issues 1-2, Elsevier, pp. 10-21. (2008)
20. Bordini, R.H., Dix, J., Dastani, M., Seghrouchni, A.E.F.: Multi-Agent Programming Languages, Platforms and Applications, Springer, (2005)
21. Bordini, R.H., Braubach, L., Dastani, M., Seghrouchni, A.E.F., Gomez-Sanz, J.J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A Survey of Programming Languages and Platforms for Multi-Agent Systems, Informatica, no.30, pp. 33-44 (2006)
22. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-Agent Systems in AgentSpeak using Jason, John Wiley & Sons, (2007)
23. Bordini, R.H., Dastani, M., Dix, J., Seghrouchni, A.E.F. (Eds.): Multi-Agent Programming: Languages, Tools and Applications, Springer (2009)

24. Brooks, R.A.: "A Robust Layered Control System for a Mobile Robot", IEEE Journal of Robotics and Automation, 2(1), pp. 14-23. (1986)
25. Brooks, R.A.: "Intelligence without Reason", Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia, pp. 569-595. (1991)
26. Brooks, R.A.: "Intelligence without Representation", Artificial Intelligence, 47, pp 139-159. (1991)
27. Budimac, Z., Ivanović, M., Popović, A.: "Workflow Management System Using Mobile Agents", Proceedings of ADBIS '99, Lecture Notes in Computer Science, Maribor, Slovenia, pp. 169-178. (1999)
28. Bussink, D.: A Comparison of Language Evolution and Communication Protocols in Multi-agent Systems. 1st Twente Student Conference on IT, Track C - Intelligent_Interaction, http://referaat.ewi.utwente.nl/ (2004)
29. Camarinha-Matos, L. M.: Multi-agent systems in virtual enterprises. Proceedings of AIS'2002 – International Conference on AI, Simulation and Planning in High Autonomy Systems, SCS publication, Lisbon, Portugal, pp. 27-36. (2002)
30. Cardelli, L., Gordon, A.D.: Mobile ambients. Foundations of Software Science and Computational Structures, Lecture Notes in Artificial Intelligence 1378, Springer, pp. 140-155. (1998)
31. Carrico, T., Greaves, M.: Agent Applications in Defense Logistics. In: Defence Industry Applications of Autonomous Agents and Multi-Agent Systems, Whitestein Series in Software Agent Technologies and Autonomic Computing, Birkhäuser Basel, pp. 51-72. (2008)
32. Clark, K. L., McCabe, F. G.: Go! – A Multi-paradigm Programming Language for Implementing Multi-threaded Agents, In Annals of Mathematics and Artificial Intelligence, Vol. 41, Issue 2 – 4, pp. 171 – 206, (2004)
33. Collier, R.W.: Agent Factory: A Framework for the Engineering of Agent-Oriented Applications, Doctoral Thesis, University College Dublin, Ireland, (2001)
34. Dastani, M.: 2APL: a practical agent programming language, International Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS), 16(3), pp. 214-248 (2008)
35. 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, ACM Press, (1994)
36. Chauhan, D., Baker, A.D.: JAFMAS: a multiagent application development system. In Proceedings of the second international conference on Autonomous agents (AGENTS '98), http://doi.acm.org/10.1145/280765.280782 (1998)
37. De Giacomo, G., Lespérance, Y., Levesque, H.J., Sardina, S.: IndiGolog: A High-Level Programming Language for Embedded Reasoning Agents, In [23], Springer, pp. 31-72. (2009)
38. Dennett, D.: Intentional Systems. In: Journal of Philosophy No. 68, pp. 87–106. (1971)
39. Dylla, F., Ferrein, A., Lakemeyer, G.: Specifying multirobot coordination in ICPGolog - from simulation towards real robots. In Proc. of the Workshop on Issues in Designing Physical Agents for Dynamic Real-Time Environments: World modeling, planning, learning, and communicating (IJCAI 03), (2003)
40. El-Akehal, E.E., Padget, J.: Pan-supplier stock control in a virtual warehouse. In: Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems AAMAS '08, pp. 11-18 (2008)
41. Seghrouchni, A.E.F., Suna, A.: CLAIM and SyMPA: A Programming Environment for Intelligent and Mobile Agents. In: [20], pp. 95-122, Springer, (2005)

42. Ferber, J., O. Gutknecht: Aalaadin: a meta-model for the analysis and design of organizations in multi-agent systems, In Proceedings of the Third International Conference on Multi-Agent Systems, ICMAS'98 pp. 128-135. (1998)
43. Ferrein, A., Lakemeyer, G.: Logic-based robot control in highly dynamic domains, Robotics and Autonomous Systems, volume 56, issue 11, North-Holland Publishing Co., 980-991, (2008)
44. Ferrein, A.: golog.lua: Towards a Non-Prolog Implementation of Golog for Embedded Systems. In: Cognitive Robotics, Dagstuhl Seminar Proceedings, no.10081, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Also in Proceedings of AAAI Spring Symposium 2010 on Embedded Reasoning, Stanford University, (2010)
45. Finin, T., Weber, J., et. al.: "Draft Specification of the KQML Agent-Communication Language", The Darpa Knowledge Sharing Initiative External Interfaces Working Group, available as http://www.cs.umbc.edu/kqml/kqmlspec.ps. (1993)
46. Finzi, A., Lukasiewicz, T.: Game-Theoretic Agent Programming in Golog, In: Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'2004, pp. 23-27, IOS Press, (2004)
47. Foundation for Intelligent Physical Agents, "FIPA ACL Message Structure Specification", available at http://www.fipa.org/specs/fipa00061/
48. Fisher M.: "Representing and Executing Agent-Based Systems", Intelligent Agents, Lecture Notes in Artificial Intelligence, Vol. 890, Springer-Verlag, pp. 307-323. (1994)
49. Fisher, M.: A Survey of Concurrent MetateM – The Language and its Applications, In Proceedings of the First International Conference on Temporal Logic, LNCS, Vol. 827, pp. 480 – 505, (1994)
50. Fisher, M., Hepple, A.: Executing Logical Agent Specifications. In [23], pp. 3-29, (2009)
51. Forget P., D'Amours S., Frayret J.M.: Multi-Behavior Agent Model for Planning in Supply Chains: An Application to the Lumber Industry, Universite Laval, Quebec, Canada, Working paper DT-2006-SD-03, https://www.cirrelt.ca/DocumentsTravail/2006/DT-2006-SD-03.pdf (2006)
52. Franklin, S., Graesser, A.: "Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents", Working Notes of the Third International Workshop on Agent Theories, Architectures and Languages, ECAI '96, Budapest, Hungary, pp. 193-206. (1996)
53. Fuggeta, A., Picco, G.P.: Understanding Code Mobility, IEEE Transactions on Software Engineering, vol.24, no.5, pp.342-361, (1998)
54. Georgakarakou, C. E., Economides, A. A.: Software agent technology: An overview. In: Software Applications: Concepts, Methodologies, Tools, and Applications, P. F. Tiako (ed.), IGI-Global ISBN: 978-1-60566-060-8 (2007)
55. Georgeff, M., Lansky, A.: Reactive reasoning and planning. In: Proceedings of the 6th National Conference on Artificial Intelligence (AAAI-87), pp. 677-682, (1987)
56. Gutknecht, O., Ferber, J.: The MADKIT Agent Platform Architecture. Agents Workshop on Infrastructure for Multi-Agent Systems, (2000)
57. Helsinger, A., Thome, M., Wright, T.: Cougaar: A Scalable, Distributed Multi-Agent Architecture, Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, 1910 – 1917, vol.2, IEEE Computer Society Press, (2004)
58. Hewitt, C.: The Challenge of Open Systems. Byte Magazine 10, 4, pp. 223-242, (April 1985)

59. Hindriks, K.V., de Boer, F.S., van der Hoek, W., Meyer, J-J.C.: A formal embedding of AgentSpeak(L) in 3APL, Advanced Topics in Artificial Intelligence, LNCS 1502, pp. 155-166, (1998)

60. Hindriks, K.V., De Boer, F.S., van der Hoek, W., Meyer, J-J.C.: Agent Programming in 3APL, Autonomous Agents and Multi-Agent Systems Volume 2, Number 4, pp. 357-401, (1999)
61. Hindriks, K.V., De Boer, F.S., van der Hoek, W., Meyer, J-J.C.: "Agent Programming with Declarative Goals", Intelligent Agents VII. Agent Theories Architectures and Languages, LNCS 1986, pp. 248-257, Springer, (2001)
62. Hindriks, K.V.: Programming Rational Agents in GOAL. In [23], 119-157, Springer, (2009)
63. Huang, J., Jennings, N., Fox, J.: "An Agent Architecture for Distributed Medical Care", Intelligent Agents, Lecture Notes in Artificial Intelligence, Vol 890, Springer-Verlag, pp. 219-232. (1994)
64. Jennings, N.R., Wooldridge, M.: "Software Agents", IEE Review, January, pp. 17-20. (1996)
65. Jeon, H., Petrie, C., Cutkosky, M.R.: JATLite: A Java Agent Infrastructure with Message Routing, IEEE Internet Computing, pp. 2-11. (2000)
66. Kaelbling, L.P.: "A Situated Automata Approach to the Design of Embedded Agents", SIGART Bulletin, 2(4), pp. 85-88, (1991)
67. Kallel I., Chatty A., Allimi A.M.: Self-Organizing Multirobot Exploration through Counter-Ant Algorithm, Proceedings Self-Organizing Systems: Third International Workshop, Iwsos 2008, Vienna, Austria, December 10-12, 2008, Springer. (2008)
68. Wu, L., Yongli, Z., Yuan, J., Li, X.: "Application of Open Agent Architecture and Data Mining Techniques to Transformer Condition Assessment System," International Journal of Emerging Electric Power Systems: Vol. 2 : Iss. 1, Article 1034. (2005)
69. Luck, M., Ashri, R., d'Inverno, M.: Agent-Based Software Development, Artech House, 2004
70. Luck, M., McBurney, P., Gonzalez-Palacios, J: Agent-Based Computing and Programming of Agent Systems. LNCS, Agent-Based Computing and Programming of Agent Systems, 3862, pp. 23-37, Springer. (2006)
71. Madejski, J.: Survey of the agent-based approach to intelligent manufacturing, Journal of of Achievements in Materials and Manufacturing Engineering, VOLUME 21, ISSUE 1, pp. 67-70. (2007)
72. Maes, P.: "The Agent Network Architecture (ANA)", SIGART Bulletin, 2(4), pp. 115-120, (1991)
73. Maes, P.: "Modelling Adaptive Autonomous Agents", Artificial Life Journal, Ed. C. Langton, Vol 1, No. 1&2, MIT Press, pp. 135-162. (1994)
74. McCabe, F. G., Clark, K. L.: April – Agent PRocess Interaction Language, In Proceedings of the workshop on agent theories, architectures, and languages on Intelligent agents, pp. 324 – 340 (1995)
75. McCarthy, J.: Situations, actions and causal laws. Technical report, Stanford University, 1963. Reprinted in Semantic Information Processing (M. Minsky ed.), MIT Press, Cambridge, Mass., pp. 410-417 (1968)
76. Muldoon, C., O'Hare, G.M.P., Collier, R.W., O'Grady, M.J.: Towards Pervasive Intelligence: Reflections on the Evolution of the Agent Factory Framework. In: [23], pp. 147-212 (2009)
77. Minsky, M.: "The Society of Mind", Simon and Schuster, New York (1986)

78. Nguyen G., Dang T.T, Hluchy L., Laclavik M., Balogh Z., Budinska I.: AGENT PLATFORM EVALUATION AND COMPARISON, Institute of Informatics, Slovak Akademy of Sciences (2002)
79. Nwana, H., & Wooldridge, M., Software Agent Technologies. BT Technology Journal 14(4), pp. 68-78. (1996)
80. Nwana, H.S.: "ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems", Proceedings of PAAM'98, London pp. 377-392. (1998)
81. Nwana, H.S., Ndumu, D.T., Lee, L.C., Collis, J.C.: " A Toolkit and Approach for Building Distributed Multi-Agent Systems ", Proceedings of the Third International Conference on Autonomous Agents (Agents'99), Seattle, WA, USA, pp. 360-361. (1999)
82. Pokahr, A., Brauhach, L., Lamersdorf, W.: Jadex: A BDI Reasoning Engine. In [20], pp. 149-174, Springer (2005)
83. Rao, A.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world, Einhoven, Netherlands, pp. 42-55 (1996)
84. Reiter, R.: Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems, MIT Press (2001)
85. Ross, R., Collier, R., O'Hare, G.: AF-APL: Bridging principles and practices in agent oriented languages. In Programming Multi-Agent Systems, Second Int. Workshop (ProMAS'04), volume 3346 of LNCS, Springer Verlag, pp. 66–88, (2005)
86. Russell, S., and Norvig, P.: Artifical Intelligence: A Modern Approach. Prentice-Hall, 2nd edition, 2002.
87. Santoro, C.: Towards an Agent Programming Language, In 10th national workshop Towards the Future of Agent-based software systems, Parma, Italy (2009)
88. Sardina, S., Lespérance, Y.: Golog Speaks the BDI Language. In: 7th International Workshop on Programming Multi-Agent Systems, ProMAS 2009, LNCS 5919, pp. 82-99, Springer (2010)
89. Schiffel, S., Thielscher, M., Trang, D.T.: An Agent Team Based on FLUX for the ProMAS Contest 2007, Proceedings of the 5th international conference on Programming multi-agent systems, ProMAS'07, LNCS 4908, pp. 261-265, Springer-Verlag (2008)
90. Shoham, Y.: "Agent-Oriented Programming", Artificial Intelligence, 60(1), pp. 51-92 (1993)
91. Suna, A., Seghrouchni, A.E.F.: Programming mobile intelligent agents: An operational semantics, Web Intelligence and Agent Systems, vol.5, no.1, pp. 47-67, IOS Press (2007)
92. Sycara, K.P., Paolucci, M., Velsen, M.V., Giampapa, J.A.: The RETSINA MAS Infrastructure, Autonomous Agents and Multi-Agent Systems, Springer, no.1-2, pp. 29-48 (2003)
93. Thielscher, M.: Introduction to the fluent calculus. Electronic Transactions on Artificial Intelligence, 2(3–4), pp. 179–192 (1998)
94. Thielscher, M.: Reasoning Robots. The Art and Science of Programming Robotic Agents, Applied Logic Series, Vol.33, Springer (2005)
95. Thomas, R.S.: "PLACA, an Agent Oriented Programming Language", PhD thesis, Computer Science Department, Stanford University, Stanford, CA 94305, (1993)
96. Unland, R., Klusch, M., Calisti, M.: Software Agent-Based Applications, Platforms and Development Kits, Birkhauser Verlag AG (2005)

97. Schip, R.C.v.h., Warnier, M., Brazier, F.M.: Deploying BDI agents in open, insecure environments, in: Proceedings of the 7th European Workshop on Multi-Agent Systems (EUMAS'09) (2009)
98. Wagner, G.: VIVA knowledge-based agent programming. Preprint, Institut fur Informatik, Universitat Leipzig, Germany, (1996)

99. Wang, J.B., Pang, J., Jiang, B.C.: The Modeling and Implementation of Virtual Enterprise Based on Multi-Agent System, Applied Mechanics and Materials (Volume 33) pp. 280-284. (2010)
100. Weerasooriya, D., Rao, A. S., Ramamohanarao, K.: Design of a Concurrent Agent-Oriented Language, LNAI, Vol 890, Springer-Verlag, (1994)
101. Weiss, G. (ed.): Multiagent Systems. A Modern Approach to Distributed Artificial Intelligence. MIT Press 2000.
102. Weiss, G.: Agent orientation in software engineering. Knowledge Engineering Review, 16(4), pp. 349–373. (2002)
103. Winikoff, M.: JACKTM Intelligent Agents: An Industrial Strength Platform, In: [20], Springer, pp. 175-193, (2005)
104. Wooldridge, M., Jennings, N.R.: "Agent Theories, Architectures, and Languages: A Survey", Intelligent Agents, LNAI, Vol 890, Springer- Verlag, pp. 1-39. (1994)
105. Wooldridge, M., Jennings, N.R.: "Intelligent Agents: Theory and Practice", available as http://www.doc.mmu.ac.uk:80/STAFF/mike/ker95/ker95-html.html, (1994)
106. Wooldridge, M.: A Knowledge-Theoretic Semantics for Concurrent MetateM, In Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages, LNCS, Vol. 1193, pp. 357 – 374, (1996)
107. Wooldridge, M.: Intelligent Agents, chapter 1 of Multiagent systems: a modern approach to distributed artificial intelligence, Massachusetts Institute of Technology, 2000, pp. 27 – 77 (2000)
108. Jinkai, X., Weihong, Y.: Study of comparison between JAFMA and JADE, Circuits, Communications and System (PACCS), 2010 Second Pacific-Asia Conference on, pp. 105 – 108 (2010)
109. Brooks, R.A.: Elephants don't play chess, In Robotics and Autonomous Systems, vol 6., pp 3-15, (1990)
110. Lee, E.A.: Cyber Physical Systems: Design Challenges, EECS Department, University of California, (2008)
111. http://www.cs.uu.nl/3apl/, accessed in January 2011
112. http://www.eil.utoronto.ca/aac/abs/, accessed in January 2011
113. http://www.agentfactory.com/index.php/AF-AgentSpeak, accessed in Jan. 2011
114. http://www.agentscape.org/, accessed in January 2011
115. http://www.cougaar.org/, accessed in January 2011
116. http://fipa.org/, accessed April 2011
117. FIPA-OS (http://fipa-os.sourceforge.net/index.htm) accessed in January 2011
118. http://www.cs.toronto.edu/cogrobo/main/systems/index.html, accessed in January 2011
119. http://jade.tilab.com/, accessed in January 2011
120. http://jadex-agents.informatik.uni-hamburg.de/, accessed in January 2011
121. http://jason.sourceforge.net, accessed in January 2011
122. http://www.madkit.org/, accessed in January 2011
123. http://www.csc.liv.ac.uk/~anthony/metatem.html, accessed in January 2011
124. [http://www.ai.sri.com/~oaa/], accessed in January 2011
125. http://www-2.cs.cmu.edu/~softagents/, accessed in January 2011

126. http://www.ai.mit.edu/people/sodabot/slideshow/total/p001.html, accessed in January 2011
127. http://activist.gpl.ibm.com:81/WhitePaper/ptc2.htm, accessed in January 2011
128. http://agents.media.mit.edu/, accessed in January 2011
129. http://193.113.209.147/projects/agents/zeus/index.htm, accessed in January 2011
130. http://www.aosgrp.com, accessed in January 2011
131. http://www.i-a-i.com/, accessed in January 2011
132. http://mmi.tudelft.nl/~koen/goal.php, accessed in January 2011
133. http://apapl.sourceforge.net/, accessed in January 2011

Costin Bădică received in 2006 the title of Professor of Computer Science from University of Craiova, Romania. He is currently with the Department of Software Engineering, Faculty of Automatics, Computers and Electronics of the University of Craiova, Romania. His research interests are at the intersection of Artificial Intelligence, Distributed Systems and Software Engineering. He authored and coauthored more than 100 publications related to these topics as journal articles, book chapters and conference papers. He prepared special journal issues and coedited 4 books in Springer's Studies in Computational Intelligence series. He coinitiated the Intelligent Distributed Computing -- IDC series of international conferences that is being held yearly. He is member of the editorial board of 4 international journals. He also served as programme committee member of many international conferences.

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.

Hans-Dieter Burkhard is Senior professor at the Institute of Informatics at Humboldt University of Berlin. He founded the Artificial Intelligence group at Humboldt University. He has studied Mathematics in Jena and Berlin, and he has worked on Automata Theory, Petri Nets, Distributed Systems, VLSI Diagnosis and Knowledge Based Systems. Current interests include Cognitive Robotics, Distributed AI, Agent Oriented Techniques, Machine Learning, Socionics, and AI applications in Medicine. He is a fellow of the ECCAI, and he was Vice President of the International RoboCup Federation. His publication activities include numerous papers and book articles, invited talks and memberships in program committees. His soccer robot teams have won several first places in the RoboCup world championships.

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 Science. 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.

Received: February 14, 2011; Accepted: April 12, 2011.

Table 3. Summary of agent languages

Name	**Web page**	**IDE**	**Implementation language**	**Agent platform integration**	**Applications**	**Paradigm**	**Text book**
AGENT0	No	No	Interpreters written in Prolog and CommonLisp	N/A	N/A	Declarative	No
PLACA	No	No	None (experimental)	N/A	N/A	Declarative, prototype	No
Agent-K	No	No	Prolog	N/A	N/A	Declarative	No
MetateM	No	No	Interpreters written in Prolog and Scheme	N/A	According to [Fisher 1994], can be used in process control, fault-tolerance, bidding, etc.	Declarative, based on discrete, linear temporal logic	No
APRIL	http://sourceforge.net/projects/networkagent/	Yes	C, Java, Prolog	No, although its execution relies on external software (e.g. April Machine, InterAgent Communication server)	Networked intelligent agents (kaccording to [McCabe 1994])	Process-oriented symbolic language, not designed specifically for multi-agent programming	No
MAIL	No	No	APRIL	N/A	N/A (development of the language was discontinued)	Hybrid	No
VIVA	No	No	PVM-Prolog	N/A	N/A	Declarative, combining concepts of Prolog and SQL	No
GO!	http://sourceforge.net/projects/networkagent/	Yes	C, Java, Prolog	N/A	Networked intelligent agents	Hybrid (according to [Bordini 2006])	No
Agent Speak	No	Yes, (indirectly (e.g. for Jason)	Several interpreters for the language exist, such as Jason, SIM_Talk, and AgentTalk	N/A	N/A	Declarative, theoretical language	Yes [22]
Jason	http://jason.sf.net	Yes	Java interpreter for AgentSpeak(L)	Yes, based on JADE and Saci; was also integrated in AgentScape [97] and Agent Factory [113]	N/A	Hybrid (according to [21])	Yes [22]
AF-APL	http://www.agentfactory.com/index.php/Main_Page	Yes, via Agent Factory	Java	Agent Factory	Robotics, virtual and mixed reality environments, and mobile computing [Collier, 2009]	According to [21] is hybrid	No

Name	Web page	IDE	Implementation language	Agent platform integration	Applications	Paradigm	Text book
3APL	http://www.cs.uu.nl/3apl/	Yes	A Java implementation and a Haskell implementation	N/A	Robot control using an API called ARIA (provided by http://www.activmedia.com), look at http://www.cs.uu.nl/3apl/thesis/verbeek/verbeekimpl.html	According to classification of [21] is hybrid	No
2APL	http://apapl.sourceforge.net/	Yes	Java on top of JADE	JADE	N/A	Hybrid	No. There is a tutorial
JACK	http://aosgrp.com/products/jack/index.html	Yes	Java	No, execution relies on the JACK agent kernel runtime	Unmanned Aerial Vehicles, surveillance, air traffic management	Imperative	A number of manuals and tutorial.
JADEX	http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview	Yes	Java	JADE	Workflow execution, self-organizing systems, treatment scheduling for patients in hospitals	Hybrid	A number of user guides and tutorials
GOAL	http://mmi.tudelft.nl/trac/goal	Yes	Java based on SWI-Prolog	According to [62], GOAL has been tested on top of JADE. However, we could not find any reference to such an experiment	It is just a prototype that is currently used for educational purposes. It can be useful in planning applications, e.g. in the transportation domain	Declarative	No. There is a tutorial on its Web site
Golog	http://www.cs.toronto.edu/cogrobo/main/	No	Prolog (Eclipse Prolog, SWI-Prolog)	N/A	Cognitive robotics, embedded systems	Declarative	Yes [84]
FLUX	http://www.fluxagent.org/home.htm	No	Two implementations available: 1. Eclipse Prolog (constraint logic programming system), and 2. Sicstus Prolog	N/A	Cognitive robotics	Declarative (also according to [21])	Yes [Thielscher, 2005a]
CLAIM	?	?	Java	SyMPA	N/A	Although in [41] it is said that CLAIM is declarative our impression is that it is hybrid	No

DOI:10.2298/CSIS101230011L

SPEM Ontology as the Semantic Notation for Method and Process Definition in the Context of SWEBOK

Miroslav Líška[1] and Pavol Navrat[1]

[1]Faculty of Informatics and Information Technologies,
Slovak University of Technology,
Ilkovičova 3, 842 16 Bratislava, Slovakia
liska@semantickyweb.sk, navrat@fiit.stuba.sk

Abstract. The Guide to the Software Engineering Body of Knowledge (SWEBOK) provides a consensually validated characterization of the bounds of the software engineering discipline and to provide a topical access to the Body of Knowledge supporting that discipline. The topic "Notation for Process Definition" references selected notations appropriate for software process definition. However all of them have weakly defined semantics, thus is not possible to use formal techniques for process model creation, validation etc. In this work we present created Software and Systems Process Engineering Meta-Model (SPEM) Ontology that improves the lack of mentioned process notations. The SPEM Ontology constitutes a semantic notation that provides concepts for knowledge based software process engineering. The work also discusses utilization of such semantic notation in other selected SWEBOK topics, the Software Project Planning, the Software Project Enactment, and the Verification and Validation.

Keywords: software and systems process engineering meta-model, web ontology language, model driven architecture, semantic web, SPEM, OWL, MDA, SWEBOK.

1. Introduction

There are a number of notations that are used to define software processes [1]. A key difference between them is in the type of information they define, capture, and use. The approaches encompass for example: natural language [2], Data Flow diagrams [3], Statecharts [4], ETVX [5], Actor-Dependency modeling [6], SADT notation and many others [7]. Unfortunately, semantics of the mentioned notations are defined weakly, thus it is not possible to make and to verify created language statements with formal techniques such as the consistency or satisfiability verification. Although standard software development process frameworks provide much useful information, typically in the form of navigable websites, this information contains only human-readable descriptions. Therefore, these kinds of frameworks cannot be used to represent machine interpretable content [8]. Moreover, these process frameworks are used in the technical spaces [9] that have model based architecture, such as MDA or Eclipse Modeling Framework (EMF) [10]. These kinds of technical spaces also limit knowledge based processing, owing to their weakly defined semantics [11]. However, at present the emerging field of Semantic Web technologies promises new stimulus for Software Engineering research [12]. The acquired opportunity to work with semantics opens door for original contributions to many problems in the field, e.g web service composition aided by semantics [51-53]. The Semantic Web is a vision for the future of the Web, in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web [13]. The today's key Semantic Web technology is Web Ontology Language (OWL). OWL is intended to be used when the information contained in documents needs to be processed by applications, as opposed to situations where the content only needs to be presented to humans [14].

Aforementioned problems in software engineering and facts about the Semantic Web implies an opportunity to support software process definition with OWL, and thus to support software process engineering with knowledge based techniques. In this work we address such an opportunity and propose an ontology based software process definition that could empower software process engineering with knowledge engineering techniques. To achieve it we need to move software process engineering to the Semantic Web technical space. We have chosen Software and Systems Engineering Meta-Model and transformed it to the OWL DL representation, so having created SPEM Ontology.

1.1. Related works

SPEM is MDA standard used to define software and systems development processes and their components [15]. A SPEM process can be systematically mapped to a project plan by instantiating the different process' breakdown structure views. Therefore a SPEM model can represent a knowledge base that can be used for verification, whether a project plan conforms to this knowledge. However, the SPEM metamodel has the semiformal architecture, thus it is not possible to make and to verify created SPEM language statements with formal techniques such as the consistency or satisfiability verification [16]. But if we transform SPEM to the Semantic Web technical space, we can use the mentioned formal techniques due to facilities of OWL. Because SPEM is based on MDA, we can utilize the research results of transforming other MDA's standards to the Semantic Web technical space.

SPEM is specified in the Meta Object Facility (MOF) language that is the key language of MDA. MOF is a language for metamodel specification and it is used for specification of all model-based MDA standards [17]. It provides metadata management framework, and a set of metadata services to enable the development and interoperability of model and metadata driven systems

[18]. On the Semantic Web side, OWL is intended to provide a language that can be used to describe the classes and relations between them that are inherent in Web documents and applications. OWL is based on Resource Description Framework Schema (RDFS) [19]. Both MOF and RDFS provide language elements, which can be used for metamodeling. Although they have similar language concepts such as `mof:ModelElement` with `rdf:Resource`, or `mof:Class` with `rdf:Class`, the languages are not equivalent. RDFS, as a schema layer language, has a non-standard and non-fixed-layer metamodeling architecture, which makes some elements in model to have dual roles in the RDFS specification [20]. MOF is also used for specification of the Unified Modeling Language (UML) that is a language for specification, realization and documentation of software systems [21]. Even if UML and RDFS are similar in the domain of system specification, they are also substantially different. One issue that has been addressed was the problem that RDF properties are first class entities and they are not defined relative to a class. Therefore a given property cannot be defined to have a particular range when applied to objects of one class and another range when applied to objects of a different class [22]. This difference has also been propagated between OWL and UML [23]. It should be noted that efforts to transfer explicit knowledge into machine processable form encompass a much wider spectrum of works, e.g. [24, 25]. Still others attempt to develop domain specific languages, incorporating knowledge on the domain, that would be adaptable [26] improving in such a way the process of software evolution [27]. At present the main bridge that connects the Semantic Web with MDA is stated in the Ontology Definition Meta-Model (ODM) [28]. ODM defines the OWL Meta-Model specified in MOF (MOF – OWL mapping) and also the UML Profile for Ontology modeling (UML – OWL mapping). This architecture can be extended with additional mappings between the UML Profile for OWL and other UML Profiles for custom domains [29, 30]. We have already utilized this principle in our previous works where we created an approach to SPEM model validation with ontology [31], an approach to project planning employing software and systems engineering meta-model represented by an ontology [32], and ontology driven approach to software project enactment with a supplier [33]. However, our works are not the only one that concern with using of SPEM in the Semantic Web technical space. In the following paragraph we reference to the three other related works.

The first work proposes to represent SPEM in Description Logic (DL) [34]. The work creates mapping from MOF to DL and mapping from OCL [35] constraints of SPEM to DL. The reason for the former mapping is to represent the SPEM MOF based metamodel with DL and the latter is to represent additional OCL constraints that supplement the SPEM metamodel with additional semantics. The second work presents a competency framework for software process understanding [36]. The motive is to create assessments for a correct understanding of a process that can be used in a software development company. The paper introduces creation of SPEM software process ontology for the Scrum software process [37] with EPF Composer. However, the third work is the closest to our approach, since it proposes project plan verification with ontology. The work intends to use SPEM process constraint definitions with the semantic rules with Semantic Web Rule Language (SWRL) [38], where SWRL is W3C language that combines OWL and RuleML [39].

1.2. Aims and objectives

We aimed in our research to devise a method that uses ontology based software process notation which could be used for ontology based software process engineering. To be more precise, we propose an extension of the SWEBOK topic “Notation for Process Definition” of the Software Engineering Process Knowledge Area with additional process notation SPEM Ontology. Consequently we present utilizations of such semantic notation in the context of SWEBOK. The SPEM Ontology is first applied to the Software Project Planning topic and then to the Software Project Enactment topics, bought belong to the Software Engineering Management Knowledge Area. Third the SPEM Ontology is discussed in the context of the SWEBOK topic “Validation and Verification“ from the Software Quality Management Process Knowledge Area. For the sake of clarity the utilizations are presented with several usage scenarios defined with description logic.

The rest of the paper is structured as follows. Section 2 presents a method of developing SPEM Ontology based on a transformation from MDA to the Semantic Web technical space. Section 3 presents relationship between SPEM Ontology and SWEBOK. Finally, Section 4 provides conclusion and future research direction.

2. Developing SPEM Ontology

SPEM is MDA standard used to define software and systems development processes and their components [15]. SPEM metamodel is based on MOF and reuses UML 2 Infrastructure Library [40]. Its own extended elements are structured into seven main meta-model packages. SPEM defines three compliance points (CP) above these packages, i.e.: the SPEM Complete CP, the SPEM Process with Behavior and Content CP and the SPEM Method Content CP. The scope of our solution is covered with Compliance Point "SPEM Process with Behavior and Content". The reason of focusing at this compliance point is because we need to work with separated reusable core method content from its application in processes, since a software method content can be used with arbitrary software process, such as iterative, agile etc...

2.1. SPEM conceptual framework

The Software and Systems Process Engineering Meta-model (SPEM) is a process engineering meta-model as well as conceptual framework, which can provide the necessary concepts for modeling, documenting, presenting, managing, interchanging, and enacting development methods and processes [15].

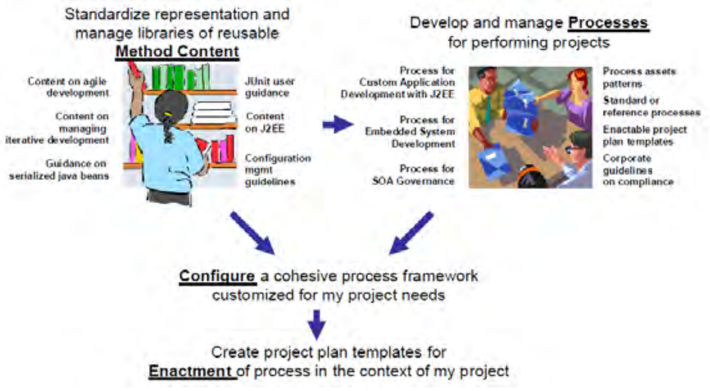

Standardize representation and manage libraries of reusable Method Content
Content on agile development
Content on managing iterative development
Guidance on serialized java beans
JUnit user guidance
Content on J2EE
Configuration mgmt guidelines
Develop and manage Processes for performing projects
Process for Custom Application Development with J2EE
Process for Embedded System Development
Process for SOA Governance
Process assets patterns
Standard or reference processes
Enactable project plan templates
Corporate guidelines on compliance
Configure a cohesive process framework customized for my project needs
Create project plan templates for Enactment of process in the context of my project

Fig. 1. SPEM 2.0's conceptual usage framework

Technically, the separation is represented by SPEM metamodel packages, i.e. the Method Content and the Process with Method metamodel packages. The former provides concepts for SPEM users and organizations to build up a development knowledge base that is independent of any specific processes and development projects. These concepts are the core elements of every method such as Roles, Tasks, and Work Product Definitions etc. The latter necessary metamodel package defines the structured work definitions that need to be performed to develop a system, e.g., by performing a project that follows the process. Such structured work definitions delineate the work to be performed along a timeline or lifecycle and organize it in so called breakdown structures. The most important elements of the Process with Method metamodel package are the Method Content Use elements. These elements are the key concept for realizing the separation of processes from method content and are great capabilities of SPEM. A Method Content Use can be characterized as a reference object for one particular Method Content Element, which has its own relationships and properties. When a Method Content Use is created, it shall be provided with congruent copies of the relationships defined for the referenced content element [15].

The last important metamodel package from the SPEM conceptual framework point of view is the Method Plugin metamodel package. The Method Plugin allows extensibility and variability mechanisms for Method Content and Process specification. It provides more flexibility in defining different variants of method content and processes by allowing content and process fragments to be plugged-in on demand, thus creating tailored or specialized content only when it is required and such that it can be maintained as separate units worked on by distributed teams [15]. Since the scope of SPEM is purposely limited to the minimal elements necessary to define any software and systems development process, the SPEM metamodel does not include elements such as Iteration, Phase etc. The reason is because for example not every software development process needs to have iterations. Therefore we had to include even the built-in SPEM Base Plugin to our method. It provides commonly used concepts for the domain of software engineering such as Phase, Iteration, Checklist etc.

2.2. Moving SPEM into the Semantic Web

In order to enable use of SPEM in the Semantic Web technical space, we make use of the fact that OWL, ODM and SPEM are serialized in XML format [41]. The OWL Metamodel is a MOF2 compliant metamodel that allows a user to specify ontologies using the terminology and underlying model theoretic semantics of OWL. The mapping between OWL and ODM is expressed in ODM that contains OWL Metamodel [42]. Thus only a mapping between SPEM and OWL is to be created. Since the hallmark work [11] proposes the transformation of a MDA standard to the Semantic Web technical space with a mapping between UML Ontology Profile and an arbitrary UML Profile, we have also used this principle. We have created a mapping between the Ontology UML Profile and the SPEM UML Profile. However, the mapping was not sufficient to create the SPEM Ontology. The main problem was that the SPEM UML Profile does not contain SPEM semantics, and moreover, it was not possible to derive a domain and range of a relationship, etc. Therefore we had decided to create semiautomatic transformation that is based on the merged SPEM metamodel to the SPEM UML Profile, where the result is the SPEM OWL DL Ontology. We have used OWL-DL, because this dialect of OWL retains computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time) [13]. To be conformed to this dialect, we have to adhere that an individual cannot be also a class, what is not violated in MDA technical space because of its 4 meta-layer architecture. For example, an analyst “Slávko Líška” is an instance of a Software Analyst SPEM class that is an instance of the Role Definition SPEM metaclass at the same time, thus the Software Analyst class is an individual and also a class. To avoid this problem in the Semantic Web technical space we have stated that a method content owl class is subclass of a SPEM owl class, and concrete individual is its instance. For example, the individual “Slávko Líška” is the instance of the Software Analyst owl class that is subclass of the Role Definition owl class from the SPEM Ontology. For more detailed and comprehensive description about the SPEM transformation to the Semantic Web technical space and its utilizations, a reader may refer to [43, 44].

3. SPEM Ontology application in the context of SWEBOK

Once we have SPEM Ontology created, we can apply it in various software process engineering cases. Since SWEBOK provides a consensually validated characterization of the bounds of the software engineering discipline and to provide a topical access to the defined software engineering Knowledge Areas [7], we also present the SPEM Ontology utilizations in such manner. The following subsections present SPEM Ontology application in selected topics of the SWEBOK Knowledge Areas.

3.1. Notation for process definition

The first SWEBOK topic that is directly related to the SPEM Ontology is the topic “Notations for Process Definition” of the Software Engineering Process Knowledge Area. Since at present the topic refers to non semantic notations only, the SPEM Ontology introduces new type of notation for process definition that is ontology based. Moreover, SPEM provides concepts also for a method definition that is another added value of such semantic notation. Forasmuch it is necessary to create correct SPEM models (method and process) it is efficient to use automated techniques for the models validation. Seeing that an OWL DL ontology supports reasoning we can also utilize it in a SPEM model verification. Two following scenarios present a SPEM model validation for the SPEM semantics verification.

- Scenario 1- SPEM method model validation with ontology: As it was already mentioned a method content model represents a model of development knowledge base that is independent of any specific processes and development projects. What the model does not define is how this method will be used in the process, whether it will be iterative, agile etc. Hence what a method content model validation can be good for? This scenario is essential to ensure that a method content is defined with the proper SPEM semantics. For example, whether a Task Definition has the proper domain of the “performs” relation that should be a Role Definition elements.

- Scenario 2- SPEM process model validation with ontology: The Method Content Use elements are the key concept for realizing the separation of processes from method content, thus a process model validation can be used to ensure, whether a process conforms to a method content definition it traces. Certainly, this validation scenario can be used similarly as the first one, hence for the validation, whether a process conforms to the SPEM semantics. For example, whether a Method Content Use element references one Method Content element only.

To be more precise, we give formally defined conditions that cover both validation scenarios. Formula 1 addresses the first validation scenario, which is the SPEM method model validation with ontology. Formula 2 addresses the second one, which is the SPEM process model validation with ontology. We say that a SPEM method model is consistent with the SPEM ontology if it is true that

$$\text{SPEM Ontology} \vdash \text{SPEM method ontology} . \qquad (\mathbf{1})$$

Similarly, we say that a SPEM process model is consistent with a SPEM method ontology and the SPEM Ontology if it true that

$$\text{SPEM Ontology} \vdash \text{SPEM method ontology} \vdash \text{SPEM process ontology}. \qquad (\mathbf{2})$$

3.2. Software Project Planning

Software project management is the art of balancing competing objectives, managing risk, and overcoming constraints to deliver a product that meets the needs of the customers and the end users [45]. Project management is accomplished through the use of processes such as: initiating, planning, executing, controlling and closing [46]. How the project will be managed and how the plan will be managed must also be planned. Reporting, monitoring, and control of the project must fit the selected software engineering process and the realities of the project, and must be reflected in the various artifacts that will be used for managing it. But, in an environment where change is an expectation rather than a shock, it is vital that plans are themselves managed. This requires that adherence to plans be systematically directed, monitored, reviewed, reported, and, where appropriate, revised [7]. To support these general objectives, we present an ontology based approach to project planning. We discuss two additional scenarios that could support the ontology oriented software process engineering. So, the third scenario presented in this work is Project plan creation with ontology scenario and the fourth one is Project plan verification with ontology.

- Scenario 3. Project plan generation with ontology. When a project manager want to create a project plan, he can create a SPEM method and process models first and then use OWL DL consistency reasoning to ensure that they are consistent. Then he can just simply transform his SPEM process model to the SPEM process ontology.

- Scenario 4. Project plan verification with ontology. This scenario is essential when a project manager wants to ensure that his already created project plan is consistent with desired method content and process. However, the scenario usually follows the previous one. A project manager obviously makes many changes to his project plan; therefore it is necessary to ensure that these changes do not violate the required consistency.

Since the third scenario is included in the fourth, we focus only at the Scenario 4. First we define the Project Plan Knowledge as a union of the SPEM Ontology, SPEM Base Plugin Ontology, a SPEM method ontology and a SPEM process ontology, as it is shown in Formula 3.

$$\text{Project Planning Knowledge} = \text{SPEM Ontology} \cup \text{SPEM Base Plugin Ontology} \cup \text{SPEM method content ontology} \cup \text{SPEM process ontology} . \quad (\mathbf{3})$$

Then we say, that the Project Planning Knowledge is satisfied in a project plan if it is true that

$$\text{Project Plan} \models \text{Project Planning Knowledge} . \quad (\mathbf{4})$$

From the First Order Logic point of view, the Project Plan Knowledge is the theory and a project plan is its model. Since a theory can have a model only if a theory is consistent [38], it is necessary, that the Formula 5 is either true

$$\text{SPEM Ontology} \vdash \text{SPEM Base Plugin Ontology} \vdash \text{SPEM method content ontology} \vdash \text{SPEM process ontology} . \quad (\mathbf{5})$$

For more information that includes either example a reader may refer to [32].

3.3. Software project enactment

The difficulty of software development is greatly enhanced when it is inevitable to cooperate with a supplier. The general issue is to manage a lot of differences such as different tasks, software work products, guidelines, roles etc [7]. The ideal state is that a company and its supplier use the same software framework and they use it in the same way. Otherwise risk of budget and time overrun together with quality decrease is greatly increased. Unfortunately, such an ideal state cannot exist. Either companies use different software frameworks, or they use the same software framework - but highly likely in different ways. It is natural that companies have different knowledge acquired from their various projects, and also have different experts with different experiences. Thus even if they use the same software framework, e.g. RUP [45], project enactment with the supplier, due to mentioned differences, is problematic.

Our approach to software process enactment with a supplier is based on OWL DL verification with a set of different method plugins, which represent different methods and processes of a company and its supplier. When OWL

DL verification results in inconsistency, it implies that the project cannot be enacted with a supplier and the source of inconsistency should be removed. Therefore the necessary condition to use this method is to have company's and supplier's software process specified with SPEM models. Next, there are presented several utilization scenarios of our approach, which extend the overall set of utilization scenarios mentioned in this work.

- Scenario 5 - Verification of the set of SPEM methods with ontology: This scenario can be used when it is necessary to verify whether at least two different SPEM method contents are consistent. Therefore its use for the software project enactment with supplier is appropriate. Since it is necessary to manage a lot of differences such as different tasks, software work products, guidelines, roles etc., this scenario can be used to reveal and to remove those differences that are inconsistent. For example a company can state that the Task Definitions "Create Requirements" and "Create Test Cases" should not be performed by the same person, because the creation of the Test Cases can reveal hidden inconsistencies that the author of requirements does not need to be aware of. On the other hand, a supplier's method can state that the same person can perform both task definitions. Hence, this is inconsistency and it should be removed.

- Scenario 6 – Verification of the set of SPEM processes with ontology: This scenario is similar to the previous, but this time processes of a software development are subject for verification. It is used to verify, whether at least two different SPEM processes are consistent. For example, a company's method content requires that the Task Definition "Create Requirements" should be executed in at least two iterations to increase the quality of requirements, but for the supplier,one iteration is also permissible. Again, this is an inconsistency and it should be removed.

- Scenario 7 – Project plan generation with the set of method plugins with ontology: This scenario can be executed when a project manager wants to create a project plan that is based on at least two method plugins. Even the ontology plays intermediate task of such scenario, its usability is crucial. First it is necessary to select the desired method contents and a process from the set of method plugins and transform them to ontologies. Then the scenario 5 and 6 are executed to reveal inconsistencies. If the ontologies are consistent, then the XSL based transformation to XML format of the project plan can be executed. Then it is quaranted that the resulted project plan is consistent with desired method plugins.

- Scenario 8 – Project plan verification of the set of method plugins with ontology: This scenario can be executed when a project manager wants to verify a project plan with a set of method plugins. The scenario has the same architecture as the project plan verification with ontology scenario (i.e., scenario #4). The only difference is in number of method contents and processes, which create the knowledge about project planning. When the mapping between the set of method plugins are created and their consistency is established, the project verification can be executed against these method plugins.

To be more precise, we give formally defined conditions that cover the mentioned utilization scenarios. Since the scenario 7 consists of the scenarios 5 and 6 we only present the formal specification of scenarios 5, 6 and 8.. Scenario 5 is covered with Formula 9, scenario 6 with Formula 10 and scenario 8 with Formula 8 and 11. First we define the two method plugins and the Project Planning Knowledge:

SPEM method plugin 1 ontology = SPEM method ontology 1 ∪ SPEM process ontology 1 (**6**)

SPEM method plugin 2 ontology = SPEM method ontology 2 ∪ SPEM process ontology 2 (**7**)

Project Planning Knowledge = SPEM Ontology ∪ SPEM Base Plugin Ontology ∪ SPEM method plugin 1 ontology ∪ SPEM method plugin 2 ontology (**8**)

Then we say that two SPEM method contents are consistent if:

SPEM method ontology ⊦ SPEM Base Plugin Ontology ⊦ SPEM method ontology 1 ⊦ SPEM method ontology 2 (**9**)

and two SPEM processes are consistent if:

SPEM method ontology ⊦ SPEM Base Plugin Ontology ⊦ SPEM method ontology 1 ⊦ SPEM method ontology 2 ⊦ SPEM process ontology 1 ⊦ SPEM process ontology 2 (**10**)

The Project Planning Knowledge is satisfied in a project plan if:

Project Plan **|=** Project Planning Knowledge (**11**)

Again, since a theory can have a model only if the theory is consistent, the necessary condition that enables Formula 11 to be true is that also Formula 10 must be true. For more informations which include also examples a reader may refer to [32].

3.4. Verification and Validation

The SPEM ontology can also be used in the context of Verification and Validation topic of the Software Quality Management Process Knowledge Area. Verification is an attempt to ensure that the product is built correctly, in the sense that the output products of an activity meet the specifications imposed on them in previous activities. Validation is an attempt to ensure that the right product is built, that is, the product fulfills its specific intended purpose [7]. Since SPEM does not concern with the content of work products (e.g. business processes, use cases), is it not possible to verify traceability between these inner elements. Therefore SPEM ontology alone is not sufficient for validation a work product. On the other hand, since the SPEM ontology consists of method and process models, it is possible to use it for evaluating whether a product is build correctly, i.e. with proper method and process. Therefore the SPEM Ontology can provide concepts for a work product verification. Usage scenarios for the Verification and Validation topic are the same as were mentioned in previous subsections.

4. Implementation

Ontologies rely on well-defined and semantically powerful concepts in artificial intelligence [47], such as description logics, reasoning, and rule-based systems [48]. Since we use OWL DL form of ontology, the implementation has a goal to present the proposed utilization scenarios with a Knowledge Representation System that supports description logics. Developing a knowledge base using a description logic language means setting up a terminology (the vocabulary of the application domain) in a part of the knowledge base called the TBox, and assertions about named individuals (using the vocabulary from the TBox) in a part of the knowledge base called the ABox [49]. In other words, the ABox describes a specific state of affairs in the world in terms of the concepts and roles defined in the TBox [11]. As we have it discussed in Subsection 2.2, our approach is conformed to the OWL DL dialect that disallows an individual to be simultaneously a class. Therefore all classes of the ontologies used in our approach constitute TBOX, whereas only individuals obtained from a project plan create ABOX as it is depicted in Table 1.

Table 1. Mapping between components of a knowledge based representation system to our approach's ontologies

Ontology type	KBRS component
SPEM Ontology	TBox
SPEM Base Method Plugin	TBox
SPEM method content ontology	TBox
SPEM process ontology	TBox
SPEM method plugin ontology	TBox
Individuals of a SPEM process ontology	ABox

For the sake of usability we have created OWL4SPEM: a semantic framework for software process engineering. It contain the SPEM Ontology, SPEM Method Plugin ontology, all mentioned XSL transformations that allows generating desired ontologies and lot of examples. For more information a reader may refer to [50].

5. Conclusion

We presented an approach to software process definition with the SPEM Ontology. For the sake of adoptability we presented applications of such semantic notation in the context of selected SWEBOK topics. First we discussed extension of the topic “Notations for Process Definition” with the SPEM Ontology and consequently we presented the ontology in the context of “Software Project Planning”, “Software Project Enactment”, and “Validation and Verification” SWEBOK topics. Since the relationship between the SPEM Ontology and the SWEBOK topics is more comprehensive than we described, it is necessary that the research will continue. However when we compare our approach with work that is perhaps closest to ours [38] it should be noted that we created not only wider method specification, but we also presented its implementation. It supports key property of SPEM, i.e. the Method Content separation from a Process and also the separation from the SPEM Base Plugin. Additionally, since a Method Plugin consists of a Method Content and a Process, our approach can be easily extended with any Method Plugin, for example, with the Rational Unified Process Plugin. However, we are aware that our research must continue in order to be applied successfully in real commercial projects. It is very difficult to imagine that for the purpose of project plan verification a project manager will use a knowledge based framework directly, without appropriate user interfaces. Therefore, we have started implementation of a macro for the MS Project that will remotely access OWL API for OWL-DL reasoning purposes and it will print verification results back into MS Project Plan. Additionally, we started to implement a semantic enterprise server with Jena, where the SPEM Ontology will stand as a facility plugin into the architecture. Finally, similarily to other related works, we have to include also SWRL to our approach to extend the expressiveness of description logic with the rule based expressions. The mentioned enhancements to our method are to be viewed as objectives of future research.

Acknowledgments. "This work was supported by the Scientific Grant Agency of Slovakia, grant No. VG1/0508/09, and it is a partial result of the Research & Development Operational Program for the project Support of Center of Excellence for Smart Technologies, Systems and Services II, ITMS 25240120029, co-funded by ERDF."

References

1. Software Productivity Consortium, "Process Definition and Modeling Guidebook," Software Productivity Consortium, SPC-92041-CMC, (1992)
2. IEEE/EIA 12207.0-1996//ISO/IEC12207:1995, Industry Implementation of Int. Std. ISO/IEC 12207:95, Standard for Information Technology-/Software Life Cycle Processes, vol. IEEE, (1996)
3. ISO/IEC TR 15504:1998, Information Technology - Software Process Assessment (parts 1-9): ISO and IEC, (1998)
4. D. Harel and M. Politi, Modeling Reactive Systems with Statecharts: The Statemate Approach: McGraw-Hill, (1998)
5. R. Radice, N. Roth, A. O. H. Jr. and W. Ciarfella, "A Programming Process Architecture," IBM Systems Journal, vol. 24, iss. 2, 79-90, (1985)
6. E. Yu and J. Mylopolous, "Understanding 'Why' in Software Process Modeling, Analysis, and Design," presented at Proceedings of the 16th International Conference on Software Engineering, (1994)
7. IEEE Computer Society. Software Engineering Body of Knowledge (SWEBOK). Angela Burgess, EUA, (2004)
8. Fujita, H., Zualkernan, I. A. (ed.): An Ontology-Driven Approach for Generating Assessments for the Scrum Software Process. In Proceedings of the seventh SoMeT_08. IOS Press, The Netherlands, 190-205, (2008)
9. Kurtev, I., Bézivin, J., Aksit, M.:Technological spaces: An initial appraisal. In Proceedings of the Confederated International Conferences, CoopIS, DOA, and ODBASE, Industrial Track, Irvine, CA, USA, (2002)
10. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.:EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley Longman, Amsterdam, (2009)
11. Gašević, D., Djurić, D., Devedžić, V.: Model Driven Engineering and Ontology Development, 2nd ed., Springer, Berlin, (2009)
12. Happel, H.J., Seedorf, S.: Applications of ontologies in software engineering. In: International Workshop on Semantic Web Enabled Software Engineering (SWESE'06), Athens, USA, (2006)
13. Mcguinness, D. L., Harmelen, F.: OWL Web Ontology Language Overview, W3C Recommendation, (2004). [Online]. Available: http://www.w3.org/TR/owl-features/ (current December 2010)
14. Smith, M.K., Welty, Ch., McGuinness, D.L.: OWL Web Ontology Language Guide, W3C Recommendation, (2004). [Online]. Available: http://www.w3.org/TR/owl-guide/ (current December 2010)
15. Object Management Group: Software and Systems Process Engineering Meta-Model 2.0, formal/2008-04-01. Object Management Group, USA, (2008). [Online]. Available: http://www.omg.org/technology/documents/formal/spem.htm (current December 2010)
16. Krdžavac, N., Gašević, D., Devedžić, V.: Model Driven Engineering of a Tableau Algorithm for Description Logics. Computer Science and Information Systems, Vol. 6, No. 1, (2009)
17. Frankel, D.S.: Model Driven Architecture. Applying MDA to Enterprise Computing. Willey, USA, (2003)
18. Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification, formal/2006-01-01. Object Management Group, USA, (2008). [Online]. Available: http://www.omg.org/spec/MOF/2.0/ (current December 2010)
19. Brickley, D., Guha, R. V., McBride, B.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation, (2004). [Online]. Available: http://www.w3.org/TR/rdf-schema/ (current December 2010)

20. Pan, J., Horrocks, I.: Metamodeling Architecture of Web Ontology Languages, In Proceedings of the First Semantic Web Working Symposium, Stanford, 131-149, (2001)
21. Object Management Group: UML 2.2 Superstructure Specification, formal/09-02-03. Object Management Group, USA, (2009). [Online]. Available: http://www.omg.org/technology/documents/formal/uml.htm (current December 2010)
22. Cranefield, S.: Networked Knowledge Representation and Exchange using UML and RDF. Journal of Digital Information, Volume 1 Issue 8, (2001)
23. Hart, L., Emery, P., Colomb, B., Raymond, K., Taraporewalla, S., Chang, D., Ye, Y., Kendall, E., Dutra, M.: OWL Full and UML 2.0 Compared. OMG TFC Report, (2004)
24. Polášek, I., Kelemen, J.: Ontologies in Knowledge Office Systems. In: KEOD 2009, 1st International Conference on Knowledge Engineering and Ontology Development, Funchal - Madeira, Portugal. INSTICC PRESS, 400-413, (2009)
25. Polášek, I., Chudá, D., Kristová, G.: Modelling System Dynamics in a Newer Version of UML (in Slovak). In: Proc. Systémová integrácia 2006. Žilina University, Žilina,311-317, (2006)
26. Hrnčič, D., Mernik, M., Forgáč, M., Kollár, J.: Evolution and Adaptation of Domain Specific Languages. In: Proc. of the Tenth International Conference on Informatics 2009, Technical University of Kosice, Kosice, 154-159, (2009)
27. Kollár, J., Porubän, J., Václavík, P., Bandáková, J., Forgáč, M.: Adaptive Language Approach to Software Systems Evolution. In: Proc. International Multiconference on Computer Science and Information Technology: 1st Workshop on Advances in Programming Languages (WAPL'07), Polish Information Processing Society, 1081-1091, (2007)
28. Object Management Group: Ontology Definition Meta-Model 1.0. formal/2009-05-01. Object Management Group, USA, (2009). [Online]. Available: http://www.omg.org/spec/ODM/1.0/ (current December 2010)
29. Gašević, D., Djurić, D., Devedžić, V.: MDA and Ontology Development. Springer, Berlin, Heidelberg, (2006)
30. Gašević, D., Djurić, D., Devedžić, V.: Bridging MDA and OWL Ontologies. Journal of Web Engineering, Vol. 4, no. 2, pp. 119–134, (2005)
31. Líška, M.: An Approach of Ontology Oriented SPEM Models Validation. In Proceedings of the First International Workshop on Future Trends of Model-Driven Development (FTMDD) in the context of the 11th International Conference on Enterprise Information Systems. Milan, Italy, 40-43, (2009)
32. Líška, M., Návrat, P.: An Approach to Project Planning Employing Software and Systems Engineering Meta-Model Represented by an Ontology. Computer Science and Information Systems Journal (COMSIS), Volume 7, Number 4, 2010, 721-736.
33. Líška, M., Návrat, P.: An Ontology Based Approach to Software Project Enactment with a Supplier. In 14th East-European Conference on Advances in Databases and Information Systems (ADBIS2010), Lecture Notes in Computer Science 6295, Novi Sad, Serbia, Springer, pp. 378-391, (2010)
34. Wang, S., Jin, L. J. CH. Represent Software Process Engineering Metamodel in Description Logic. In Proceedings of World Academy of Science, Engineering and Technology, vol. 11, (2006)
35. Object Management Group: Object Constraint Language 2.2. formal/2010-02-01. Object Management Group, USA, (2010). [Online]. Available: http://www.omg.org/spec/OCL/2.2/ (current December 2010)

36. Zualkernan, I. A. An Ontology-Driven Approach for Generating Assessments for the SCRUM Process. New Trends in Software Methodologies, Tools and Techniques. IOS Press, (2008)
37. Schwaber, K., Beedle, M. Agile Software Development with SCRUM. Prentice Hall, (2002)
38. Rodríguez, D., Sicilia, M., A.: Defining SPEM 2 Process Constraints with Semantic Rules Using SWRL. In Proceedings of the Third International Workshop on Ontology, Conceptualization and Epistemology for Information Systems, Software Engineering and Service Science held in conjunction with CAiSE'09 Conference. Amsterdam, The Netherlands, pp. 95-104, (2009)
39. Horrocks, I., Patel-Schneider, P., F., Boley, H., Tabet, T., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language, Combining OWL and RuleML. W3C Member Submission, (2004). [Online]. Available: http://www.w3.org/Submission/SWRL/ (current December 2010)
40. Object Management Group: UML 2.2 Infrastructure Specification, formal/2009-02-04. Object Management Group, USA, (2009). [Online]. Available: http://www.omg.org/spec/UML/2.2/ (current December 2010)
41. Object Management Group: MOF 2.0 / XMI Mapping Specification, v2.1.1, formal/2007-12-01. Object Management Group, USA, (2007). [Online]. Available: http://www.omg.org/spec/XMI/2.1.1/ (current December 2010)
42. Djurić, D.: MDA-based ontology infrastructure, Computer Science and Information Systems, Vol. 1, no. 1, pp. 91–116, (2006)
43. Líška, M.: Extending and Utilizing the Software and Systems Process Engineering Metamodel with Ontology. PhD Thesis, ID: FIIT-3094-4984. Slovak University of Technology in Bratislava, (2010)
44. Líška, M. Extending and Utilizing the Software and Systems Process Engineering Metamodel with Ontology. Information Sciences and Technologies, Bulletin of the ACM Slovakia, Vol. 2, No. 2, pp. 8-15, (2010)
45. Kruchten, P.: The Rational Unified Process: An Introduction. (3rd edition). Addison-Wesley, USA, (2003)
46. Project Management Institute: A Guide to the Project Management Body of Knowledge (PMBOK– 4th edition). Project Management Institute, USA, (2008)
47. Návrat, P. et al.: Artificial Intelligence, 2002, Slovak University of Technology in Bratislava. STU Press, (2002)
48. Kvasnička, V., Pospíchal, J.: Mathematical Logic. Slovak University of Technology in Bratislava. STU Press, (2005)
49. Baader, F., Horrocks, I., Saatler, U.: Description Logics. In Steffen Staab and Rudi Studer, editors, Handbook on Ontologies, International Handbooks on Information Systems, Springer. 3-28, (2004)
50. Líška, M., Návrat, P.: OWL4SPEM – A semantic framework for software process engineering, (2010). [Online]. Available: http://www.ontologia.sk/owl4spem/ (accessed February 22, 2011)
51. Habala O., Paralič M., Rozinajová V., and Bartalos P.: Semantically-Aided Data-Aware Service Workflow Composition. In: M. Nielsen et al. (Eds.): SOFSEM 2009, LNCS 5404, pp. 317–328, 2009, Springer-Verlag Berlin Heidelberg 2009
52. Bartalos, Peter - Bieliková, Mária: QoS Aware Semantic Web Service Composition Approach Considering Pre/Postconditions. In: IEEE ICWS 2010, Eighth International Conference on Web Services, Miami, Florida, 5-10 July 2010 : Proceedings. - Los Alamitos : IEEE Computer Society, 2010. - ISBN 978-0-7695-4128-0. - pp. 345-35254.

53. Bartalos, P. Effective Automatic Dynamic Semantic Web Service Composition. Information Sciences and Technologies, Bulletin of the ACM Slovakia, Vol. 3, No. 1, (2011)

Miroslav Líška received the M.S. degree in informatics from the Technical University in Košice, Slovakia in 2002, and the PhD. degree in software and information systems from the Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technologies, Slovak University of Technology in Bratislava in 2010. His interests include semantic web, ontologies, software process engineering and semantic enterprise oriented architectures. He currently works as a semantic web architect in Datalan, in Bratislava, Slovakia.

Pavol Navrat received his Ing. (Master) cum laude in 1975, and his PhD. degree in computing machinery in 1984 both from Slovak University of Technology in Bratislava. He is currently a professor of Informatics at the Slovak University of Technology and serves as the director of the Institute of Informatics and Software Engineering. During his career, he was also with other universities abroad. His research interests include related areas from software engineering, artificial intelligence, and information systems. He published numerous research articles, several books and co-edited and co-authored several monographs. Prof. Navrat is a Fellow of the IET and a Senior Member of the IEEE and its Computer Society. He is a Senior Member of the ACM and chair of the ACM Slovakia Chapter. He is also a member of the Association for Advancement of Artificial Intelligence, Slovak Society for Computer Science and Slovak Artificial Intelligence Society. He serves on the Technical Committee 12 Artificial Intelligence of IFIP as the representative of Slovakia.

Received: December 30, 2010; Accepted: March 02, 2011.

DOI: 10.2298/CSIS101231019C

Ontology Driven Development of Domain-Specific Languages

Ines Čeh, Matej Črepinšek, Tomaž Kosar, and Marjan Mernik

Faculty of Electrical Engineering and Computer Science, Smetanova 17,
2000 Maribor, Slovenia
{ines.ceh, matej.crepinsek, tomaz.kosar, marjan.mernik}@uni-mb.si

Abstract. Domain-specific languages (DSLs) are computer (programming, modeling, specification) languages devoted to solving problems in a specific domain. The development of a DSL includes the following phases: decision, analysis, design, implementation, testing, deployment, and maintenance. The least-known and least examined are analysis and design. Although various formal methodologies exist, domain analysis is still done informally most of the time. A common reason why formal methodologies are not used as often as they could be is that they are very demanding. Instead of developing a new, less complex methodology, we propose that domain analysis could be replaced with a previously existing analysis in another form. A particularly suitable form is the use of ontologies. This paper focuses on ontology-based domain analysis and how it can be incorporated into the DSL design phase. We will present the preliminary results of the Ontology2DSL framework, which can be used to help transform ontology to a DSL grammar incorporating concepts from a domain.

Keywords: domain-specific language, domain analysis, ontology.

1. Introduction

Programming languages are used for human-computer interaction. Depending on the purpose of their use, programming language can be divided into general-purpose languages (GPLs) and domain-specific languages (DSLs) [1], [2], [3], [4]. GPLs, such as Java, C and C#, are designed to solve problems from any problem area. In contrast to GPLs, DSLs, such as Latex, SQL and BNF, are tailored to a specific application domain.

When developing new software, a decision must be made as to which type of programming language will be used: GPL or DSL. The issue is further complicated if an appropriate DSL does not exist. Then, the decision becomes whether to start to develop with a GPL language or to start with the development of the required DSL and then develop the software system with it. Reasons for using a DSL are as follows: easier programming, re-use of semantics, and the easier verification and programmability for end-users [1], [2]. However, using a DSL also has its disadvantages, such as high development costs [1], [5]. The key is to answer the question: »When to develop a DSL?« The simplest answer to this question is: a DSL should be developed whenever it is necessary to solve a problem that belongs to a problem family and when we expect that in the future more problems from the same problem family will appear. A more detailed response can be found in [1].

DSL development consists of the following phases: decision, analysis, design, implementation, testing, deployment and maintenance [1], [6], which are discussed in greater detail in Section 2. While the implementation phase has attracted a lot of researchers [5], some of the DSL development phases are less known and are not as closely examined (e.g. analysis, design).

The knowledge of the problem domain and its definition is achieved at the domain analysis phase. Various methodologies for domain analysis have been developed. Examples of such methodologies include: DSSA (Domain Specific Software Architectures) [7], FODA (Feature-Oriented Domain Analysis) [8], and ODM (Organization Domain Modeling) [9]. Often, formal methodologies are not used due to complexity and the domain analysis is done informally. This has the consequence of complicating future DSL development. Even if the domain analysis is done with a formal methodology, there are not any clear guidelines on how the output from domain analysis can be used in a language design process. The outputs of domain analysis consist of domain-specific terminology, concepts, commonalities and variabilities. Variabilities would have been entries in the design of DSL, while terminology and concepts should be reflected in the DSL constructs, and commonalities could be incorporated into the DSL execution environment. Although it is known where the outputs of the domain analysis should be used, there is a need for clear instructions on how to make good use of the information, which are retrieved during the analysis phase, in the design stage of the DSL.

To partially solve the aforementioned problems, we propose that domain analysis (hereinafter referred to as classic domain analysis (CDA)) be performed with the use of existing techniques from other fields of computer science. A particularly suitable one is the use of ontologies [10], [11], [12]. An ontology provides the vocabulary of a specialized domain. This vocabulary represents the domain objects, concepts and other entities. Some types of domain knowledge can be obtained from the relationships of the entities, as presented by the vocabulary. Ontologies in the CDA have already been used in [13]. Whereas Tairas et al. apply ontologies in the early stages of domain analysis to identify domain concepts; we propose that an ontology replace the CDA. They also investigated how ontologies contribute to the design of the language [13]. Ontologies in connection with DSL are also used by other authors. Miksa et al. applied ontology-enabled software engineering in the area of DSL engineering [14]. Guizzardi et al. proposed the use of an upper ontology (top-level ontology) [15] to design and evaluate domain concepts [16]. Walter et al. applied ontologies to describe DSL [17]. Bräuer and

Lochmann proposed an upper ontology to describe interoperability among DSLs [18].

The proposed solution of the first problem, the use of ontologies, has a significant effect on the second problem, related to CDA. It translates the problem »How to make good use of the information, retrieved during the analysis phase, in the design stage of the DSL?« into the problem »How to make good use of the information contained in an ontology in the design stage of a DSL?« This paper focuses on ontology-based domain analysis (OBDA) and how it can be incorporated into the DSL design phase. We will present the preliminary results of the Ontology2DSL framework, which can be used to help transform an ontology to a DSL grammar.

The organization of this paper is as follows. Section 2 presents the background information required for the understanding of this paper. Section 3 is intended to demonstrate the similarities and variabilities between the CDA and OBDA. Section 4 presents the transformation rules used for the development of a DSL from an ontology, as well as the example of an ontology to a DSL transformation. Section 5 presents the framework Ontology2DSL and its architecture. The conclusion and future work are summarized in Section 6.

2. Background

2.1. DSL development phases

In [1], the authors have identified the following DSL development phases: decision, analysis (CDA), design, implementation, and deployment. The additional phases are testing and maintenance. The maintenance phase was introduced in [6]. Fig. 1 presents these phases along with the input and output of every phase and examples of patterns for the individual phases. The decision phase provides the answer to the question of when to develop a DSL. Other phases focus on the question of how to develop it. DSL and GPL development processes have a few differences with respect to the phases of development, since the phases are identical. The differences are in the activities, approaches and techniques used in the individual phases. The difference is expressed in the greater diversity of the activities, approaches and techniques in DSL development. It should be taken into consideration that DSL development is not a simple sequential process. Often, the phases overlap one another. For instance, the design of the DSL is influenced by the decision on the implementation approach. In the following section, the DSL development phases are briefly discussed.

Decision. It is often far from evident that a DSL might be useful or that developing a new one might be worthwhile. The concepts underlying a suitable DSL may emerge only after a lot of GPL programming has been done. Decision patterns [1] describe situations (e.g., task automation, domain-specific Analysis, Verification, Optimization, Parallelization, and Transformation (AVOPT)) for which, in the past, developing a new DSL was fruitful.

Domain analysis (CDA). The precondition of the design and implementation of a DSL is a detailed domain analysis. The goal of CDA is to select and define the domain of focus and collect appropriate domain information and integrate them into a coherent domain model; the result of CDA [19]. A representation of the domain system properties and their dependencies is the domain model. The properties are either common or variable, which is represented in the model along with the dependencies between the variable ones. Besides the development of the domain model, CDA also includes domain planning, identification and scoping. The inputs to the domain analysis are different sources of implicit and explicit domain knowledge. The information sources for the analysis are: technical literature, existing implementations, customer demands, expert advice, and current and future requirements [4]. An important note is the fact that the domain analysis process not only collects existing information. The systematic and organized collection of existing information enables and encourages the extension of information with new knowledge. In some cases, CDA can be informal, while in others it incorporates different methodologies. Methodologies differ based on the degree of formality, information extraction techniques or their products. We have listed the most known methodologies in the introduction. FODA has been proven as the most commonly used formal methodology in DSL development. The domain analysis can result in different DSLs. However, they all share essential information acquired in the domain analysis phase.

Design. Language design includes the definition of constructs and language semantics. The semantics formalize the meaning of every construct in the language and the behavior not specified in the program. The approaches to the design of a DSL can be classified into two orthogonal dimensions: the relation between DSL and a computer language and the formality degree of the DSL description [1]. The first dimension refers to the exploitation of an existing language (GPL or DSL) or the invention of a new language. The most basic method for DSL construction is if the DSL is based on an existing language. The existing language can be: partially reused (piggyback pattern), limited (language specialization pattern) or extended (language extension pattern) [1]. The advantages of building a DSL on an existing language are: easier implementation and the familiarity of the development environment to users who are experienced with the existing language. If the connection between the DSL and an existing language does not exist, a new language must be developed from the beginning. The second dimension refers to the informal and formal design of the language. With informal design, the specification is usually in the natural language with optional program examples. When the design is formal, the specification is usually in the form of a well-known formal definition method (BNF for syntax specification, attribute grammars, denotational semantics, or algebraic specifications for semantic specification).

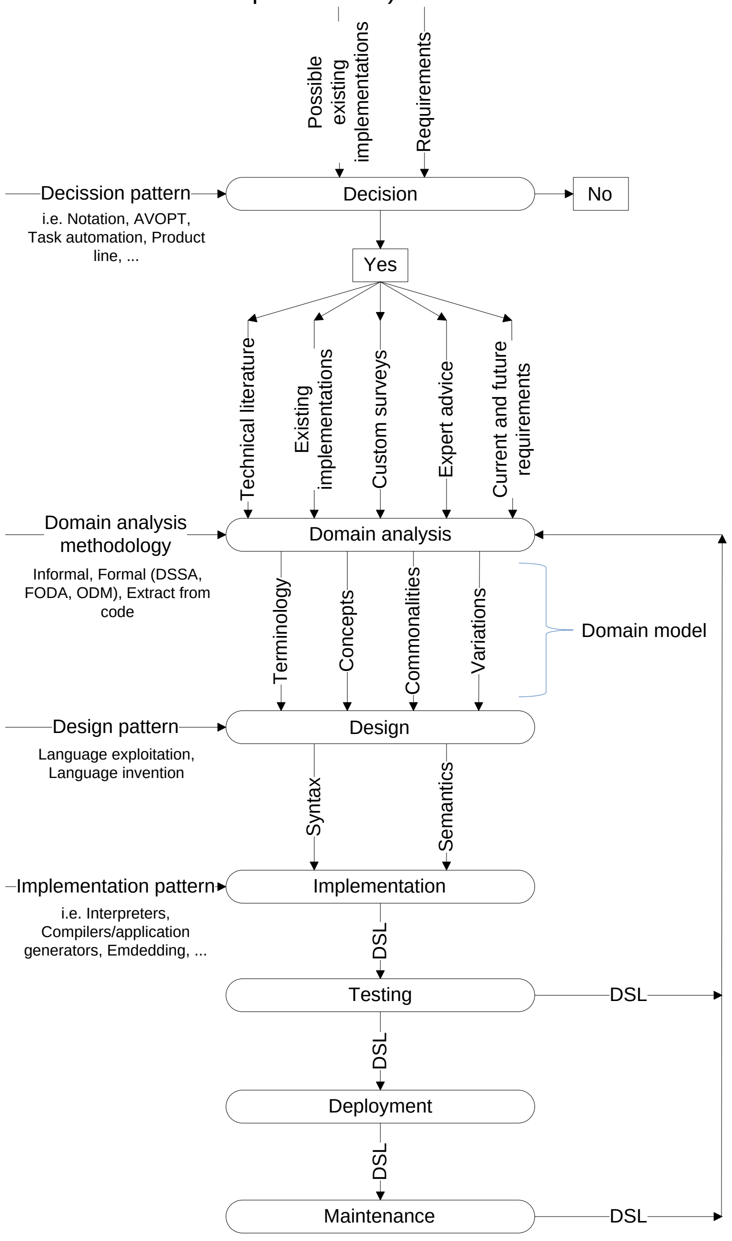

Possible existing implementations
Requirements
Decission pattern
i.e. Notation, AVOPT, Task automation, Product line, ...
Decision
No
Yes
Technical literature
Existing implementations
Custom surveys
Expert advice
Current and future requirements
Domain analysis methodology
Informal, Formal (DSSA, FODA, ODM), Extract from code
Domain analysis
Terminology
Concepts
Commonalities
Variations
Domain model
Design pattern
Language exploitation, Language invention
Design
Syntax
Semantics
Implementation pattern
i.e. Interpreters, Compilers/application generators, Emdedding, ...
Implementation
DSL
Testing
DSL
DSL
Deployment
DSL
Maintenance
DSL

Fig. 1. DSL development phases

Implementation. Different approaches for DSL development can be used, such as: interpreter, compiler/application generator, embedding, preprocessing, extensible compiler/interpreter, Commercial Off-The-Shelf (COTS), and hybrid. The approaches are presented in greater detail in [1]. Cleary we want to select the approach that requires the least effort during implementation and offers the greatest efficacy to the end user. The link between the implementation approaches, the effort of implementation and the efficacy to the end user is presented by the authors of [5].

Testing. In this phase, a DSL evaluation is performed. As shown in the study [20], this phase is often skipped or relaxed by language developers. The skipping or relaxing of this phase is not desirable because it may lead to the development of inadequate languages.

Deployment. In this phase, DSLs and applications constructed with them are used.

Maintenance. In this phase, the DSL is updated to reflect new requirements.

The number of phases and their individual complexity result in the discovery of high costs when developing a new DSL. DSL development requires domain knowledge and language expertise [1], [5]. These are the main reasons why DSLs are not so often used for solving software engineering problems. The development cost is seen as the greatest disadvantage of DSLs [5].

The main goal of the presented research is to investigate if the classic domain analysis (CDA) phase can be adequately replaced with an already existing domain knowledge and representation (e.g., ontology). In this manner the DSL development cost could be minimized.

2.2. Ontology

There are many definitions of ontologies in existing literature and one of the most commonly used definitions is that of Studer et al. They defined an ontology as follows: »An ontology is a formal, explicit specification of a shared conceptualization.« [12]. The meaning of the Studer et al. definition is detailed in [21]. Formal refers to the fact that it is machine readable. The specification is explicit because it summarizes the concepts, properties and relations between concepts. Furthermore, the shared conceptualization contains knowledge that a group of experts has agreed upon. Conceptualization refers to the fact that it incorporates the target domain completely.

Ontologies are commonly encoded using ontology languages. Ontology languages allow for the acquisition of knowledge about specific domains and often include rules that allow the processing of knowledge in existing ontologies. Ontology languages can be divided into two major groups: traditional (i.e. Flogic, Ontolingua) and web-based languages (i.e. RDF(S), OWL) [22]. Recently, a new group of languages, rule-based (i.e. RuleML, SWRL) [23], has emerged. These languages differ in their purpose and in their expressive power. The main requirements for an ontology language are:

a well-defined syntax, well-defined semantics, efficient reasoning support, sufficient expressive power and convenience of expression [21].

3. Comparison of CDA and OBDA

Subsection 3.1 presents the FODA methodology with which the domain analysis is performed. Subsection 3.2 introduces the ontology language OWL. The examples in both subsections are for the case of a home robot [24]. Subsection 3.3. compares the information obtained through FODA and through ontology domain analysis.

3.1. FODA

FODA is a CDA method that was developed by the Software Engineering Institute [19]. It is known for its models and feature modeling. In FODA, a feature is an end-user characteristic of a system. A FODA process consists of two phases: context analysis and domain modeling. The goal of context analysis is to determine the boundaries (scope) of the analyzed domain. The purpose of domain modeling is to develop a domain model. The FODA domain modeling phase is comprised of the following steps: information analysis, features analysis, and operational analysis. The main goal of information analysis is to capture domain knowledge in the form of domain entities and the links between them. The result of information analysis is the information model. The result of feature analysis is a feature model, which is presented below. An operational analysis results in the operational model. It represents how the application works and covers the links between objects in the informational model and the features in the feature model. An important product from the phase of domain modeling is the domain dictionary. It defines the terminology used in the domain and it also includes textual definitions of domain concepts and features.

A feature model consists of the following:

- The Feature diagram (FD) represents a hierarchical decomposition of features and their kinds (mandatory, alternative, and optional feature). Mandatory features are those that each system must have in the domain. Alternative features are features that a system can only possess one at a time. Optional features are features that a system may or may not have. A system can also have more than one feature at a time. These features are called or-features. Features are also classified as atomic or composite. Whereas atomic features cannot be further subdivided into other features, composite features are defined in terms of other features. The root node of the diagram represents a concept and the remaining nodes represent features. An example of a feature diagram is shown in Fig. 2.
- Feature definitions describe all features (semantics).

- Composition rules for features describe which combinations are valid or invalid.
- Rationale for features represents the reasons for choosing a feature.

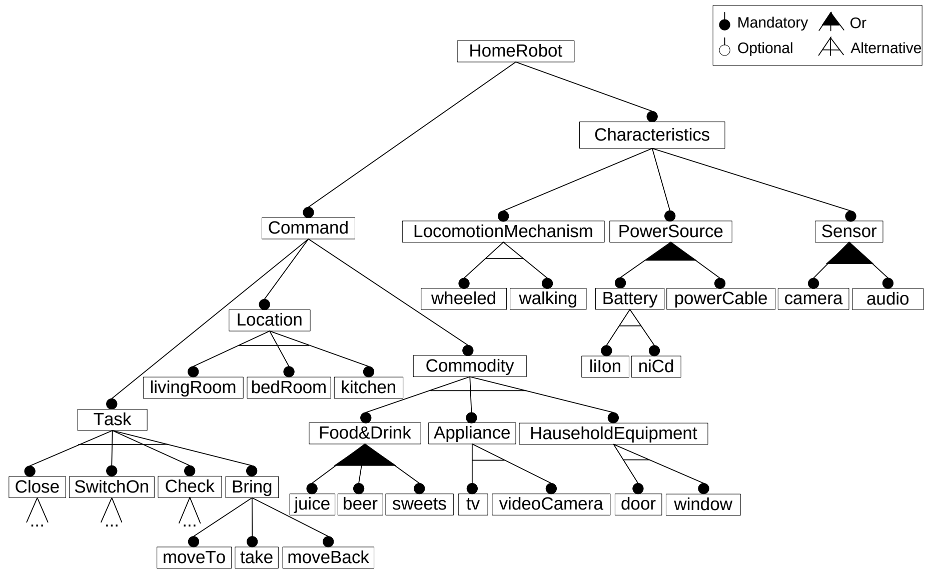


Fig. 2. Feature Diagram for a concept of a HomeRobot

Fig. 2 represents a simple FD of a HomeRobot. The root node of the diagram, HomeRobot, represents a concept; the remaining nodes represent its features. Whereas mandatory features are indicated by a filled circle, optional features are indicated by an empty circle. Alternative and or-features are both indicated by a triangle, the former with an empty one and the latter with a filled triangle. The names of atomic features are written in lower-case while the composite features are written with their first letter in upper-case. Every house robot has individual characteristics and executes some commands. Every house robot has to have a PowerSource, uses some sensors for its locomotion and moves in a particular manner. Every robot can have multiple power sources but only one mechanism of motion. Every robot performs tasks, which are comprised of subtasks. The robot executes one order per location, every task is focused on an item, while some tasks focus on multiple items at the same time.

Feature models are not only represented in the visual form of FDs but also in textual form. Van Deursen and Klint have proposed the feature description language (FDL) for the textual representation. The FDL definition constitutes the feature definitions followed by a colon (":") and the features expression. Possible feature expression forms are presented in [25]. FDL exceeds the graphic feature diagram in terms of expressive power and is appropriate for automatic processing. The FD for a home robot in FDL is shown in Fig. 3.

```
HomeRobot: all ( Command, Characteristics )
Command: all ( Task, Location, Commodity )
Characteristics : all ( LocomotionMechanism,
PowerSource, Sensor )
Task: one-of ( Close, SwitchOn, Check, Bring )
Bring: all ( moveTo, take, moveBack )
Location: one-of ( livingRoom, bedRoom, kitchen )
Commodity: one-of ( Food&Drink, Appliance,
                    HauseholdEquipment )
Food&Drink: more-of ( juice, beer, sweets )
Appliance: one-of ( tv, videoCamera )
HauseholdEquipment : one-of ( door, window )
LocomotionMechanism: one-of ( wheeled, walking )
PowerSource: more-of ( Battery, powerCable )
Sensor : more-of ( camera, audio )
Battery: one-of ( liIon, niCd )
```

Fig. 3. FD for the home robot in FDL

An important role of the FDs is to describe the variability of the programming system. The number of all possible configurations per system can be calculated with the use of variability rules, as presented in [25].

Constraints, which are intended for variability reduction, are an optional component of the FDs. The constraints are enforced with satisfaction rules [25]. The constraints are of two types [25]: diagram constraints and user constraints. The former include the “A1 requires A2” (if the feature A1 is presented, then feature A2 should also be presented) and “A1 excludes A2” (if feature A1 is presented, then feature A2 should not be presented) constraints, while the latter include the “include A” (feature A should be present) and “exclude A” (feature A should not be present) constraints.

3.2. OWL

OWL is the most commonly used ontology language. It was created on the basis of RDFS [10], [11]. It has three sublanguages; OWL Full, OWL DL, and OWL Lite [10], [11], [21]. These sublanguages have different levels of expressiveness. Whereas OWL Full is the most expressive, OWL Lite is the least expressive. Only OWL-DL allows automated reasoning.

The three components of OWL are: classes, properties, and individuals.

Classes are interpreted as sets that contain individuals. Classes may be organized into a hierarchy. This means that a class can subsume other classes or it can be subsumed by other classes. The consequence of the subsumption relation is inheritance. Inheritance refers to the inheritance of properties, which the children inherit from their parents. Whereas some ontologies only allow single inheritance, most ontologies, like OWL, allow multiple inheritance. OWL defines two special classes called „Thing“ and

„Nothing“. Class Thing is the most general class and it is the superclass of every class that is included in the ontology. Class Nothing is empty and it is the subclass of every included class. The class hierarchy of the Home robot ontology (HRO) is represented by Fig. 4. HRO is based on [24]. The main functionalities of the robot are comprised of common household tasks, such as checking if the window is closed. HRO formalizes terms for three classes: locations, items and tasks. The locations are physical places where tasks are performed. Items are part of the tasks in the manner that the same action is performed on them or with them. The tasks are the actions being performed. Each task is comprised of subtasks. For the HRO annotation we used the OWL-DL, a sublanguage of the Ontology Web Language (OWL). The tool used for the creation of the ontology was Protégé [26], [27].

The second component, the properties, is a binary relation. OWL defines two main kinds of properties: object properties and datatype properties. Whereas object properties relate objects to other objects, datatype properties relate an object to datatype values. OWL supports XML schema primitive datatypes.

The third component, the individuals, is the basic component of an ontology. They represent objects in the domain of discourse. They can be concrete individuals (i.e. animals, airplanes, and people) as well as abstract individuals (i.e. words and numbers).

3.3. Comparison

Both analysis incorporate a concept vocabulary, enable the display of property and class hierarchies, and provide a constraint mechanism (see Table 1). The CDA uses this mechanism for variability reduction while the OBDA uses it for the description of class properties. Both types of analysis describe semantics and are machine readable.

Table 1. Comparison of CDA and OBDA

Property	FD + FDL	OWL ontology
Concept vocabulary	Features names	Name Class or property
Hierarchy	Feature diagram	Class hierarchy
Constraints	FDL constraints	Restrictions
Rationale	FD rationale properties	No
Objects	No	Individuals
Possible combinations	Variability rules (FDL)	No
Reasoning support	No	Reasoners (i.e. FaCT++)
Machine readable	Yes	Yes
Tools	Yes	Yes (i.e. Protege)
Semantics	Yes	Yes
Query support	No	Yes (DL Query)

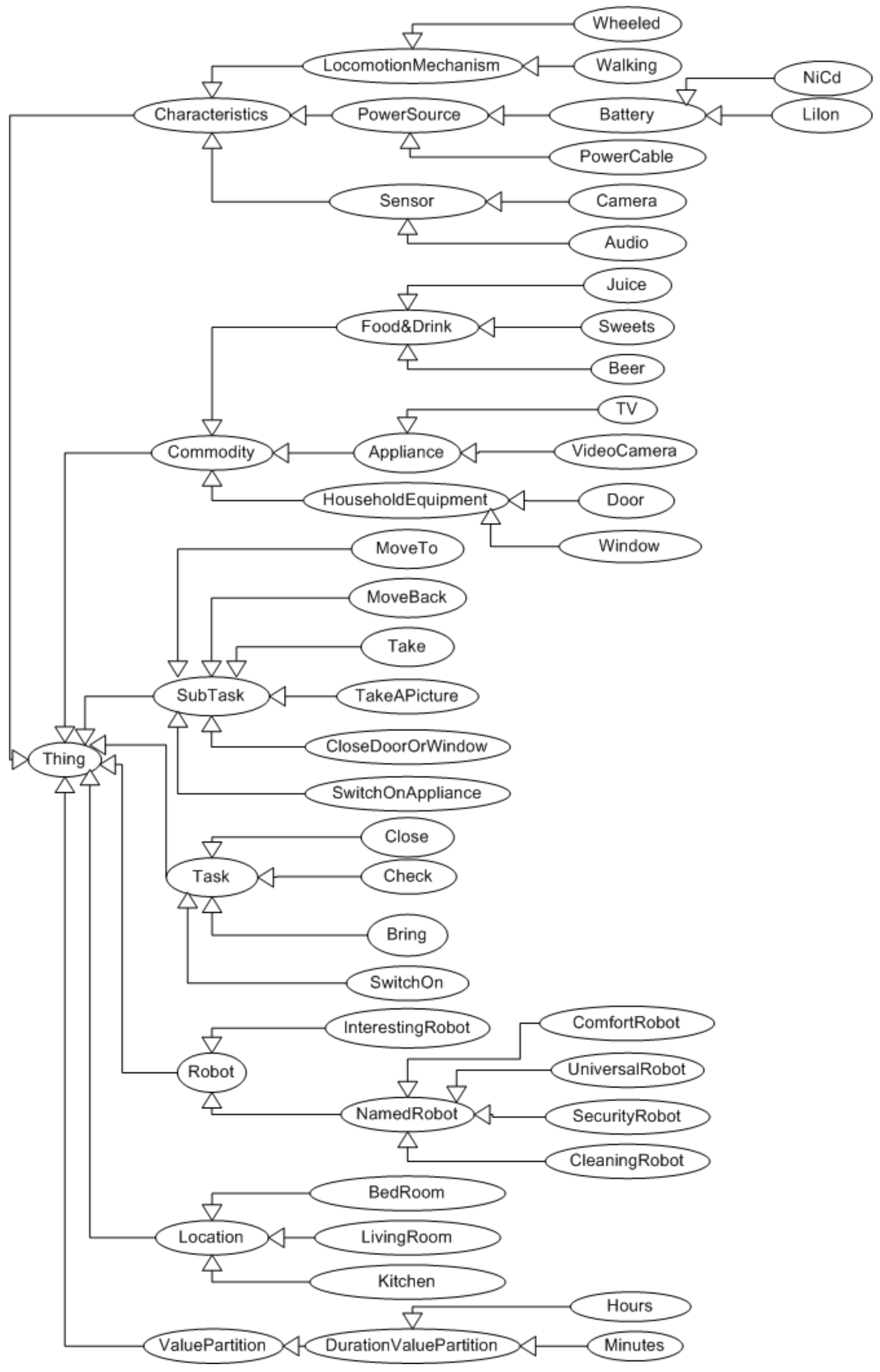


Fig. 4. Class hierarchy of a Home robot ontology

The CDA differs from OBDA in its capability to record the reasons for the use of a particular property (rationale) and the calculation of all possibilities. OBDA, on the other hand, provides the existence of objects, reasoning and querying. Numerous tools are available for it and ontologies are created across diverse research areas and are therefore available for use.

The comparison shows that OBDA is capable of most of what the CDA is capable of doing. The advantages of an ontology are reasoning and querying, because they enable the validation of an ontology. A valid ontology significantly reduces or prevents errors in DSL development. Semantics, which are inherently defined with the ontology, are also of great use when developing language semantics. Existing tools provide easy access to the ontology and enable efficient information extraction procedures. It is also a very important fact that ontologies are present in different areas of research. This provides the method for elimination of the domain analysis phase in DSL development and might significantly reduce the time needed for language development.

The comparison leads to the conclusion that the CDA can indeed be replaced with OBDA, primarily because the ODBA provides everything needed for DSL development and also adds new capabilities.

4. Designing the DSL grammar

While the previous chapter shows that OBDA is appropriate for DSL development, this chapter demonstrates the process of grammar [28] construction from an OWL ontology. Section 4.1 introduces the rules used in the transformation from an ontology to its corresponding grammar. Section 4.2, however, is the application example which demonstrates the usage of the rules presented in 4.1.

4.1. Transformation rules

The input in the transformation is a data structure named the ontology data structure (ODS), which carries the data extracted from the OWL document. The result of the transformation is again a custom data structure, in this case the grammar data structure (GDS). GDS is a formal annotation of the resulting grammar.

With regard to the effect the transformation has on the ODS, the rules can be divided into the following two groups: (1) rules that do not affect the ODS and (2) rules that do affect and alter it. The results of the former can only be observed on the GSD, while the results of the latter are identifiable on both ODS and GDS. The rules that affect ODS can be used to alter the ontology.

The rules that do not alter the primary data structure:

1. $R_1(C) \rightarrow N_1 \in N$ (R - rule, C - class, N_1 - nonterminal, N - set of nonterminals). Rule R_1 is used to convert class C into nonterminal N_1. An application example for rule R_1 can be seen in chapter 3.2, step 1.

2. $R_2(C) \rightarrow T_1 \in T$ (T_1 - terminal, T - set of terminals). Rule R_2 is used to convert class C into terminal T_1. The user can, if necessary, change the name of terminal T_1. Name changes must be recorded in the dictionary.

3. $R_3(C) \rightarrow P \cup \{C ::= "C_N"\}$ (C_N - class name, P - set of productions, G - grammar ($G =< T, N, S, P >$)). Rule R_3 is used to add production into the grammar G. An application example for the rule R_3 can be seen in chapter 3.2, step 2.

4. $R_4(C_1, C_2) \rightarrow P \cup \{C_1 ::= C_2\}$. Rule R_4 is used to add production into the grammar G.

5. $R_5(C, CL_D, type) \rightarrow P \cup \{C ::= C_1 | C_2 | ... | C_n | type\}, C_1 ... C_n \in CL_D$ if $C_D \in L(CH) \rightarrow C_D \in T$ (L - list, CH - class hierarchy, CL_D - list of disjoint classes [17], C_D - disjoint class, $type$ - string, integer, …). The rule R_5 accepts the following inputs: class C, list of disjoint classes CL_D and $type$. The inputs CL_D and $type$ are optional. In any case, at least one of them must be present. If the inputs C and CL_D exist we are talking about the rule R_{5a}. If inputs C and $type$ exist we are talking about the rule R_{5b}. If all three inputs exist we are talking about the rule R_{5c}. Rule R_{5a} is used to add a production in the form $C ::= C_1 | C_2 | ... | C_n$ into grammar G. Rule R_{5c} is an extension of rule R_{5a} and is used to add productions in the form $C ::= C_1 | C_2 | ... | C_n | type$ into grammar G. A precondition for the successful transformation is that the children are disjointed; otherwise the resulting grammar is not a context free grammar. Classes from CL_D that

are also leafs of the class diagram, are transformed, with the use of rule R_3 to the set of terminals T. Rule R_{5b} is used to add productions in the form $C ::= type$ into grammar G. The rule enables grammar generalization, as described by class C, and the associated part of ontology. Each rule is used according to the bottom up principle; first on the lower of the class hierarchy levels, followed by the higher classes. An application example for the rules R_{5a} and R_{5c} can be seen in chapter 3.2, step 2.

6. $R_6(C, PL) \rightarrow \text{if } PL = \{\ \} \mid (\forall C \in PL) \cap LAC \rightarrow P \cup \{S ::= C\}$

   (PL - parent list, S - start symbol, LAC - list of anonymous classes). Rule R_6 is used to define grammar start symbols. Class C is a possible grammar start symbol in the case that its parent list PL is empty or if all classes from PL are anonymous classes. Grammars can have more than one start symbol. An application example for the rule R_6 can be seen in chapter 3.2, step 5.

The rules that affect and alter the primary data structure:

7. $R_7(C, O, NC) \rightarrow P \cup \{C ::= typeTrans(NC)\}, O \in \{*, +, ?\}$ (O - operator). Rule R_7 is used to formalize the number of repetitions of some classes. The rule accepts the following inputs: class C, operator O, which defines the number of repetitions of some class and the new class NC. The rule is carried out in three steps. In the first step, the children of class C are assigned to class NC ($Child(NC) = Child(C)$). In the second step, the children of class C are removed ($RemoveChilds(C)$). In the third step the production is formalized. An application example for rule R_7 can be seen in chapter 3.2, step 6.

8. $R_8(C, T_L, T_R, NC) \rightarrow P \cup \{C ::= T_L\ Nc\ T_R\}$ (T_L - left terminal, T_D - right terminal). Rule R_8 is used to enrich the syntax. Either the left or the right terminal can be omitted. The rule is carried out in three steps. In the first step, the children of class C are assigned to the class NC ($Child(NC) = Child(C)$). In the second step, the children of class C

are removed ($\mathrm{RemoveChilds(C)}$). In the third step, the production is formalized. An application example for rule R_8 can be seen in chapter 3.2, step 7.

9. $\mathrm{R}_{9some}(\mathrm{C}_1, \mathrm{C}_2, \mathrm{T}_\mathrm{M}, \mathrm{NC}) \rightarrow \mathrm{P} \cup \{\mathrm{C}_1 ::= \mathrm{NC}\,\mathrm{T}_\mathrm{M}\,\mathrm{C}_2\}$ (T_M - middle terminal). Rule R_{9some} is used to formalize productions which describe restrictions (some). The rule accepts the following inputs: class C_1 to which the restriction refers, class C_2 which determines the possible values of class C_1, the middle terminal T_M and the new class NC. The rule is carried out in three steps. In the first step, the children of class C_1 are assigned to class NC ($\mathrm{Child(NC)} = \mathrm{Child(C)}$). In the second step the children of class C_1 are removed ($\mathrm{RemoveChilds(C_1)}$). In the third step the production is formalized.

10. $\mathrm{R}_{10}(\mathrm{C}, \mathrm{LofCs}) \rightarrow \mathrm{P} \cup \{\mathrm{C} ::= \mathrm{C}_1\ \mathrm{C}_2 ... \mathrm{C}_\mathrm{n}\}, \mathrm{C}_1\ \mathrm{C}_2 ... \mathrm{C}_\mathrm{n} \in \mathrm{LofCs};$ $\text{if } \mathrm{C}_\mathrm{i} \in \mathrm{L(LofCs)} \rightarrow \mathrm{C}_\mathrm{i} \in \mathrm{T}$ (LofCs - list of classes) The rule R_{10} accepts the following inputs: class C and a list of classes LofCs. Rule R_{10} is used to add a production in the form $\mathrm{C} ::= \mathrm{C}_1\ \mathrm{C}_2\ ...\ \mathrm{C}_\mathrm{n}$ into the grammar G. Classes from LofCs that are also leafs of the class diagram, are transformed, with the use of rule R_3 to the set of terminals T. The rule is used on the first level. The class Thing is ignored in the transformation. An application example for rule R_{10} can be seen in chapter 3.2, step 4.

This chapter lists some of the rules necessary for the transformation of an onotology into a DSL grammar.

4.2. Ontology to DSL transformation: Home robot example

The prerequisite of the Ontology to DSL transformation (Ontology2DSL) is a proper understanding of the target ontology. The language designer must understand what the ontology describes and why it was designed. Moreover, the language designer needs to know what the DSL requirements are, and what the purpose of the DSL is. In most cases, the DSL requirements and the ontology do not overlap in all concepts. A single ontology, for instance the

HRO, can be used to develop many different DSLs. We continue with the examination of the DSL used for the home robot. The robot is tasked with performing various chores on different locations in the household.

The transformation requires a list of ontology classes and a collection of individually disjoint classes. All the required data was obtained from the OWL document. During the transformation, we also relied on the class hierarchy presented in Fig. 4.

Classes. `Commodity, Food&Drink, Juice, Sweets, Beer, Appliance, TV, VideoCamera, HauseholdEquipment, Door, Window, Task, Close, Bring, Check, SwitchOn, Robot, Location, BedRoom, LivingRoom, Kitchen`.
(Class Thing and other classes from Fig. 4, which are not mentioned in the above list, are ignored in the transformation.)
Disjoint classes.

- `Food&Drink, Appliance and HauseholdEquipment`
- `Juice, Sweets and Beer`
- `TV and VideoCamera`
- `Door and Window`
- `Close, Check, Bring and SwitchOn`
- `Robot, Commodity, Location and Task`
- `BedRoom, LivingRoom and Kitchen`

Step 1. In the first step, all classes are converted into nonterminals.

```
R1(Robot)
N = {Robot}
```

Rule R_1 in this step is used on all the classes and results in the following set of nonterminals N.

```
N = {Commodity, Food&Drink, Juice, Sweets, Beer,
     Appliance, TV, VideoCamera, HauseholdEquipment,
     Door, Window, Task, Close, Bring, Check, SwitchOn,
     Robot, Location, BedRoom, LivingRoom, Kitchen}
```

Step 2. The transformation is continued on the lowest, third, level. It is performed with the rules R_3, R_{5a}, and R_{5c}.

```
R5c(Food&Drink, {Juice, Sweets, Beer}, string)
R3(Juice)
R3(Sweets)
R3(Beer)
R5a(Appliance {TV, VideoCamera})
R3(TV)
R3(VideoCamera)
R5a(HauseholdEquipment {Door, Window})
R3(Door)
R3(Window)
```

```
T = {»juice«, »sweets«, »beer«, »TV«, »videoCamera«,
     »door«, »window«}
P = { Food&Drink ::= Juice | Sweets | Beer | string
      Juice ::= »juice«
      Sweets ::= »sweets«
      Beer ::= »beer«
      Appliance ::= TV | VideoCamera
      TV ::= »TV«
      VideoCamera ::= »videoCamera«
      HauseholdEquipment ::= Door | Window
      Door ::= »door«
      Window ::= »window«}
```

Step 3. The transformation is continued on the second level. The rules used are R_3 and R_{5a}.

```
R5a(Commodity, {Food&Drink, Appliance,
    HouseholdEquipment})
R3(Food&Drink)
R3(Appliance)
R3(HouseholdEquipment)
R5a(Location, {BedRoom, LivingRoom, Kitchen})
R3(BedRoom)
R3(LivingRoom)
R3(Kitchen)
R5a(Task, {Close, Check, Bring, SwitchOn})
R3(Close)
R3(Check)
R3(Bring)
R3(SwitchOn)

T = {…, »food&Drink«, »appliance«, »householdEquipment«,
     »bedRoom«, »livingRoom«, »kitchen«, »close«,
     »check«, »bring«, »switchOn«}
P = { …, Commodity ::= Food&Drink | Appliance |
                    HouseholdEquipment
      Food&Drink ::= »food&Drink«
      Appliance ::= »appliance«
      HouseholdEquipment ::= »householdEquipment«
      Location ::= BedRoom | LivingRoom | Kitchen
      BedRoom ::= »bedRoom«
      LivingRoom ::= »livingRoom«
      Kitchen ::= »kitchen«
      Task ::= Close | Bring | Check | SwitchOn
      Close ::= »close«
      Check ::= »check«
      Bring ::= »bring«
      SwitchOn ::= »switchOn«}
```

Step 4. The first level is transformed with the R_{10} rule.

```
R10(Robot, {Task, Commodity, Location})
T = {…, »Task«, »Commodity«, »Location«}
P = {…, Robot ::= Task Commodity Location}
```

Step 5. In the next step, all possible grammar start symbols are extracted.

```
R6(Commodity, {})
R6(Task, {})
R6(Robot, {})
R6(Location, {})
R5a(S, {Commodity, Task, Robot, Location})
S = {Commodity, Task, Robot, Location }
P = {…, S ::= Commodity | Task | Robot | Location}
```

Step 6. In the next step, rule R_7 is used. Strikethrough production is eliminated from the set of production.

R7(Commodity, +, Commodities)
P = { …, ~~Commodity :: = Food&Drink | Appliance |~~
~~HauseholdEquipment~~
Commodity::= Commodities$^+$
Commodities ::= Food&Drink | Appliance |
HouseholdEquipment}

Step 7. In the last step the syntax is enriched. Strikethrough productions are eliminated from the set of production.

R8(Location, {»from«| »in«}, {}, LocationE)
LocationE:: = {»from«| »in«} Location
P = { …, ~~S ::= Robot | Commodity | Location | Task~~
~~Robot ::= Task Commodity Location~~
S ::= Robot | Commodity | LocationE | Task
Robot ::= Task Commodity LocationE}

Obtained grammar:

```
P = { Robot ::= Task Commodity LocationE

      Task ::= Close | Check | Bring | SwitchOn
      Close ::= »close«
      Check ::= »check«
      Bring ::= »bring«
      SwitchOn ::= »switchOn«

      Commodity::= Commodities+
      Commodities ::= Food&Drink | Appliance |
                      HouseholdEquipment
      Food&Drink ::= Juice | Sweets | Beer | string
      Food&Drink ::= »food&Drink«
      Juice ::= »juice«
      Sweets ::= »sweets«
      Beer ::= »beer«
      Appliance ::= TV | VideoCamera
      Appliance ::= »appliance«
      TV ::= »TV«
      VideoCamera ::= »videoCamera«
      HouseholdEquipment ::= Door| Window
      HouseholdEquipment ::= »householdEquipment«
      Door ::= »door«
      Window ::= »window«

      Location ::= BedRoom | LivingRoom | Kitchen
      BedRoom ::= »bedRoom«
      LivingRoom ::= »livingRoom«
      Kitchen ::= »kitchen«
      LocationE :: = {»from«| »in«} Location}
```

Program examples:

```
close door in bedRoom
check window in kitchen
switchOn TV in livingRoom
bring beer chips from kitchen
```

5. Ontology2DSL

The Ontology2DSL framework enables automated grammar construction as well as one or more programs from a target ontology. The framework accepts an OWL document as an input, parses it and uses the information retrieved to create and fill internal data structures. Then a transformation pattern, annotated with the proper rule execution order, is applied over the data structures and the corresponding grammar and programs are constructed. The resulting grammar, acquired fully automatically, is then inspected by a DSL engineer in order to verify it and find any irregularities. If any irregularities are found, they are tasked with their resolution with regard to the source and type. The engineer can either correct the constructed grammar, programs or the transformation pattern (i.e. change the order in which the rules are applied or construct new rules and include them in the pattern). The framework then rebuilds the grammar and programs as required. The rebuild process can utilize a new transformation pattern on an old ontology, an old pattern on a new ontology, or a new pattern on a new ontology. The process is repeated until the DSL engineer can no longer find any irregularities. The framework also has the option of constructing in sequential steps instead of the fully automated method. In that case, the engineer can execute each rule individually and can, at any time, return to a previous step if the result proves to be unsatisfactory. This method allows for complete control over the grammar and the resulting program's construction process. The final (correct) grammar can later be used by the DSL engineer for the development of DSL tools. The latter are developed with the use of language development tools, such as LISA [29] or VisualLISA [30]. The development of DSL tools from an ontology is a process demonstrated in the workflow of Fig. 5.

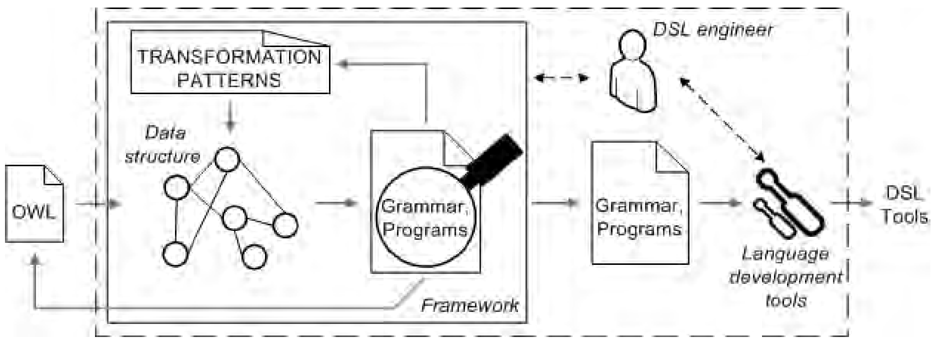


Fig. 5. Ontology2DSL workflow

5.1. Architecture of the framework

The architecture of the Ontology2DSL framework, shown in Fig. 6 is comprised of the following:

– **OWL parser.** The parser is tasked with the parsing of OWL documents and the filling of the data structure with the retrieved data. The data structure is composed of the following individual data structure types: a class tree, an object properties tree, a datatype properties tree, a list of anonymous classes, a list of disjoint classes, a list of instances and a list of ID's of all the ontology building blocks in the aforementioned lists. Part of the data structure for the HRO is presented in Fig. 7. The building of hierarchy objects (trees) and lists is done with a sequential scan of the OWL document. Each retrieved element is added to an appropriate list and is assigned all the necessary information. A check is also performed to determine if the new element possesses any new information that should be assigned to other elements. In instances where the new element has some information that is important for the elements that have not yet been added to any of the trees or lists, that information is cached until the required elements are not added to the data structure.

– **Rule reader.** The reader is tasked with the sequential read operations on the rules list. The reader forwards each rule to the rule execution and transaction logger components.

– **Rule execution component (REC)** is used for the execution of individual rules. The necessary data for the execution is retrieved automatically by REC from the data structure. After a rule is executed, REC refreshes the data structure if the rule execution result requires it. Also, the set of grammar elements are refreshed and parts of the code are written out. The element set of the grammar in the final result becomes the final grammar and the code parts become the programs that represent the final output of the Ontology2DSL framework.

– **Transaction logger.** After the execution of every rule, the system's current state is logged by the transaction logger. The logger stores the entire content of the data structure, the last executed rule, the output of the rule execution component and the current grammar and program parts.

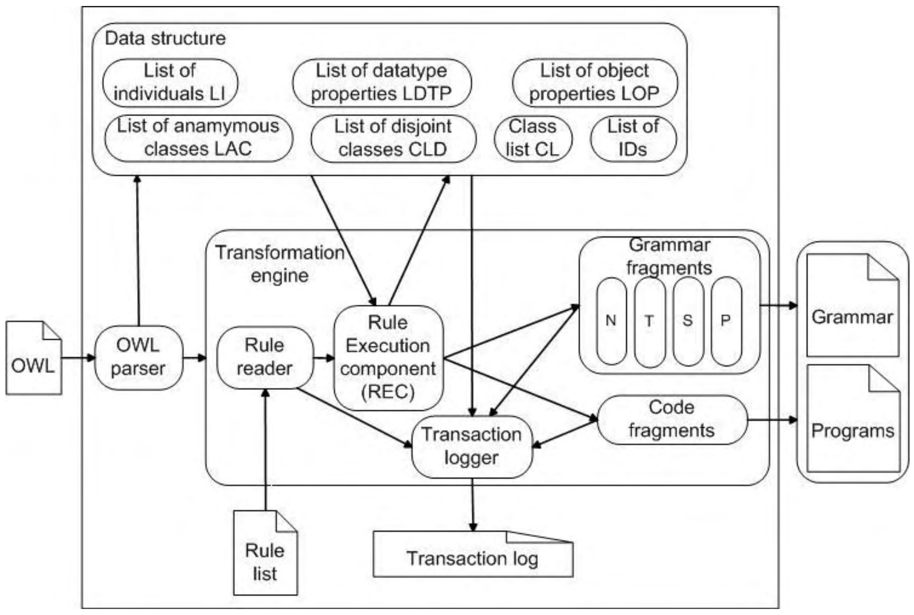
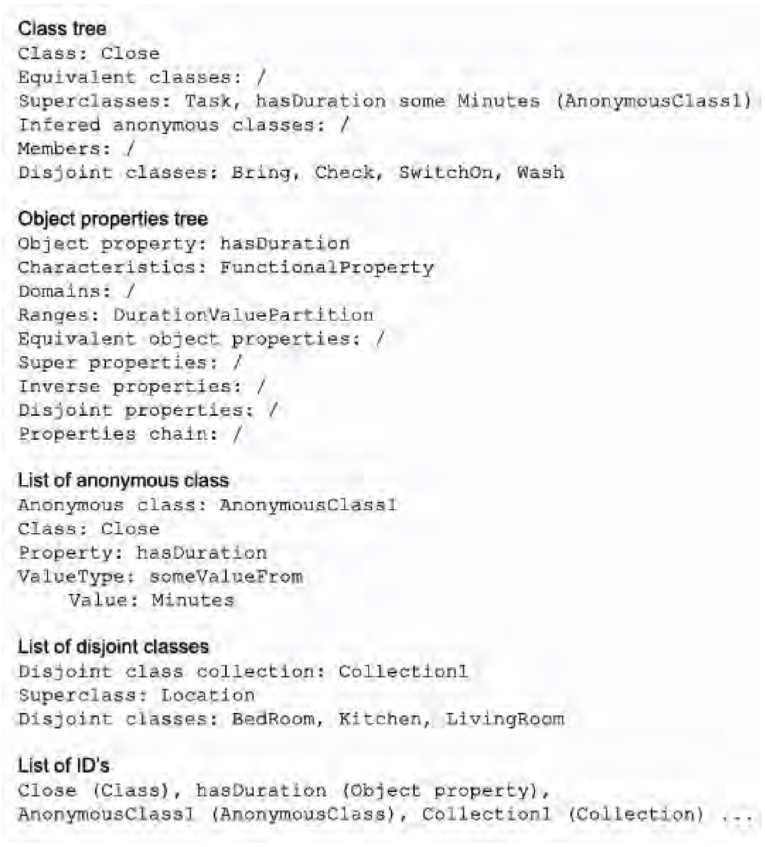


Fig. 6. Architecture of the framework



Class tree
```
Class: Close
Equivalent classes: /
Superclasses: Task, hasDuration some Minutes (AnonymousClass1)
Infered anonymous classes: /
Members: /
Disjoint classes: Bring, Check, SwitchOn, Wash
```

Object properties tree
```
Object property: hasDuration
Characteristics: FunctionalProperty
Domains: /
Ranges: DurationValuePartition
Equivalent object properties: /
Super properties: /
Inverse properties: /
Disjoint properties: /
Properties chain: /
```

List of anonymous class
```
Anonymous class: AnonymousClass1
Class: Close
Property: hasDuration
ValueType: someValueFrom
     Value: Minutes
```

List of disjoint classes
```
Disjoint class collection: Collection1
Superclass: Location
Disjoint classes: BedRoom, Kitchen, LivingRoom
```

List of ID's
```
Close (Class), hasDuration (Object property),
AnonymousClass1 (AnonymousClass), Collection1 (Collection) ...
```

Fig. 7. An excerpt of the data structure for HRO

6. Conclusion and future work

In this paper, we focused on the presentation of a new design methodology that enables the development of a language grammar based on the OBDA. The limitations of the CDA have been examined and a replacement in the form of an OBDA has been proposed. Both analyses have been presented and compared for similarities and differences. Grammar development, based on the OBDA, and the Ontology2DSL framework were also briefly presented.

The results of the comparison between both analyses show that the OBDA is comparable to the CDA and also provides some additional information that can be used to specify language behavior. As such, it is also suitable as an alternative to CDA for grammar development. The framework Ontology2DSL is still under development. The current version is composed of all of the basic components: an OWL parser, a rule reader, REC and a transaction logger. As opposed to other components that are fully developed, REC is not fully developed, as it does not yet construct code fragments. The framework in the current development phase can only be used to construct grammar. Additionally, in the current version, a DSL engineer cannot add custom rules and create custom transformation patterns. In the future, we intend to fully develop the Ontology2DSL framework. We will also focus on validating the developed grammar and the use of previously unused information (i.e. for semantics development) that was acquired with an OBDA. The results of our research work will also include the transformation of the developed DSL to a form that is compatible with compiler generators, such as LISA [29] or VisualLISA [30]. Our future work also encompasses empirical studies to evaluate the success of our methodology and to compare it with the existing methodologies. One of our future activities, to complete the methodology Ontology2DSL, will be an evaluation of DSLs. As shown in study [20], this activity is often underestimated by language developers. There is a plan to support this activity with a tool based on a questionnaire similar to [31] which will further improve the language.

References

1. Mernik, M., Heering, J., Sloane, A. M.: When and how to develop domain-specific languages. ACM Computing Surveys, Vol. 37, No. 4, 316-344. (2005)
2. Kosar, T., Oliveira, N., Mernik, M., Veranda Pereira, M. J., Črepinšek, M., da Cruz, D., Henriques, P. R.: Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. Computer Science and Information Systems, Vol. 7, No. 2, 247-264. (2010)
3. Thibault, S., Marlet, R., Consel, C.: Domain-Specific Languages: From Design to Omplementation Application to Video Device Drivers Generation. Conception, Implementation and Application. IEEE Transactions on Software Engineering, Vol. 25, No. 3, 363-377. (1999)
4. Thibault, S.: Domain-Specific Languages: Conception, Implementation and Application. Phd thesis. Université de Rennes, France. (1998)
5. Kosar T., Martínez López P.E., Barrientos P.A., Mernik M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology, Vol. 50, No. 5, 390-405. (2008)
6. Mernik, M., Hrnčič, D., Bryant, B. R., Javed, F.: Applications of grammatical inference in software engineering : domain specific language development. In: Martin-Vide, C. (ed.): Scientific applications of language methods, Vol. 2. Imperial College Press, London, 421-457. (2011)
7. Taylor, R. N., Tracz, W., Coglianese, L.: Software development using domain-specific software architectures. ACM SIGSOFT Software Engineering Notes, Vol. 20, No. 5, 27-38. (1995)
8. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA). Technical report. (1990)
9. Simons, M., Anthony, J.: Weaving the Model Web: A Multi-Modeling Approach to Concepts and Features in Domain Engineering. In Proceedings of the 5th International Conference on Software Reuse. IEEE Computer Society, Victoria, BC, Canada, 94-102. (1998)

10. Lacy, L.: Representing Information Using the Web Ontology Language. Trafford Publishing. (2005)
11. Hebeler, J., Fisher, M., Blace, R., Perez-Lopez, A.: A Semantic Web Programming. Wiley Publishing. (2009)
12. Studer, R., Benjamins, R., Fensel, D.: Knowledge engineering: Principles and methods. Data & Knowledge engineering, Vol. 25, No. 1-2, 161-198. (1998)
13. Tairas, R., Mernik, M., Gray, J.: Using Ontologies in the Domain Analysis of Domain-Specific Languages. In: Chaudron, M. R. V (ed.): Models in Software Engineering. Lecture Notes in Computer Science, Vol. 5421. Springer-Verlag, Berlin Heidelberg New York, 332-342. (2009)
14. Miksa. K., Sabina, P., Kasztelnik, M.: Combining Ontologies with Domain Specific Languages: A Case Study from Network Configuration Software. In: Assmann, U., Bartho, A., Wende, C. (eds.): Reasoning Web. Semantics technologies for software engineering, Vol. 6325. Springer-Verlag, Berlin Heidelberg New York, 99-118. (2010)
15. Guarino, N.: Semantic Matching: Formal ontological distinctions for information organization, extraction, and integration. In: Pazienza, M. T.: Information Extraction: A Multidisciplinary Approach to an Emerging Information Technology. Lecture Notes in Computer Science, Vol. 1299. Springer-Verlag, Berlin Heidelberg New York, 139-170. (1997)
16. Ontology-Based Evaluation and design of domain-specific visual modeling languages, http://www.loa-cnr.it/Guizzardi/ISD2005.pdf.
17. Walter, T., Parreiras, F. S., Staab, S.: OntoDSL: An Ontology-Based Framework for Domain-Specific Languages. In: Schürr, A., Selic, B. (eds.): Model Driven Engineering Languages and Systems. Lecture Notes in Computer Science, Vol. 5795. Springer-Verlag, Berlin Heidelberg New York, 408-422. (2009)
18. Bräuer, M., Lochmann, H.: An Ontology for Software Models and Its Practical Implications for Semantic Web Reasoning. In: Beckhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.): The Semantic Web: Research and Applications. Lecture Notes in Computer Science, Vol. 5021. Springer-Verlag, Berlin Heidelberg New York, 34-48. (2008)
19. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools and Applications. ACM Press/Addison-Wesley Publishing Co. (2000)
20. Gabriel, P., Goulão, M., Amaral, V.: Do Software Languages Engineers Evaluate their Languages? In Proceedings of the XIII Congreso Iberoamericano en "Software Engineering" (CIbSE'2010). Cuenca, Ecuador, 149-162. (2010)
21. Stabb, S., Studer, R., editors. Handbook on Ontologies. Springer Verlag Berlin Heidelberg. (2009)
22. Corcho, Ó., Gómez-Pérez, A.: A Roadmap to Ontology Specification Languages. In: Dieng, R., Corby, O.: Knowledge Engineering and Knowledge Management. Lecture Notes in Computer Science, Vol. 1937. Springer-Verlag, Berlin Heidelberg New York, 80-96. (2000)
23. Milanović, M., Gašević, D., Giurca, A., Wagner, G., Lukichev, S., Devedžić, V.: Model Transformations to Bridge Concrete and Abstract Syntax of Web Rule Languages. Computer Science and Information Systems, Vol. 6, No. 2, 47-85. (2009)
24. Cho, K., Kawamura, T.: Blogalpha: Home automation robot using ontology in home environment. In Proceedings of the 25th International Multi-Conference Artificial Intelligence and Applications. ACTA Press Anaheim, CA, USA, 197-203. (2007)

25. Van Deursen, A., Klint, P.: Domain-specific Language Design Requires Feature Descriptions. Journal of Computing and Information Technology, Vol. 10, No. 1, 1-17. (2002)
26. Welcome to Protégé. [Online]. Available: http://protege.stanford.edu/ (current April 2011)
27. Jung, H., Park, S.: A Grammar-based Model for the Semantic Web. Computer Science and Information Systems, Vol. 8, No. 1, 73-100. (2011)
28. Aho, A. V., Lam, M. S., Sethi, R., Ullman, J. D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, USA. (2007)
29. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: LISA: An Interactive Environment for Programming Language Development. In: Horspool, R. N. (ed.): Compiler Construction. Lecture Notes in Computer Science, Vol. 2304. Springer-Verlag, Berlin Heidelberg New York, 1-4. (2002)
30. Oliveira, N., Veranda Pereira, M. J., Henriques, P. R., da Cruz, D., Cramer, B.: VisualLISA: A Visual Environment to Develop Attribute Grammars. Computer Science and Information Systems, Vol. 7, No. 2, 247-264. (2010)
31. Haugen, O., Mohagheghi, P.: A Multi-dimensional Framework for Characterizing Domain Specific Languages. In Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07). Montréal, Canada. (2007)

Ines Čeh received the B.Sc. degree in computer science at the University of Maribor, Slovenia in 2008. Her research interests include domain-specific languages and ontologies. She is currently a Ph.D student, employed as a researcher at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Matej Črepinšek received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research interests include grammatical inference, evolutionary computations, object-oriented programming, compilers, grammar-based systems and Android application development. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Tomaž Kosar received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research is mainly concerned with design and implementation of domain-specific languages. Other research interest in computer science include also domain-specific modelling languages, empirical software engineering, software security, generative programming, compiler construction, object oriented programming, object-oriented design, refactoring, and unit testing. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Marjan Mernik received the M.Sc. and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998, respectively. He is currently Professor of Computer Science at the University of Maribor. He is also Visiting Professor of Computer and Information Sciences at the University of Alabama at Birmingham, 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.

Received: December 31, 2010; Accepted: May 13, 2011.

DOI: 10.2298/CSIS101228017O

Domain-Specific Language for Coordination Patterns

Nuno Oliveira[1], Nuno Rodrigues[1,2], and Pedro Rangel Henriques[1]

[1] University of Minho - Department of Computer Science,
Campus de Gualtar, 4715-057, Braga, Portugal
{nunooliveira,prh}@di.uminho.pt
[2] IPCA – Polytechnic Institute of Cavado and Ave
Campus do IPCA, Barcelos, Portugal
nfr@ipca.pt

Abstract. The integration and composition of software systems requires a good architectural design phase to speed up communications between (remote) components. However, during implementation phase, the code to coordinate such components often ends up mixed in the main business code. This leads to maintenance problems, raising the need for, on the one hand, separating the coordination code from the business code, and on the other hand, providing mechanisms for analysis and comprehension of the architectural decisions once made.
In this context our aim is at developing a domain-specific language, `CoordL`, to describe typical coordination patterns. From our point of view, coordination patterns are abstractions, in a graph form, over the composition of coordination statements from the system code. These patterns would allow us to identify, by means of pattern-based graph search strategies, the code responsible for the coordination of the several components in a system. The recovering and separation of the architectural decisions for a better comprehension of the software is the main purpose of this pattern language.

Keywords: coordination patterns, software architectures, domain-specific languages, CoordInspector.

1. Introduction

Software Architecture [17] is a discipline within Software Development [16] concerned with the design of a system. It embodies the definition of the structure and the organisation of components which will be part of the software system. The architecture design also concerns the way these components interact with each other as well as the constraints in their interactions. In their turn, software components [12] may be seen as objects in the object-oriented paradigm, however, besides data and behaviour, they may embody whatever one prefers as a software abstraction. Although they may have their own functionality (sometimes a component is a remote system), most of the times they are developed to be composed with other components within a software system and to be reused from one system to another, giving birth to component-based software engineering methodology [14].

The definition of the interaction between the components of a system may be seen from two perspectives: (i) integration and (ii) coordination. The differences between these two perspectives is slightly none. The former is related with the integration of some functionalities of a system into a second one, which needs to borrow such a computation; the latter is concerned with the low level definition of the communication and its constraints between the components of a system. Such interaction definition between the components can be either *endogeneous* or *exogeneous*. In the latter, the coordination of components is made from the outside of a component, not needing to change its internals to make possible the communication with other components [4], the former is the dual methodology.

This *rule* of separating computational code from coordination code is not always adopted by software developers. The code is often weaved in a single layer where there is no space for separation of these kind of concerns. This behaviour could raise problems in the future of the software system, namely in maintenance phase. These problems are mainly concerned with the comprehension of both the code and the architectural decisions, which hampers their analysis.

Reverse engineering [28] of legacy systems for coordination layer recovery would play an important role on maintenance phases, diminishing the difficulties on analysing the architectural decisions. But, extracting code dedicated to the coordination of the system components from the entire intricate code is not an easy task. There is not a standard (nor unique) way of programming the interactions between the components. However, and fortunately, there is a great number of code patterns, which the majority of the developers use to write coordination operations. Once the code of a system can be represented as a graph of dependencies between the statements and procedures, the so-called *System Dependence Graph* (`SDG`) [15], one is also able to represent code patterns as graphs, allowing the search for these patterns in the `SDG`.

In this context, we define the notion of *coordination patterns* as follows:

> Given a *dependence graph* $\mathcal{G}$, as in [26], a coordination pattern is an equivalence class, a *shape* or a sub-graph of $\mathcal{G}$, corresponding to a trace of coordination policies left in the system code.

In this paper we show how we developed a *Domain-Specific Language* (`DSL`) [8, 21] named `CoordL`, to write coordination patterns. The main objective of this language is to translate `CoordL` specifications into a suitable graph representation. Such representation would feed a graph-based search algorithm, to be applied to a dependence graph, in order to find the coordination code weaved in the system code. In Section 2 we address related work; in Section 3 we present and describe the syntax of `CoordL`; in Section 4 we address its semantics; in Section 5 we show how we used the `AnTLR` system to define the syntax and the semantics of `CoordL`; in Section 6 we expatiate upon actual and future applications of the language and the patterns. Finally, Section 7 presents some conclusions about the presented research.

2. Related Work

`CoordL` is a `DSL` to write coordination patterns with the purpose of extracting and separating the coordination layer from the source code of a component-based software system. The main domain of this work is the reverse engineering (of legacy software systems architecture) and the idea of the code separation between concern-oriented layers aims to recover architectural decisions and ease the comprehension of the entire system and its architecture which is one of the most important parts within the maintenance phase of software engineering.

The recovery of the system architecture for software comprehension is not a novelty. Tools like Alborz [27] or Bauhaus [24] recover the blueprints of an object-oriented system. Bauhaus recovers architectures as a graph where the nodes may be types, routines, files or components of the system, and the edges model the relations between these nodes. Such architectural details are presented in different views for an easy understanding of the global architecture. Alborz presents the architecture as a graph of components and keeps a relation between this graph and the source code of the software system. Our main aim is not at visualising the blueprints of a system, but to provide mechanisms for understanding the rationale behind the architectural decisions once made. This embodies the recovering of the coordination code. Although visualization is really important for architectural analysis, the tools mentioned before, and even more recent and focused tools like [19] do not support the mentioned feature and do not take advantage of code patterns to do the job.

Although we may reference Architectural Patterns [6] or Design Patterns [9] as related work, because of the common methodology of patterns and the borrowed notions and description topics, there is a huge difference between their application. While coordination patterns are used to lead a reverse engineering to recover architectural decisions, and are focused on low-level compositions of code, the architectural/design patterns work at a higher level, being used to define the architecture of a system in earlier phases of the software development process [16]. Patterns and patterns finding are a very interesting matter on data mining discipline [5], and are being applied on several areas, with focus on social networks [18, 13]. Although the work reported in this paper is far from the area of data mining and social networks, the purpose of `CoordL` is to extract useful information based on patterns that are a result of previous knowledge on how coordination of components is made. The recovering algorithms that we may use may be based on those used to mine data on social networks since they also rely on graph-based search. The applications of the recovering strategies on data mining is specially concerned with optimization and adaptability to new contexts. By recovering the architectural decisions, also optimizations and adaptability (on a different perspective than that of data mining) may be also a valuable application of the subject in this paper. In fact there is, somehow, a parallel between data mining and our work that we may follow, but the essence of both kind of works is very different. The same does not happen with process mining [7], which, in fact, is very close to our work. The search for business process workflows [1] makes heavy use of process mining techniques. Although process mining is typically based on event logs, a dependence graph may also uncover traces of a workflow. A workflow between components or modules is what, at the end, we want to obtain for the coordination code separation.

Architecture Description Language's (ADL), are languages used to formally describe a system components' and the interactions between them. Although CoordL is not to be considered an ADL, we must acknowledge that there are some similarities in the concepts embodied in these languages and those encapsulated in our. Some well-known examples of these languages are ACME [11], ArchJava [2], Wright [3], and Rapide [20]. The great majority of these languages has tool support for analysing the described architecture. Such analysis, made at high level, allows one to reason about the correctness of the system, and may provide important information about future improvements that can, or can not, be done according to the actual state of the architecture.

According to our knowledge, there is no other language with the same specific purpose as CoordL.

3. CoordL - Design and Syntax

The design of a DSL is always a task embodying some well defined steps. As a first step, one needs to collect all the information about the domain in which the language will actuate. Afterwards, this information must be properly organised using, for instance, ontologies [29]. Once the main concepts of the domain are identified, one needs to choose those that are really needed to be encapsulated in the syntax of the language; this leads to the last step which concerns the choice of a suitable syntax for the language.

Figure 1 presents an ontology to organise the domain knowledge of the area where we want to actuate. The main concept of this domain is the coordination pattern. The majority of the concepts incorporated in this domain description are wider than what we show, however, to keep the description limited to the domain, we narrowed the possible relations between each concept, as well as the examples they may have.

Note that in this ontology we use operational relations (marked as dashed arrows) besides the normal compositional ones. This provides a deeper comprehension of how the concepts interact between each other in the domain.

The core of the knowledge base represented in Figure 1, describes that a coordination pattern is a part of a *coordination dependence graph* (CDG) [26], abstracting code which is seen as a composition of statements concerned with coordination aspects, and are used to analyse architectures. As a novelty, the web of knowledge shows that coordination patterns communicate with each other through ports.

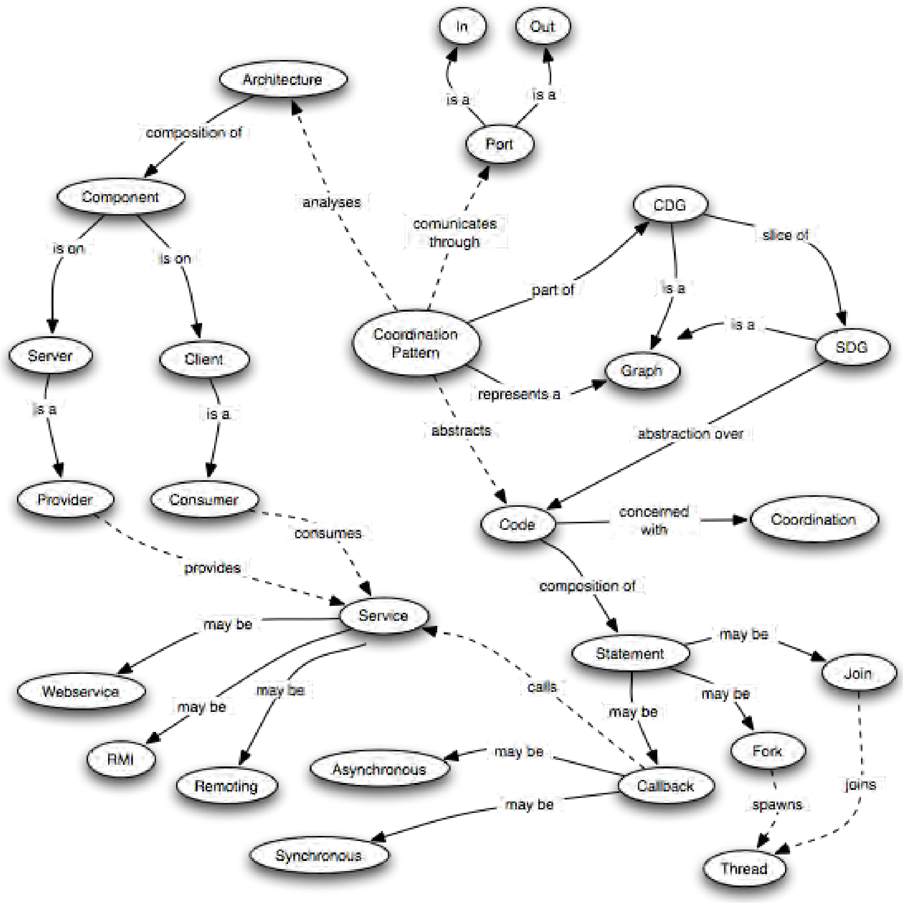


Fig. 1. Ontology Describing the Coordination Pattern Domain Knowledge

From this description, and knowing that the main objective of `CoordL` is to define a graph over the composition of statements in the source code of a system, one needs some kind of graph representation to be embodied in the language. An obvious reference for representing graphs in a textual form is the DOT language [10], so, `CoordL` borrows some aspects from that language. The notion of communication ports (in and out) came from the `ACME` language [11], although the notion of *ports* is very different in these two contexts. To know which ports exist in a pattern, the notion of *arguments* – taken from any *general-purpose programming language* (`GPL`) – was adopted . The description of what are these ports led to the introduction of declarations and initialisations in the language. Declarations describe the types of statements represented as nodes in the graph, while initialisations describes the service call which is performed by the node.

From this textual description we defined a syntax by means of a context free grammar (partially) shown in Listing 1.1.

Listing 1.1. Partial grammar for `CoordL`

```
lang → pattern+
pattern → ID '(' ports '|' ports ')' '{' decls root graph '}'
ports → lstID
decls → (decl ';')+
decl → 'node' lstID '=' nodeRules | 'fork' lstID | 'join' lstID |
  'ttrigger' lstID | ID instances
instances → instance (',' instance)*
instance → ID '(' ports '|' ports ')'
...
root → 'root' ID ';'
graph → aggregation | connections
aggregation → patt_ref ('+' patt_ref)*
patt_ref → cnode | '(' aggregationn ')' connection
...
cnode → node | ID '.' propTT
...
connection → '{' operations '}' '@' '[' ports_alive '|' ports_alive ']'
...
operation → cnode link cnode | fork | join | ttrigger
...
fork → node sp_link '{' cnode ',' cnode '}'
...
link → '-' ID '-' '>' | '-' '(' ID ',' ID ')' '-' '>'
...
```

Figure 2 presents two examples of patterns written with `CoordL`. Pattern a), known as the *Asynchronous Sequential Pattern*, is a pattern often used when the system has to invoke a series of services but the order of the answer is not important. Pattern b), known as the *Joined Asynchronous Sequential Pattern*, is a transformation of the first pattern to impose order in the responses.

Both of these patterns address different aspects of the syntax, but the main structure of the patterns is the same. Moreover, they address the composition and reuse of patterns.

Regard, for instance, the pattern in Figure 2.a). It has a unique identifier (`pattern_1`) and declares *in* and *out* ports, identified by p_0 and p_1, p_2 and p_3 respectively. The *in* ports go on the left side of the '|' (bar) symbol, and the *out* ports on the right. Then, a space is reserved for node declarations and initialisations. There are 5 types of nodes in `CoordL`, namely *node*, *fork*, *join*, *ttrigger* and *pattern instance*. In Figure 2.a) we use the node and fork types, and in Figure 2.b) we use node, join and pattern instance types. The ttrigger type is similar to fork and join.

Nodes of type *node* require an initialisation, describing a list of rules addressing the corresponding coordination code fragment, the type of interaction and the calling discipline. These rules are composed using the $\&\&$ (and) and/or || (or) logical operators, and the list must, at least, embody one of the following: (i) Statement (st), presents the code fragment of the statement responsible by the coordination request. This statement may be described by a regular expression or may be a complete sentence; (ii) Call Type (ct), defines the type of service requested. The options are not limited, but some of the most used are web services, RMI or .Net Remoting; (iii) Call Method (cm), defines the method in which the request is made. It can be either synchronous or asynchronous, and (iv) Call Role (cr), describes the role of the component that is requesting the service. It can be either consumer or producer.

```
1  pattern_1 (p0 | p1, p2, p3){
2    node p0, p3 { st == "*" }
3    node p1, p2 {
4      st == "calling(*)" &&
5      ct == webservice &&
6      cm == sync &&
7      cr == consumer
8    };
9    fork f1, f2;
10   root p0;
11
12   {f2 -(x,w)-> (p3, p2)}
13   {f1 -(x,y)-> (f2, p1)}
14   {p0 -x-> f1}
15 }
```

(a)

```
pattern_2 (p1 | p2){                          1
  node p1, p2, pa = {st == "*"};              2
  pattern_1 patt(i1 | o1, o2, o3);            3
  join j1, j2;                                4
  root p1;                                    5
                                              6
  (p1 + patt + p2)                            7
  {p1 -x-> patt(i1),                          8
  (patt(o1), patt(o3)) -(x,y)-> j1}           9
  {( j1, patt(o2)) -(x,w)-> j2}               10
  {j2 -x-> p2}                                11
}                                             12
```

(b)

Fig. 2. Definition of Two Coordination Patterns with `CoordL`

Pattern instance nodes have the type of an existent pattern. In Figure 2.b), line 3, it is declared an instance of pattern `pattern_1`. Each instance of a pattern must be initialised with unique identifiers referring to all the `in` and `out` ports of the pattern typing it.

In `CoordL`, we define patterns by giving them a name, defining the ports and their body. However, nodes and other anonymous structures used (within the body) to define a pattern are also seen as patterns (or *pseudo-patterns* for disambiguation purposes). The main operations over patterns (including *pseudo-patterns*) are the aggregation and the connection. Aggregation[3] is the

[3] Aggregation may be used alone in a pattern body definition, but will never define a usable pattern.

combination of two or more patterns by putting them *side-by-side*, this is, not making any connection between their ports. The syntax for the aggregation operation is presented at line 7 of Figure 2.b). Connection is the combination of two nodes by means of an edge with the identification of, at least, a running thread. Examples may be seen in lines 12, 13 and 14, of `pattern_1` and 8, 9, 10 and 11 of `pattern_2`.

These two operations are used to build the pattern graph, which comes after all node declarations and identification of the root node[4]. There are two ways of defining the graph: (i) the implicit composition, where there are only connection operations and (ii) the explicit composition, where aggregation and connection operations are used simultaneously. The graph of `pattern_1` uses implicit composition, while `pattern_2` uses explicit composition.

The connection operation uses one or more `out` nodes and one or more `in` nodes (depending on the type of `in` and `out` nodes). When the connection uses these nodes, their implicit `in` or `out` ports are closed, meaning that no newer connection can use these nodes as `in` or `out` ports again. Sometimes one needs to reuse a node as an `in` or an `out` port of a connection. This leads to the re-opening of a port to be used in the sequent connections. In order to facilitate this, we introduce the '@' (alive) operator.

We acknowledge that with all the operators and the associated syntax, the pattern code is not easily readable. This way, we define a visual notation with a suitable "translation" from the textual notation of `CoordL`. In Figure 3 we present the components of the visual notation, corresponding to the textual elements that define the graph.

In Figure 4 we present how the patterns in Figure 2 look like in this notation.

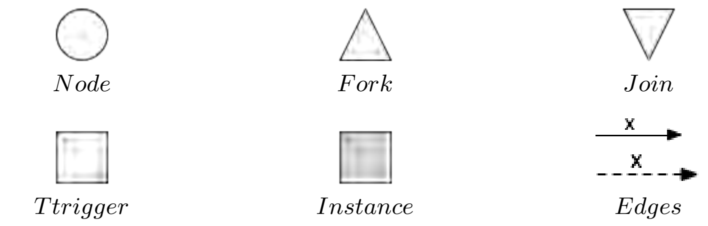


Fig. 3. Components of the Visual Notation for `CoordL`

[4] The root node identifies the start node of the pattern and is only useful for graph-based search of these patterns in a `CDG`. It must be one of the `in` ports of the pattern, chosen nondeterministically, by the pattern definer.

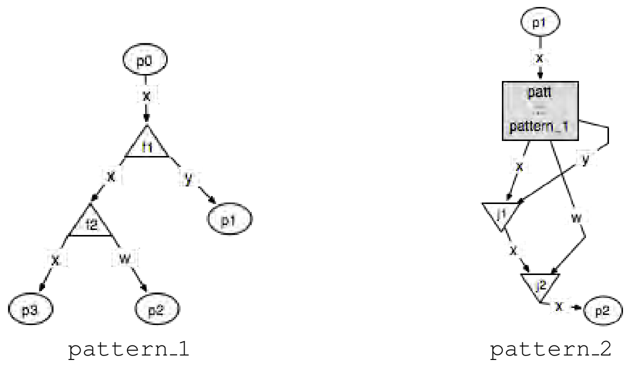


Fig. 4. Visual Representation of Two Coordination Patterns

4. CoordL - The Semantics

The constructs presented in Section 3 have a precise meaning in `CoordL`. In some cases it is possible to draw a mapping between the meaning of a construct and the dependence graph, which is extracted from the source code of the system being analysed. The following paragraphs provide an informal semantics of each construct in the language.

Bar: $|$

This construct separates a list of identifiers into two. The identifiers on the left side list are called `in` ports and those on the list at the right side are called `out` ports. It may appear in the *signature* of a pattern, or in the graph of a pattern, whenever it is needed to keep ports opened for further use.

Aggregation: $pp_1 + pp_2$

This construct sets two patterns side by side, but it doesn't connect them. This is used to reinforce the existence of the patterns in the graph, before connecting their ports. The aggregation operation (meaning collecting patterns, in our standpoint) is not required to build a graph, however, for completeness of the language and for a calculus of `CoordL` language (as envisaged as future work), this would be an important operation.

Connection: n_1 –x–> n_2

This construct creates a link between two nodes in the graph of the pattern. It means that in the dependence graph $\mathcal{G}$, where the pattern will be applied, there is a path of one or more edges going from n_1 to n_2 through one or more edges in a thread identified by x.

Fork Connection: f –(x, y)–> (n_1, n_2)

This construct creates a link between three nodes in the graph of the pattern, where the start node is a fork. It means that in the dependence graph $\mathcal{G}$, where the pattern will be applied, there are two parallel paths (p_1 and p_2) going from f to n_1 through one or more edges in a thread identified by x, and from f to n_2 in a freshly spawned thread identified by y, respectively. A necessary pre-condition

is that in the dependence graph, there is some path p_0 from any node to f in a thread identified by x.
Join Connection: (n_1, n_2) –(x, y)–> j
This construct creates a link between three nodes in the graph of the pattern, where the end node is a join. It means that in the dependence graph $\mathcal{G}$, where the pattern will be applied, there are two parallel paths (p_1 and p_2) going from n_1 to j through one or more edges in a thread identified by x, and from n_2 to j in a thread identified by y, respectively. A necessary pre-condition is that in the dependence graph, there are two paths (p_0 and p'_0) from a fork node to n_1 in a thread identified by x and from the same fork node to n_2 in a thread identified by y, respectively.
Thread Trigger Connection: (n_1, n_2) –(x, y)–> $tt.sync$, (n_1, n_2) –(x, y)–> $tt.fail$
This construct creates a link between three o nodes in the graph of the pattern, where the end node is a ttrigger. It means that in the dependence graph $\mathcal{G}$, where the pattern will be applied, there are two parallel paths (p_1 and p_2) going from n_1 to tt through one or more edges in a thread identified by x, and from n_2 to tt in a thread identified by y, respectively. This meaning aims at expressing what happens when the threads synchronise (tt.sync), or when the threads synchronisation fails (tt.fail). A necessary pre-condition is that in the dependence graph there are two paths (p_0 and p'_0) from a fork node to n_1 in a thread identified by x and from the same fork node to n_2 in a thread identified by y, respectively.
List of Connections: $\{\ connection, \ldots\ \}$
This construct creates a list of independent connections. That is, a connection inside this list does not depend on any node, node property or even on other connections that are used and defined in the list. This independence resorts to the fact that there is no order between the connections inside a list of connections. Subsequent lists of connections may, but are not obliged to, depend on previous lists.

Along with this construct comes the notion of *fresh nodes*. A fresh node is a control node (like a fork, join or ttrigger) that is firstly used in a connection, and cannot be reused in the same list because of the dependence order. For instance, a fork node must be used as an out port in a connection before being used as a `in` node.
Alive: @
This construct instructs that a list of identifiers is kept alive as `in` and `out` ports. Ports need to be reopened because once a connection uses a node, the implicit port of such node is *killed*. The '@' construct is followed by a list of identifiers divided into two by the bar construct.

5. CoordL - Compiling & Transforming

We used `AnTLR` system [23] to produce an attribute-grammar-based parser for `CoordL`. Taking advantage of `AnTLR` features we adopted a *separation of concerns* method to generate the full-featured compiler. Figure 5 shows the architecture of the compiler system. The main piece of the compiler system is the syntax module where we specify both the concrete and abstract syntax for `CoordL`, using the context free grammar presented in Listing 1.1. Based on the abstract syntax, `AnTLR` produces an intermediate structure of that grammar known as a tree-grammar.

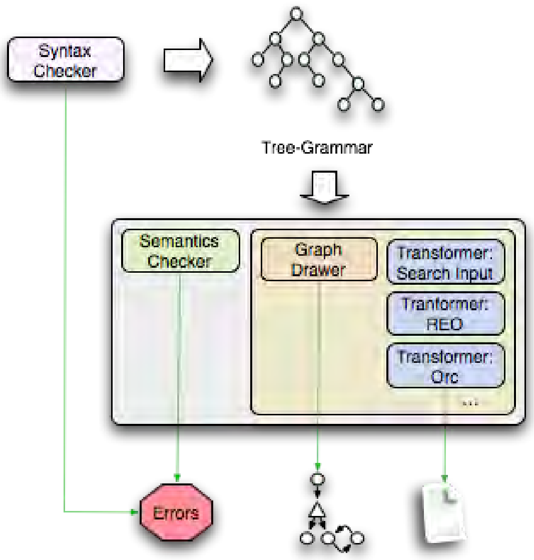


Fig. 5. `CoordL` Compiler Architecture

From the tree-grammar (using attribute grammars methodology) we were able to define new modules that do not care about the concrete syntax. These modules embody the semantics checker, the graph drawer and the unimaginable number of possible transformations applied to that tree-grammar.

The following hierarchical dependence on these modules is observed: the semantics module depends on the syntax module; the graph drawer and the transformation modules depend on the semantics module, so, by transitivity, they also depend on the syntax module. This holds the requirement that some modules may only be used if the syntax and the semantics of the `CoordL` sentence are correct.

We recognise that the separation of concerns in the modules and the dependence between them may be seen as a problem in maintaining the compiler. For instance, if something in the abstract syntax of the language changes, these changes must be performed in every dependent module. Nevertheless, this method also brings positive aspects: (i) the number of code lines in each file decreases, easing the comprehension of the module for maintenance; (ii) since each module defines an operation over the coordination patterns' code, the compiler may be integrated in a software system providing independent features to manipulate the patterns and (iii) the separation of concerns into modules eases the maintenance of each feature.

The transformation modules have, as main objective, to provide perspectives about the coordination patterns, namely, their transformation into Orc [22] or REO [4] specifications. The transformation of the patterns in Orc is more or less simple since it may be used an algorithm similar to that presented in [26] adapted to `CoordL` (since it is originally adapted to the `CDG`). Concerning REO, adequacy of transforming these patterns into REO circuits cares for deep research, since the paradigms are, somehow, different.

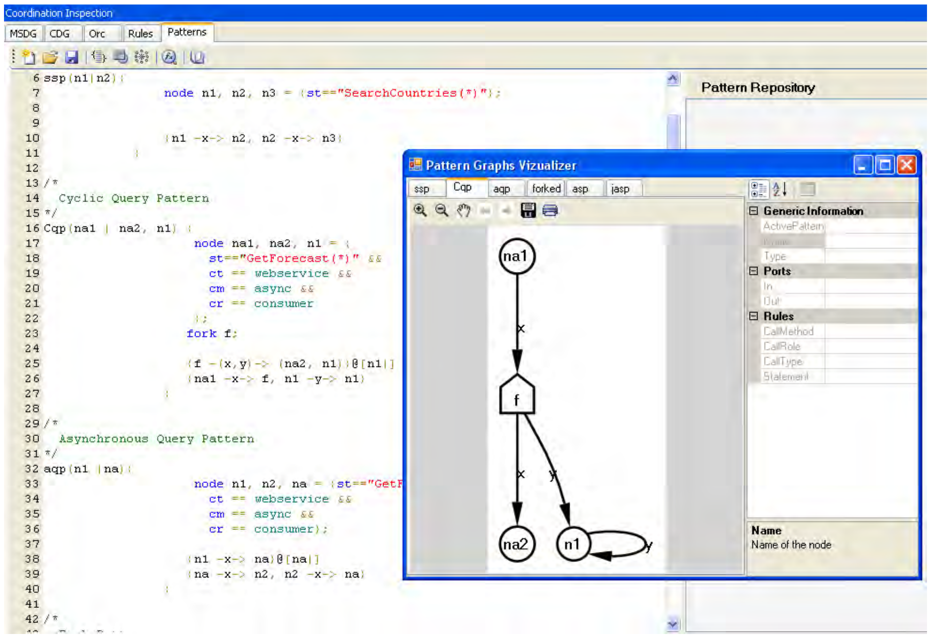
An important module to be considered is the transformation of the pattern code into a suitable input to search for these patterns in the dependence graph of a system's code. As for the syntax and the semantic modules, their main output is the syntactic and semantic errors, respectively. The graph drawer module outputs the visual representation of the coordination patterns.

The transformation of the patterns in their visual notation is the most direct and easy transformation from the mentioned ones. Technically, it was defined a new module using the tree-grammar for `CoordL` that is created by `AnTLR`. The main idea of the transformation is to define visual representation of all (complete) patterns sent as input of that module. This way, the module receives a string with `CoordL` patterns and outputs a list of visual representations. Then, on the interesting parts of the `CoordL` abstract grammar, we introduce blocks of code that define the visual representation of each single pattern. To be more precise, the graph of a pattern is constructed, in several productions of the grammar, using C$\#$ objects that are synthesized as attributes of the grammar. Later, these objects are encapsulated in a *Graph* object and set into a slot of the output list. Each of these *Graph* objects will be processed (by the GLEE/MSAGL library mechanisms) in order to produce the visual representation of the pattern. An example of a visual representation can be viewed on Figure 6. Although with different objectives, the main process for all the other modules is similar to this one. In fact, this one is imposed by the attribute grammar methodology, and is very efficient and intuitive and with good support on `AnTLR`.

6. Applications and Further Work

Being a DSL, the range of possible applications of `CoordL` is very narrow. Its precise objective of matching coordination traces in a dependence graph reduces its applicability to other areas. Nevertheless, the area of architectural analysis and comprehension allows a deep application of this language.

`CoordInspector` [25] is a tool to extract the coordination layer of a system and to represent it in suitable visual ways. In a fast overview, `CoordInspector` processes *common intermediate language* (`CIL`), meaning that systems written in more than 40 `.NET` compliant languages can be processed by the tool. The tool works by transforming `CIL` code into an `SDG` which is sliced to produce a `CDG`. The tool then uses *ad-hoc* graph notations and rules to perform a blind search for non-formalised patterns in the CDG. Here is where CoordL has its relevance. Due to its systematisation and robust formal semantics, the process of matching patterns in the code can be more reliable than using the *ad-hoc* rules. The integration of CoordL in CoordInspector led to the development of an editor to deal with the language. Figure 6 presents an overview of the editor integrated in CoordInspector. The editor makes heavy use of the CoordL compiler system, namely the syntax and semantics modules in order to check whether there are or there are not errors in the patterns' specification, and also performs transformations of the patterns into their visual representation.



```
ssp(n1|n2){
                node n1, n2, n3 = {st=="SearchCountries(*)"};


                {n1 -x-> n2, n2 -x-> n3}
            }

/*
  Cyclic Query Pattern
*/
Cqp(na1 | na2, n1) {
                    node na1, na2, n1 = {
                      st=="GetForecast(*)" &&
                      ct == webservice &&
                      cm == async &&
                      cr == consumer
                    };
                   fork f;

                   {f -(x,y)-> (na2, n1)}@[n1|]
                   {na1 -x-> f, n1 -y-> n1}
                }

/*
  Asynchronous Query Pattern
*/
aqp(n1 |na){
                    node n1, n2, na = {st=="GetF
                      ct == webservice &&
                      cm == async &&
                      cr == consumer};

                   {n1 -x-> na}@[na|]
                   {na -x-> n2, n2 -x-> na}
                }

/*
```

Name
Name of the node

Fig. 6. CoordInspector with CoordL editor

CoordInspector is used for integration of complex information systems, resorting to the recovering of coordination patterns. The use of CoordL in this task is crucial for a faster and systematised search for such parts of code. Although the graph-based search is not finished yet, there is a contract to follow, as close as possible, the algorithm defined in [26], that is the same used to perform the searching for those *ad-hoc* patterns mentioned before.

In order to avoid the repetition of writing recurrent patterns, we decided to create a repository of coordination patterns. The repository may be accessed by means of web services from the editor in CoordInspector. The repository main objective is to give developers and analysts the possibility of expressing recurrent coordination problems in a `CoordL` pattern and documenting them with valuable information[5]. The existence of the repository of coordination patterns and the fact of being possible the definition of a *calculus* over the language, allows the creation of relations between the patterns, defining an order of patterns.

In what concerns to further work, we believe that the development of a *calculus* over the language would allow the development of a model checker for analysing the properties of these patterns. Also, the application of these patterns to pursuit the work of *van der Aalst* [1] in workflow mining, but applied on a low-level dependence-graph-based search is an interesting perspective for later work. The completion of the graph-based search algorithm and the extraction of code to a newly separated layer is the most important and urgent work to do, in order to finish our approach and start to work in other perspectives.

7. Conclusion

In this paper we introduced a domain-specific language named `CoordL`. This language is used to describe coordination patterns for posterior use in finding and extracting recurrent coordination code compositions in the tangled source code of a software system.

We explained how the language was designed resorting to (i) the application domain description, by means of an ontology, and (ii) existing programming language and associated knowledge. We proceed showing how we took advantage of `AnTLR` to define a full-featured and concern-separated compiler for the language. The adoption of this systematic development of modules for the compiler and the dependencies between them may raise some discussions about the flexibility at maintenance phase. We are aware of such problems, nevertheless we argue that the separation of concerns by modules allows for a better use of the compiler when integrated in other tools, and the problems of maintenance are not that numerous, since the comprehension of the modules is easier due to having a small number of lines of code, and the issue solved in these lines is known *a priori*.

Finally, we argue for the applicability of `CoordL` along with `CoordInspector`[6], a tool to aid in architectural analysis and systems reengineering, and the creation of a pattern repository for (i) cataloguing of valuable information about these coordination patterns and (ii) allowing their adoption reuse by developers and analysts.

References

1. van der Aalst, W., van Dongen, B., Herbst, J., Maruster, L., Schimm, G., Weijters, A.: Workflow mining: A survey of issues and approaches. Data & Knowl-

[5] `http://gamaepl.di.uminho.pt/coordinspector/patternlist.aspx`
[6] `http://gamaepl.di.uminho.pt/coordinspector/`

edge Engineering 47(2), 237–267 (Nov 2003), http://dx.doi.org/10.1016/S0169-023X(03)00066-1

2. Aldrich, J., Chambers, C., Notkin, D.: Archjava: connecting software architecture to implementation. In: ICSE '02: Proceedings of the 24th International Conference on Software Engineering. pp. 187–197. ACM, New York, NY, USA (2002), http://dx.doi.org/10.1145/581339.581365
3. Allen, R.: A Formal Approach to Software Architecture. Ph.D. thesis, Carnegie Mellon, School of Computer Science (January 1997)
4. Arbab, F.: Reo: a channel-based coordination model for component composition. Mathematical. Structures in Comp. Sci. 14(3), 329–366 (June 2004), http://dx.doi.org/10.1017/S0960129504004153
5. Bishop, C.: Pattern Recognition and Machine Learning. Springer, Berlin, 1st edn. (2006)
6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, Chichester, UK (1996)
7. Cook, J.E., Wolf, A.L.: Discovering models of software processes from event-based data. ACM Trans. Softw. Eng. Methodol. 7(3), 215–249 (Jul 1998), http://dx.doi.org/10.1145/287000.287001
8. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices 35, 26–36 (2000), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.33.8207
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional (1995)
10. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Softw. Pract. Exper. 30(11), 1203–1233 (2000)
11. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press (2000)
12. Goguen, A.J.: Reusing and interconnecting software components. Computer 19(2), 16–28 (1986), http://dx.doi.org/10.1109/MC.1986.1663146
13. Goldberg, M., Hayvanovych, M., Hoonlor, A., Kelley, S., Magdon-Ismail, M., Mertsalov, K., Szymanski, B., Wallace, W.: Discovery, analysis and monitoring of hidden social networks and their evolution. In: IEEE Conference on Technologies for Homeland Security (2008). pp. 1–6 (Oct 2008), http://dx.doi.org/10.1109/THS.2008.4637294
14. Heineman, G.T., Councill, W.T. (eds.): Component-based software engineering: putting the pieces together. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
15. Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation. vol. 23, pp. 35–46. ACM, New York, NY, USA (July 1988), http://dx.doi.org/10.1145/53990.53994
16. Jacobson, I., Booch, G., Rumbaugh, J.: The unified software development process. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999), http://portal.acm.org/citation.cfm?id=309683
17. Jen, L.r., Lee, Y.j.: IEEE Recommended Practice for Architectural Description of Software-intensive Systems. IEEE Architecture pp. 1471–2000 (2000), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.102.9904

18. Lauw, H., Lim, E.P., Pang, H., Tan, T.T.: Social network discovery by mining Spatio-Temporal events. Computational &#38; Mathematical Organization Theory 11(2), 97–118 (Jul 2005), http://dx.doi.org/10.1007/s10588-005-3939-9
19. Lee, L., Kruchten, P.: A Tool to Visualize Architectural Design Decisions. In: Becker, S., Plasil, F., Reussner, R. (eds.) Quality of Software Architectures. Models and Architectures, Lecture Notes in Computer Science, vol. 5281, chap. 3, pp. 43–54. Springer Berlin / Heidelberg, Berlin, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-87879-7_3
20. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using rapide. IEEE Trans. Softw. Eng. 21(4), 336–355 (1995), http://dx.doi.org/10.1109/32.385971
21. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4), 316–344 (December 2005), http://dx.doi.org/10.1145/1118890.1118892
22. Misra, Jayadev, Cook, William: Computation orchestration: A basis for wide-area computing. Software and Systems Modeling (SoSyM) 6(1), 83–110 (March 2007), http://dx.doi.org/10.1007/s10270-006-0012-1
23. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf, Raleigh (2007), http://www.amazon.de/Complete-ANTLR-Reference-Guide-Domain-specific/dp/0978739256
24. Raza, A., Vogel, G., Plödereder, E.: Bauhaus - a tool suite for program analysis and reverse engineering. In: Reliable Software Technologies - Ada-Europe 2006, pp. 71–82. LNCS (4006) (June 2006), http://dx.doi.org/10.1007/11767077_6
25. Rodrigues, N.: Slicing Techniques Applied to Architectural Analysis of Legacy Software. Ph.D. thesis, Engineering School, University of Minho (October 2008)
26. Rodrigues, N.F., Barbosa, L.S.: Slicing for architectural analysis. Science of Computer Programming (March 2010), http://dx.doi.org/10.1016/j.scico.2010.02.002
27. Sartipi, K., Ye, L., Safyallah, H.: Alborz: An interactive toolkit to extract static and dynamic views of a software system. In: ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension. pp. 256–259. IEEE Computer Society, Washington, DC, USA (2006), http://dx.doi.org/10.1109/ICPC.2006.8
28. Storey, M.A.: Theories, tools and research methods in program comprehension: past, present and future. Software Quality Journal 14(3), 187–208 (September 2006), http://dx.doi.org/10.1007/s11219-006-9216-4
29. Tairas, R., Mernik, M., Gray, J.: Using ontologies in the domain analysis of domain-specific languages. Models in Software Engineering pp. 332–342 (2009), http://dx.doi.org/10.1007/978-3-642-01648-6_35

Nuno Oliveira received, from University of Minho, a degree in Computer Science (2007) and a M.Sc. in Informatics (2009), for his thesis "Improving Program Comprehension Tools for Domain Specific Languages". He is a member of the Language Processing group at CCTC (Computer Science and Technology Center) , University of Minho. He participated in several projects with focus on Visual Languages and Program Comprehension. Currently, he is starting his PhD studies on Architectural Reconfiguration of Interacting Services, under a research grant funded by FCT.

Nuno F. Rodrigues got a degree in "Mathematics and Computer Science", at University of Minho, and finished a Ph.D. thesis in "Software Architectures" also at University of Minho. Currently he is an Assistant Professor at the Polytechnic Institute of Cavado and Ave, where he is also the director of the Digital Games Development Degree and head of the Digital Games Research Centre.

Pedro Rangel Henriques got a degree in "Electrotechnical/Electronics Engineering", at FEUP (Porto University), and finished a Ph.D. thesis in "Formal Languages and Attribute Grammars" at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the "Language Processing group" at CCTC (Computer Science and Technologies Center). He teaches many different courses under the broader area of programming: Programming Languages and Paradigms; Compilers, Grammar Engineering and Software Analysis and Transformation; etc. Pedro Rangel Henriques has supervised Ph.D. (11), and M.Sc. (13) thesis, and more than 50 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; code analysis, program visulaization/animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He is co-author of the "XML & XSL: da teoria a prática" book, published by FCA in 2002; and has published 3 chapters in books, and 20 journal papers.

Received: December 28, 2010; Accepted: May 10, 2011.

DOI:10.2298/CSIS101231009K

From DCOM Interfaces to Domain-Specific Modeling Language: A Case Study on the Sequencer

Tomaž Kos[1], Tomaž Kosar[2], Jure Knez[1], and Marjan Mernik[2]

1 DEWESoft d.o.o., Gabrsko 11a, 1420 Trbovlje, Slovenia
{tomaz.kos, jure.knez}@dewesoft.si
2 University of Maribor, Faculty of Electrical Engineering and Computer Sciences, Smetanova ulica 17, 2000 Maribor, Slovenia
{tomaz.kosar, marjan.mernik}@uni-mb.si

Abstract. Software development is a demanding process, since it involves different parties to perform a desired task. The same case applies to the development of measurement systems – measurement system producers often provide interfaces to their products, after which the customers' programming engineers use them to build software according to the instructions and requirements of domain experts from the field of data acquisition. Until recently, the customers of the measurement system DEWESoft were building measuring applications, using prefabricated DCOM objects. However, a significant amount of interaction between customers' programming engineers and measurement system producers is necessary to use DCOM objects. Therefore, a domain-specific modeling language has been developed to enable domain experts to program or model their own measurement procedures without interacting with programming engineers. In this paper, experiences gained during the shift from using the DEWESoft product as a programming library to domain-specific modeling language are provided together with the details of a Sequencer, a domain-specific modeling language for the construction of measurement procedures.

Keywords: domain-specific modeling languages, data acquisition, measurement systems.

1. Introduction

Data acquisition is the process of capturing and measuring physical data and the conversion of these results into digital form that is further manipulated by a computer program. Data acquisition systems, also called measurement systems, are used in various fields, ranging from the automotive industry to the aircraft industry, the space industry and electrical engineering. For instance, Fig.1 shows data acquisition during a flight test with the DEWESoft product. The measurements were made on a military helicopter to analyze the vibrations on the human body. The measurements in this industry, as well as others, are quite demanding, with many repetitions on different settings. Most of the measurement procedures can be done automatically using the prepared measurement programs; however some needed to be designed manually at the time of measurement.

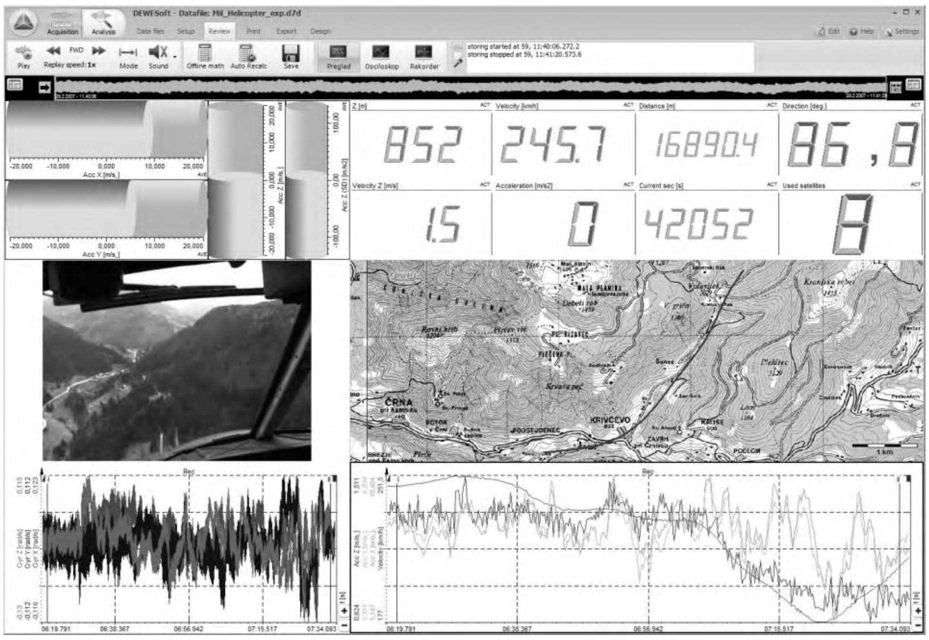


Fig. 1. Measurement system DEWESoft during helicopter vibration test

Many measurement system producers provide application programming interfaces (APIs) to use their products. Those APIs are further used by the customer’s programming engineers to build software according to their specific needs. However, a customer’s programming engineers do not necessarily posses knowledge about the problem domain; therefore they have to work with domain experts to prepare the desired product. In this way, prepared measurement procedures can be defined by programming engineers and further used by domain experts. As stated before, sometimes prepared measurement procedures are unsuitable and need to be repeated with slightly different settings. In that case, domain experts need to work with programming engineers to prepare another measurement procedure. Such development is time-consuming. An ideal measurement system would be, if domain experts could prepare the measuring procedures alone without the interference of programming engineers.

To support domain experts in programming their own measurement procedures and to be able to fine tune them during measurement, DEWESoft developed a domain-specific modeling language (DSML) called Sequencer. Our concrete motivation for this product was to enable domain experts to program/model their own data acquisitions and tune them during measurements without any help from programming engineers. Domain-specific languages (DSLs) provide notations and constructs tailored toward a particular application domain [1] and therefore are suitable for domain experts that have minor programming experience and expertise in the target problem domain [2]. Compared to general-purpose languages (GPLs), like C, C++, Java, etc, DSLs are much more expressive and easy of use [3] for the domain in question. However, DSL development is often accused of having disadvantages, since it requires both domain knowledge and, in particular, language development expertise, which is rare in the programming engineering community. Therefore, it is important to present practical evidence of developing DSLs in the industry [4] and provide results regarding the end-users' satisfaction. Also, the experiences gained through the development of the Sequencer are reported in this paper.

The line between DSLs and DSMLs is often blurred and it is hard to distinguish DSLs from DSMLs. The classification often depends on personal viewpoint. Up to now, DSLs are usually textual [5, 6, 7, 8], while DSMLs further raise abstraction level, expressiveness and ease of use, since models are specified in a visual manner and coding phase is moved to specification and design phase [9, 10]. With the Sequencer, measurement procedures are possible to specify in both text and visual form. Both options are alternatives to the previous one – to construct measurement procedures with an API, which is a standard development method when using GPLs. From that prospective, in this paper some of the experience are reported regarding which notation is more popular among DEWESoft customers, as well as their feedback.

The organization of the paper is as follows. In Section 2 related work on DSMLs is presented. The design details and characteristics of the Sequencer are described in Section 3. In Section 4, development and deployment together with our experiences are presented. Finally, contributions and concluding remarks with an outline for future work are summarized in Section 5.

2. Related work

Currently, scientists and engineers in diverse areas of work, as well as end-users with specific domain expertise, require computational processes to complete a task. However, such users are typically unfamiliar with programming languages and completing their task becomes a challenge. Model-driven engineering (MDE) is an approach that provides higher levels of abstraction to allow such users to focus on the problem, rather than the specific solution on particular technology platforms. An important part of MDE is a domain-specific (modeling) language DS(M)L that fit the domain of an end-user by offering intentions, abstractions, and visualizations for domain concepts. Many papers have been published recently on this topic and some of the most relevant ones are discussed in this section.

Jimenez, et al, show that combining a DSML with an MDE approach can enhance the quality and portability of home automation systems [11]. Most home automation systems are currently developed using proprietary low-level procedures that are platform dependent. To enhance productivity, flexibility, interoperability and end-user programming, a visual modeling language called Habitation has been designed and developed which enables the description of home automation systems using only domain concepts. The Eclipse Graphical Modeling Framework (GMF) has been used to automatically generate a graphic editor, while transformations are defined using the graph grammar approach (EMT - Eclipse Model Transformation). The main difference with our work is domains (home automation systems vs. measurement systems) and how both DSMLs have been developed. While Habitation has been developed using already existing metamodeling tools, we were not able to use them due to strong dependency on DEWESoft software.

Mathe, et al, present a Clinical Process Management Language (CPML) for capturing health treatment protocols [12]. The CPML is a formally specified visual modeling language developed using the metamodeling tool GME. The semantics have been specified using operational behavioral semantics. The semantics of the Sequencer is currently given by attribute grammars, which is used in the implementation phase, but do not enable a high level verification and analysis. In the future, our aim is to define Sequencer semantics using graph grammars.

Venigalla, et al, present a domain specific modeling language BASSML targeting spacecraft designers [13]. The BASSSML is a part of BASS, a prototype modeling tool for spacecraft systems. BASS consists of a model interpreter, which translates the captured spacecraft design models into machine-readable CSP (Communication Sequential Processes) that can be formally verified using a model checker. Using BASS, the authors show that spacecraft subsystem interfaces and interactions can be rigorously specified and analyzed. Hence, obscure subtle ambiguities and inconsistencies can be detected much earlier, thereby reducing developing costs.

Merilinna presents an end-user driven development of navigation applications for mobile phones [14]. For this purpose, a DSML was developed using the modeling environment MetaEdit+. The authors provide yet another piece of evidence that end-users, who are non-programmers, can actively participate in the development of navigation applications or develop applications completely by themselves using DSMLs within a narrow domain.

Živanov, et al, present KAG (Kiosk Application Generator), a DSL that can generate applications to be deployed on kiosks with touch-screen monitors. KAG is a nice example of DSL that upon textual specifications generates graphical-user interfaces using standard compiler generator tools (lex/yacc). Authors debate that comparing development of applications with KAG (and previous way, with general-programming languages), reduced number of programming errors and made kiosk applications development significantly faster.

A DOMMLite is the next example of DSMLs [16]. DOMMLite is used for definition of state structures of database applications. It was developed using generator framework openArchitectureWare. The domain-specific notation is defined with a metamodel supplemented by validation rules based on Check language and extensions based on Extend language that are parts of the openArchitectureWare framework. Semantics can be defined with specifications through source code generation for the supported target platforms. DOMMLite is supported with textual Eclipse editor.

DSMLs are prone to change much more often comparing to GPLs [17]. This is an emergent research area in MDE where models and modeling languages are subject to change [18]. However, in some environments, like DEWESoft, even dynamic language evolution might be necessary. In that case a system requires run-time adaptation without stopping an application. Possible solutions for adaptive DSML evolution are presented in [19, 20, 21].

3. Domain-specific modeling language Sequencer

Various implementation techniques to implement a DSL exist, such as: preprocessing, embedding, compiler/interpreter, compiler generator, extensible compiler/interpreter, and commercial off-the-shelf [1]. Of course, the language designer has to choose the most suitable implementation approach, according to the project influences [22]. In our case, the development was influenced by the fact that DS(M)L has to be included in the already-existing data acquisition software DEWESoft and that this product is developed in Object Pascal, more specifically in Delphi [23]. These limitations lead us to decide for compiler/interpreter implementation approach, where some of the compiler generator tools were used.

3.1. Construction of a textual concrete syntax

The development of DSML with the compiler/interpreter implementation approach gave us more freedom and flexibility than using other implementation approaches mentioned in [1]. In this approach, the standard compiler/interpreter techniques are used to implement a DSML. In the case of the compiler, a complete static analysis is done on the DSML program/specification. The most important advantage of this implementation approach is that the syntax is closer to the notation used by domain experts, and good error reporting. The compiler generator approach is similar to the previous one, except that some of the compiler/interpreter phases (lexical, syntax, and semantic analysis) are implemented using language development systems or so-called compiler writing tools (compiler-compilers) (e.g. Lex/Yacc [24], ANTLR [25], LISA [17], YAJCo [26]). In this manner, the implementation effort is minimized when compared to the previous approach.

Generally, the idea of a lexical analyzer is relatively simple. However, the construction and implementation of a lexical analyzer is time-consuming. Therefore, in the construction of a lexical analyzer, a compiler generator implementation approach can be used to speed up this process. In the case of the Sequencer, the help of DLex was used during the lexical analysis that generated a lexical analyzer in the programming language Delphi. With regular expressions, the formal description of the lexical analyzer was provided. Part of the DLex formal description of the Sequencer is presented in Fig. 2.

```
INTEGER                    [+-]?[0-9]+
FLOAT                      [+-]?[0-9]+(\.[0-9]+)?
BOOL                       "True"|"False"
STRING                     ['][a-zA-Z0-9.,
                           ;:%!?{|}#$&()<>=+@[\\\]/_-]*[']
COMMENT                    [/]{2,2}.*
IGNORE                     \n|\r|\r\n|" "|\t|\b
SEPARATOR                  "("|")"|","
FUNC                       "Action"  |"LoadSetup" |"If"
                           |"Loop"   |"WaitFor"   |"Delay"
                           |"AvdioVideo |"Formula"|"CustomBlock"
                           |"LaunchApplication"   |"Macro"
SPECWORDS                  "Begin"   |"End"
CONDTYPE                   "ctUser"  |"ctValue"   |"ctTrigger"
OPERATOR                   ">" | "<" | "=" | "!="
BOPERATOR                  "or" | "and"
LSTYPE                     "Static" | "Dynamic"
%%
{STRING}     begin
                   TokenList.Add(TToken.Create(yytext, tString,
                         yycolNo, yyLineNo));
             end;
...
```

Fig. 2. Lexical specification of Sequencer using DLex

The syntax and semantic analyzer has been developed independently of existing compiler generator tools. The syntax of the Sequencer was described using standard BNF notation. Part of the Sequencer's BNF is presented in Fig. 3. From the starting non-terminal NT_START, it can be seen that the reserved words (Begin, End) embody DSL statements that represent functionalities (non-terminal NT_LINE) to be performed from the measurement system DEWESoft. There are various non-terminals derived from the non-terminal NT_LINE: Action, LoadSetup, If, Loop, WaitFor, etc. For example, Action represents the basic functionality of the Sequencers' program (load project, export data, print, etc). If the load project is specified with an Action, then the hardware setup for a measurement procedure is performed. The non-terminal NT_ACTION is defined with non-terminals NT_B_ITEM (beginning parenthesis "("), NT_PACTION (action properties), and NT_E_ITEM (ending parenthesis ")" with reference to the following functionality: non-terminal NT_LINE). The non-terminal NT_PACTION contains specific properties for the current functionality, while the non-terminal NT_PROP contains generic properties. In non-terminal NT_PROP, first terminal (#integer) presents the ID of a construct, then #string represents the text info that will be presented to the Sequencer's user interface, #boolean terminal carries information if the Sequencer will notify the end-user with text-to-speech functionality, etc.

```
NT_START ::= "Begin" NT_LINE "End"
NT_LINE  ::= "Action" NT_ACTION
           | "LoadSetup" NT_LOADSETUP
           | "If" NT_IF
           | "Loop" NT_LOOP
           | "WaitFor" NT_WAITFOR
           | "Delay" NT_DELAY
           | "AvdioVideo" NT_AVDIOVIDEO
           | "Formula" NT_FORMULA
           | "CustomBlock" NT_CUSTOM_BLOCK
           | "LaunchApplication" NT_LAUNCHAPP
           | "Macro" NT_MACRO
           | epsilon
NT_ACTION ::= NT_B_ITEM NT_PACTION NT_E_ITEM
NT_B_ITEM := "(" NT_PROP
NT_E_ITEM ::= ")" NT_LINE
NT_PROP   ::= #integer "," #string "," #boolean "," #integer ","
              #integer "," #integer
...
```

Fig. 3. Syntax specification of Sequencer

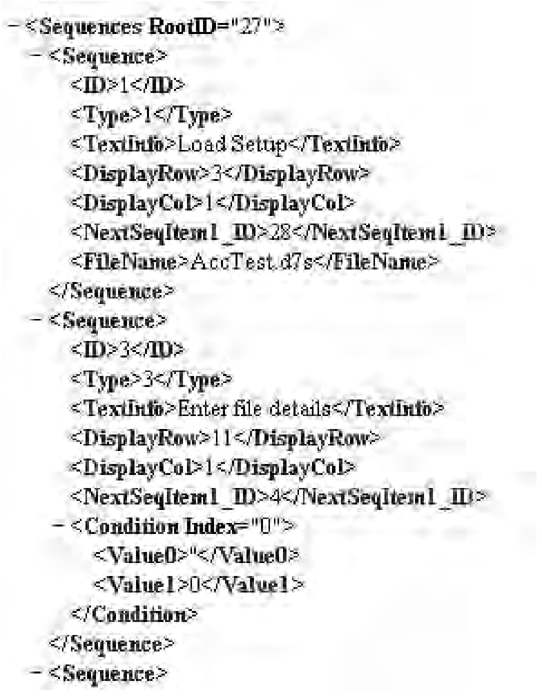
```
function TSeqParser.NT_LINE(Lexer : TLexer; Group :
                                    TSeqGroup) : Boolean;
var
  Item : TSeqItem;
  I : Integer;
begin
  Result := False;
  Item := nil;
  if (Lexer.CurrentToken.AType = tFunc) then
  begin
    if (Lexer.CurrentToken.Lexem = 'Action') then
    begin
      Lexer.NextToken;
      Item := Group.SeqItems.AddNewItem(it_Action);
      Result := NT_ACTION(Lexer, Group, Item);
    end
    else if (Lexer.CurrentToken.Lexem = 'LoadSetup') then
    begin
...
```

Fig. 4. Semantic of the Sequencer's non-terminal NT_LINE

The semantics of the Sequencer is described using attribute grammars from which a compiler is automatically generated. In the semantic part, attributes carry the values of actions defined in a DSL program and are responsible for calling functionalities from DEWESoft environment. Fig. 4 presents the part of the Pascal code for production NT_LINE. First, the token has to be checked which should be “tFunc” and the lexem should be “Action”. After that the lexical analyzer goes to the next token and to the next production which is in our case NT_ACTION.

The language processing effort is usually divided into syntax and semantic parts. In the syntax, the lexical analyzer and syntax analyzer size has been checked and 2,787 lines of code (LOC) have been generated or written. The semantic part of a code that contains all library calls to the DEWESoft framework contains 5,102 LOC. All together, the Sequencer DSL contains 7,889 LOC, which was developed in six engineer months. Since the Sequencer’s first release, new features and updates were occasionally introduced over the following six months, which were not counted as development time.

3.2. Transformations in the Sequencer



```
- <Sequences RootID="27">
  - <Sequence>
      <ID>1</ID>
      <Type>1</Type>
      <TextInfo>Load Setup</TextInfo>
      <DisplayRow>3</DisplayRow>
      <DisplayCol>1</DisplayCol>
      <NextSeqItem1_ID>28</NextSeqItem1_ID>
      <FileName>AccTest.d7s</FileName>
    </Sequence>
  - <Sequence>
      <ID>3</ID>
      <Type>3</Type>
      <TextInfo>Enter file details</TextInfo>
      <DisplayRow>11</DisplayRow>
      <DisplayCol>1</DisplayCol>
      <NextSeqItem1_ID>4</NextSeqItem1_ID>
    - <Condition Index="0">
        <Value0>"</Value0>
        <Value1>0</Value1>
      </Condition>
    </Sequence>
  - <Sequence>
```

Fig. 5. Sequencer’s code in XML

Transformations in the Sequencer are an important part of the tool. Their purpose is to transform programs into execution code that is further executed in the Sequencer’s framework. In the case of the Sequencer, the transformation occurs when a program is transformed into another presentation, execution model or vice versa. The transformation is carried out according to the selected initial and final model.

All transformations are in the group of exogenous transformations [27], because a model could never be transformed in the same model. Transformations enable one to change programs from XML to text or visual notation without any loss.

XML is also used in the Sequencer as an export and saving format (Fig. 5). Execution code in transformed into XML and with that feature, the portability and ability to exchange sequences between end-users and customers is supported.

3.3. Construction of visual concrete syntax

Beside textual notation, also visual notation has been developed for the Sequencer. For this purpose metamodels are often used. Usually, the metamodel is constructed using a standalone metamodeling tool [28, 29], a specialized software for the construction of DSMLs. However, DSML can have an implicit metamodel and in the case of the Sequencer, it was decided to prepare a fixed metamodel where the models were transferred to the execution model. In the Sequencer's metamodel the following domain concepts have been defined:

- a set of classes,
- associated attributes for each class,
- the relationship between classes, and
- constraints between classes.

Regarding the constraints in the Sequencer: there are no constraints on relations in the modeling language – each class can be connected to the others.

For each class a building block (concrete syntax) has been defined. In general, building blocks are separated into shapes and links. Each shape has a unique presentation in the form of a color, size and shape type (rectangle, diamond, ellipse, etc.). In the Sequencer, links have a unified form (line with arrow). Each shape belongs to exactly one building block and the link corresponds to a relationship. Each building block represents an action from a measurement system. Actions start their execution in the initial building block (marked with a circle) and continue to the next building block that is connected with the link.

Building blocks also contain local and global variables (that represent channels in DEWESoft). Their purpose is to store specific values in measurements. History is available for those variables and this is further used to plot graphs after the measurement is finished.

Regarding the Sequencer’s visual notation, a custom block has been introduced, that embodies several building blocks in a single one. When there are a lot of building blocks in a measurement procedure, a model can become unmanageable. With custom blocks, larger sequences can become more readable.

Nowadays, most measurement software is designed for capturing, storing and analyzing the measured data and do not allow the manual construction of the measuring process. They provide customizations, where you can tune the measurement procedure with only a few options. With the Sequencer, DEWESoft has decided to step forward and has developed a powerful DSML for the purpose of measurement procedures.

4. Results

In this section, the experience of using the Sequencer is discussed. Firstly, the Sequencer DSML is compared to other selected DSLs to observe its size and complexity. Then, Sequencer programs are compared to previous applications developed with DCOM objects. In the end, some experiences are reported from the end-users and numbers are given about how many customers are already using the Sequencer; the new feature of a DEWESoft product.

4.1. Sequencer complexity

From the language developer's point of view, it is worthwhile to observe the size of a language. The easiest way to do this is to compare it to other languages. It has been decided to compare just the Sequencers' textual notation and the following DSLs were chosen for comparison with Sequencer:

– Production Grammars (PG) for software testing [6],
– A DSL that allows experimentation for the different regulation of traffic lights (RoTL) and supports the domain-specific analysis of junctions [7],
– Context-Free Design Grammar (CFDG)[1], designed for generating pictures from specifications,
– GAL, a well-known DSL used to describe video device drivers [8].

One can get grammar examples with various compiler tools; however these are unsuitable for a comparison with the ones used in practice, since they are usually small owing to the fact that their value is in learning a specific tool notation. Our aim was to compare the Sequencer's grammar with the ones already applied in practice. In existing literature, those grammars are often partially presented, since they are usually too long to fit in the paper. Therefore, the above grammars were selected for comparison since they are used in practice and a full grammar was available to the authors of this paper.

[1] Context-Free Design Grammar, available at http://www.chriscoyne.com/cfdg/index.php

The size of a DSL can be compared to others using grammar metrics [30, 31]. In [30] grammar metrics are divided into size and structural metrics. For the purpose of our comparison we took the following size metrics:

- term – number of terminals,
- var – number of non-terminals,
- avs – average of right hand side size,
- mcc – McCabe cyclomatic complexity, and
- hal – Halstead effort.

Let us briefly discuss the above-mentioned metrics. A greater maintenance is expected if a grammar has a large number of non-terminals (var). The metrics mcc measure the number of alternatives for grammars' non-terminals. A high value indicates a potentially larger effort for grammar testing and a greater potential for parsing conflicts. A big avs value usually means that grammar is less readable. The Halstead effort metric (hal) estimates the effort required to understand the grammar. Grammar metric comparisons between the Sequencer and selected DSLs were obtained by the tool gMetrics [31] (Table 1). From the results of the size metrics, it can be concluded that the Sequencer is comparable to many of the selected DSLs. Of course, DSL complexity depends on the domain and can be much larger than other DSLs (observe GAL results on grammar metrics in Table 1).

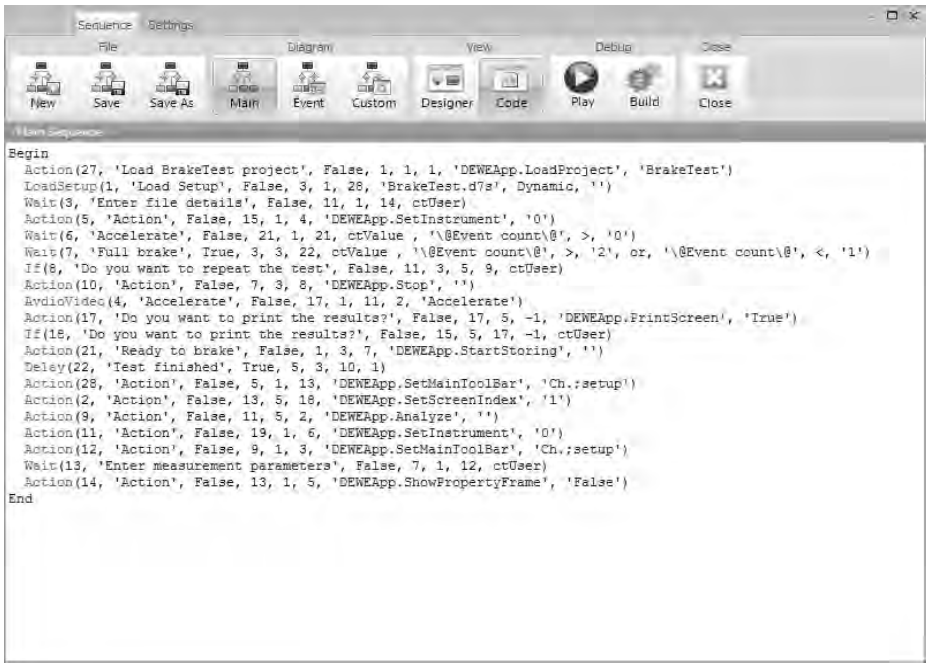
Table 1. Comparison of Sequencer with other DSLs

DSL	TERM	VAR	AVS	MCC	HAL
Sequencer	24	31	4.61	0.52	16.21
PG	10	5	3.80	1	0.89
RoTL	23	12	4.83	0.5	3.89
CFDG	24	13	6	2.38	6.57
GAL	71	74	3.88	1.20	33.36

4.2. Comparison of DCOM applications with the Sequencer's programs

The advantage of Sequencer over application development with DCOM objects can be observed when comparing a program from Fig. 6 with the DCOM application in Fig. 7. The advantages compared to APIs are obvious in respect to the clarity and understandability of the code.

Both programs (Fig. 6 and 7) describe the procedure (sequence) which is prepared to guide one through the entire car acceleration test maneuver. Besides the acceleration test, in the automotive industry, different measurements are applied to cars, like brakes, tires, a fuel consumption test, etc. The sequence in programs (Fig. 6 and 7) starts with the project and setup file load and the setup screen is shown. The start and stop speed can be set here. The next step is file details. Here the end-user has to set the file name and some test details (car type, driver, place, road surface, etc.). After this, the end-user starts driving. When reaching certain conditions (speed, temperature, pressure, distance) that are necessary to perform the acceleration test, the system advises the user to accelerate to the target speed. During the measurement process, the end-user can observe vehicle speed, vehicle acceleration, acceleration distance, temperature, etc. When the measurement is finished the end-user has the option of repeating the test or continuing to analyze and then printing out the stored data.



```
Begin
  Action(27, 'Load BrakeTest project', False, 1, 1, 1, 'DEWEApp.LoadProject', 'BrakeTest')
  LoadSetup(1, 'Load Setup', False, 3, 1, 28, 'BrakeTest.d7s', Dynamic, '')
  Wait(3, 'Enter file details', False, 11, 1, 14, ctUser)
  Action(5, 'Action', False, 15, 1, 4, 'DEWEApp.SetInstrument', '0')
  Wait(6, 'Accelerate', False, 21, 1, 21, ctValue , '\@Event count\@', >, '0')
  Wait(7, 'Full brake', True, 3, 3, 22, ctValue , '\@Event count\@', >, '2', or, '\@Event count\@', <, '1')
  If(8, 'Do you want to repeat the test', False, 11, 3, 5, 9, ctUser)
  Action(10, 'Action', False, 7, 3, 8, 'DEWEApp.Stop', '')
  AvdioVideo(4, 'Accelerate', False, 17, 1, 11, 2, 'Accelerate')
  Action(17, 'Do you want to print the results?', False, 17, 5, -1, 'DEWEApp.PrintScreen', 'True')
  If(18, 'Do you want to print the results?', False, 15, 5, 17, -1, ctUser)
  Action(21, 'Ready to brake', False, 1, 3, 7, 'DEWEApp.StartStoring', '')
  Delay(22, 'Test finished', True, 5, 3, 10, 1)
  Action(28, 'Action', False, 5, 1, 13, 'DEWEApp.SetMainToolBar', 'Ch.;setup')
  Action(2, 'Action', False, 13, 5, 18, 'DEWEApp.SetScreenIndex', '1')
  Action(9, 'Action', False, 11, 5, 2, 'DEWEApp.Analyze', '')
  Action(11, 'Action', False, 19, 1, 6, 'DEWEApp.SetInstrument', '0')
  Action(12, 'Action', False, 9, 1, 3, 'DEWEApp.SetMainToolBar', 'Ch.;setup')
  Wait(13, 'Enter measurement parameters', False, 7, 1, 12, ctUser)
  Action(14, 'Action', False, 13, 1, 5, 'DEWEApp.ShowPropertyFrame', 'False')
End
```

Fig. 6. Sequencer program in textual notation

```
unit Unit2;
interface
uses
  Windows,  Messages,  SysUtils,  Variants,  Classes,  Graphics,
Controls,
  Forms, Dialogs, AdvGlowButton, AdvToolBar, StdCtrls,
  AdvCaptionPanelUnit, DEWEsoft_TLB, ExtCtrls;
const
  bt_Yes = 1;
  bt_No = 2;
  bt_Continue = 4;
  SVSFlgAsync = $00000001;
type
  TForm2 = class(TForm)
    SequencerControlPanel: TAdvCaptionPanel;
    SequenceInfoLabel: TLabel;
    SequenceSeparator: TAdvToolBarSeparator;
    SequencePlayButton: TAdvGlowButton;
    ...
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Panel1Resize(Sender: TObject);
    ...
  private
    DeweApp : App;
    CurrState : Integer;
    oVoice : OLEVariant;//TTS
    procedure KillProcess(const ProcName : string);
    procedure SetHeader(Caption : string; Buttons : Integer);
  public
  end;
var
  Form2: TForm2;
implementation
uses
  Registry, TlHelp32, ComObj;
{$R *.dfm}
procedure TForm2.Panel1Resize(Sender: TObject);
begin
  if Assigned(DeweApp) then
  begin
    DeweApp.Left := 0;
    DeweApp.Top := 0;
      DeweApp.Width := panel1.Width;
             DeweApp.Height := panel1.Height;
  end;

end;
...
```

Fig. 7. DCOM application

Table 2. Comparison of Sequencer applications with DCOM applications in LOC

DSL	DCOM application	Sequencer	Ratio
Application 1	308	22	14
Application 2	298	15	19,86
Application 3	301	23	13,09
Application 4	280	20	14
Application 5	325	15	21,66

Another advantage can be observed if the Sequencer programs are compared with the DCOM application with the number of lines of code. In Table 2, the size of code (LOC) is presented for five different applications developed with Sequencer and DCOM objects. All Sequencer programs and DCOM applications have the same functionality. Table 2 confirms the advantage of Sequencer compared to the API solution (observe the ratio column in Table 2), since the Sequencer programs were at least 13 times shorter than the same DCOM applications. Similar productivity increase has been reported also elsewhere (e.g., [28]). Note, that applications in Table 2 are case study problems.

4.3. Customers' experiences

The DEWESoft product has already been successfully applied to the car industry. For example, the DEWESoft product is used by TÜV, an independent German consultant organization that validates the safety of products, like motor vehicles. Also, DEWESoft's measurement units (together with its software solution) are used in aviation, construction, electric and even aerospace industry. NASA awarded the DEWESoft product as "Product of the year" in 2009. From Table 3, it can be observed that DEWESoft has over 500 end-users who are using measurement systems for their specific measurements. Also, there are over 40 programming engineers who are using our DCOM objects to develop measurement procedures for their end-users.

Since January 2010, when Sequencer was released with DEWESoft ver. 7, over 150 end-users have already used the measurement procedures with the Sequencer. More than 30 domain experts are already developing sequences with the new feature of DEWESoft.

The real value of the Sequencer can be found in the last column of Table 3, which shows how many new domain experts have started using DEWESoft since the product became easier to use.

Table 3. DEWESoft customers

DCOM application end-users	DCOM programmers	Sequencer end-users	Sequencer domain experts	New domain experts on Sequencer
500	40	150	30	20

4.4. Sequencers' textual vs. visual notation

Both textual as well as visual concrete syntaxes have implemented the exact same functionalities and can therefore be transformed from one notation to another, as described in subsection 3.3. From the Sequencer developers' point of view, both notations are available to customers of the measurement system DEWESoft and they were not encouraged to use either of them.

Fig. 8 presents the Sequencers' modeling environment. The building blocks are on the left side of the environment. On the right side of the environment there are variables that can be selected for each individual building block. In the middle of the environment, the end-user can construct the measuring sequence with visual notation. Visual building blocks are used with "drag and drop" functionality. The Sequencer leads the end-user through a measurement procedure using static analysis, thereby reducing the possibility of human error and increasing the efficiency of the test itself.

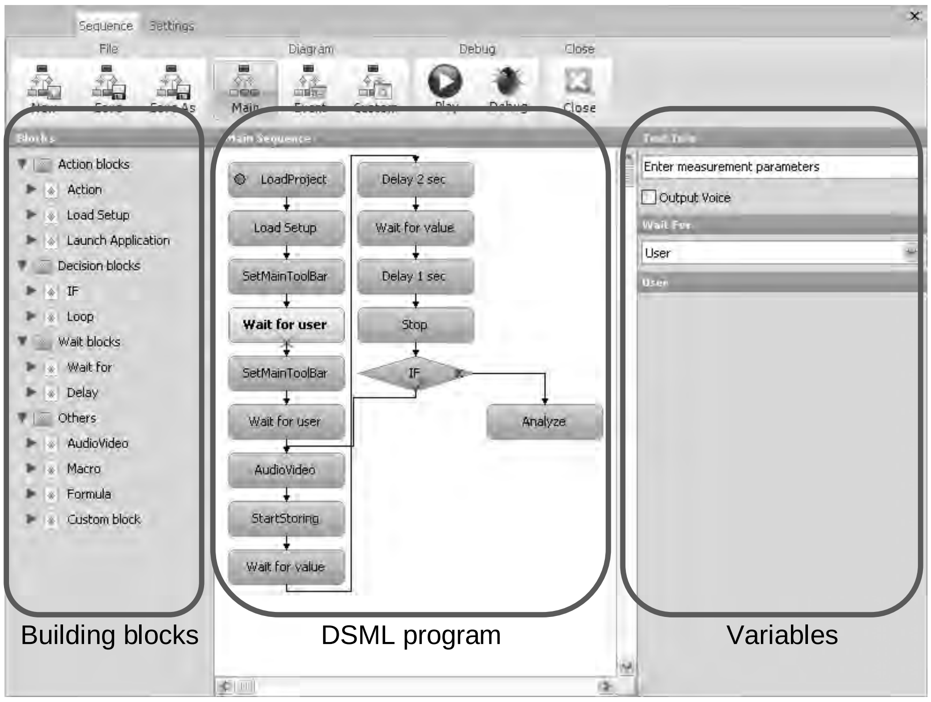


Fig. 8. Sequencer's modeling environment

Studying the Sequencers' domain experts revealed that most of them are using this visual notation rather than the text version of the Sequencer. The most probable explanation for this lies in the abstraction level of both notations. Also, there appears to be a general opinion that in order to use textual notation, the end-user needs a certain degree of programming experience. Both reasons, probably influenced end-users to prefer using the visual version of the Sequencer.

5. Conclusion and future work

The purpose of the Sequencer was to enable the easier construction of measurement procedures inside the measurement system DEWESoft. The main goal of the Sequencer is to push the development of the application from using DCOM objects to a specialized tool that enables domain experts to develop measurement sequences efficiently in a simple manner, without the need of support from programming engineers. Sequences can be developed in a textual or visual mode, which are customized for application development in the measurement domain. In this paper, the experiences in the development of Sequencer as well as experience with end-users were presented. According to the opinion of domain experts, the construction of the Sequencer has been a good step in simplifying complicated measurement development in many different fields.

From a usability point of view, the Sequencer's next feature is to record a sequence execution and save it in text format. In this manner, sequences can be analyzed in time to see more details. Currently, the system enables users to study the final results of the measurement test. From a DSML point of view, the next development effort will be to support domain experts with domain-specific debugging facilities similar to one presented by Wu, et al [32].

DS(M)Ls are promising for the future development of software, since current software development, centered on GPLs, is becoming more and more complex and software customization usually involves a larger effort on the part of programming engineers. On the other hand, DSLs enable domain experts to program and make changes in software and with that they can quicken development and reduce maintenance costs.

References

1. Mernik M., Heering J., Sloane A.M.: When and how to develop domain-specific languages. ACM Computing Surveys, Vol. 37, No. 4, 316–344. (2005)
2. Sprinkle J., Mernik M., Tolvanen J.-P., Spinellis D.: Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? IEEE Software, Vol. 26, No. 4, 15-18. (2009)
3. Kosar T., Oliveira N., Mernik M., Varanda Pereira M.J., Črepinšek M., da Cruz D., Henriques P.R.: Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. Computer Science and Information Systems, Vol. 7, No. 2, 247-264. (2010)
4. Ferreira E., Paulo R., da Cruz D., Henriques P.R.: Integration of the ST Language in a Model-Based Engineering Environment for Control Systems – An Approach for Compiler Implementation. Computer Science and Information Systems, Vol. 5, No. 2, 87-101. (2008)
5. Arora R., Bangalore P., Mernik M.: Raising the level of abstraction for developing message passing applications, The Journal of Supercomputing, (2010). Accepted for publication, doi: 10.1007/s11227-010-0490-3
6. Sirer E. G., Bershad B.N.: Using production grammars in software testing. In: Proceedings of the 2nd Conference on Domain-Specific Languages, pages 1-14. USENIX Association (1999)
7. Mauw S., Wiersma W.T., Willemse T.A.C.: Language-driven system design. International Journal of Software Engineering and Knowledge Engineering, Vol. 6, No. 14, 625-664. (2004)
8. Thibault S., Marlet R., Consel C.: Domain-specific languages: from design to implementation - application to video device drivers generation. IEEE Transactions on Software Engineering, Vol. 25, No. 3, 363-377. (1999)
9. Schmidt C.: Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer, Vol. 39, No. 2, 25-31. (2006)

10. Gray J., Tolvanen J.-P., Kelly S., Gokhale A., Neema S., Sprinkle J.: Domain-Specific Modeling. Handbook of Dynamic System Modeling. Boca Raton, Florida: CRC Press (2007)
11. Jimenez M., Rosique F., Sanchez P., Alvarez B., Iborra A.: Habitation: A Domain-Specific Language for Home Automation. IEEE Software, Vol. 26, No. 4, 30-38. (2009)
12. Mathe J., Ledeczi A., Nadas A., Sztipanovits J., Martin J., Weavind l., Miller A., Miller P., Maron D.: A Model-Integrated, Guideline-Driven, Clinical Decision Support System. IEEE Software, Vol. 26, No. 4, 54-61. (2009)
13. Venigalla S., Eames B., McInnes A.: A Domain Specific Design Tool for Spacecraft System Behavior. In DSM, Nashvile, TN (2008)
14. Merilinna J.: Domain-Specific Modelling Language for Navigation Applications on S60 Mobile Phones. In DSM Nashvile, TN (2008)
15. Živanov Ž., Rakić P., Hajduković M.: Using Code Generation Approach in Developing Kiosk Applications. Computer Science and Information Systems, Vol. 5, No. 1, 41-59. (2008)
16. Dejanović I., Milosavljević G., Perišić B., Tumbas M.: A Domain-Specific Language for Defining Static Structure of Database Applications. Computer Science and Information Systems, Vol. 7, No. 3, 409-440. (2010)
17. Mernik M., Žumer V.: Incremental programming language development. Computer Languages, Systems & Structures, Vol. 31, No. 1, 1-16. (2005)
18. Meyers B., Vangheluwe H.: A framework for evolution of modeling languages, Science of Computer Programming, doi:10.1016/j.scico.2011.01.002. (2011)
19. Kollár J., Václavík P., Wassermann Ľ.: Data driven Executable Language Model. In: Proceedings of the International Multiconference on Computer Science and Information Technology, pages 667-675, Polish Information Processing Society (2009)
20. Forgáč M., Kollár J.: Adaptive Approach for Language Modification. Journal of Computer Science and Control Systems, Vol. 2, No. 1, 9-12. (2009)
21. Kollár J., Forgáč M.: Combined Approach to Program an Language Evolution. Computing and Informatics, Vol.29, 1103-1116. (2010)
22. Kosar T., Martínez López P.E., Barrientos P.A., Mernik M.: A preliminary study on various implementation approaches of domain-specific language. Information and Software Technology, Vol. 50, No. 5, 390-405. (2008)
23. Cantù M.: Mastering Delphi 7. Sybex Inc. Alameda, CA. 2003
24. Levine J. R., Mason T., Brown D.: Lex & Yacc. O'Reilly, Cambridge, MA. 1992
25. Parr T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf. 2007
26. Porubän J., Forgáč M., Sabo M., Běhálek M.: Annotation Based Parser Generator. Computer Science and Information Systems, Vol. 7, No. 2, 291-307. (2010)
27. Bézivin J.: From Object-Composition to Model-Transformation with the MDA. In: TOOLS-USA'2001, Santa Barbara, USA. (2001)
28. Kelly S., Tolvanen J.-P.: Domain-Specific Modeling Enabling Full Code Generation. John Wiley & Sons, Inc. (2008)
29. Buchwalder O.: MEtaGile: An Agile Domain-Specific Modeling Environment. (2008)
30. Power, J. F. and Malloy, B. A.: A metrics suite for grammar-based software. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 16, No. 6, 405–426. (2004)
31. Črepinšek M., Kosar T., Mernik M., Cervelle J., Forax R., Roussel G.: On Automata and Language Based Grammar Metrics. Journal on Computer Science and Information Systems, Vol. 7, No. 2, 310-329. (2010)

32. Wu H., Gray J. G., Mernik M.: Grammar-driven generation of domain-specific language debuggers. Software practice and experience, Vol. 38, No. 10, 1073-1103. (2008)

Tomaž Kos has graduated at the Faculty of Electrical Engineering and Computer Science, University of Maribor, in 2009. Currently, he is a PhD student and he works for DEWESoft company as a researcher. His main research interests include programming languages, domain-specific (modeling) languages, testing, data acquisition, and measurement systems.

Tomaž Kosar received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research is mainly concerned with design and implementation of domain-specific languages. Other research interest in computer science include also domain-specific modelling languages, empirical software engineering, software security, generative programming, compiler construction, object oriented programming, object-oriented design, refactoring, and unit testing. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Jure Knez received the M.Sc. adn Ph.D degrees in mechanical engineering from the University of Ljubljana in 1999 and 2002 respectively. He is currently employed as CTO of Dewesoft d.o.o. in Trbovlje, Slovenia. The company is developing the software and measurement instrument widely used in automotive, aerospace, industrial, power distribution and civil engineering applications.

Marjan Mernik received the 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 at 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 (modelling) languages, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Received: December 31, 2010; Accepted: March 24, 2011.

DOI: 10.2298/CSIS101229018L

A DSL for PIM Specifications: Design and Attribute Grammar based Implementation

Ivan Luković[1], Maria João Varanda Pereira[2], Nuno Oliveira[3], Daniela da Cruz[3], and Pedro Rangel Henriques[3]

[1] University of Novi Sad, Faculty of Technical Sciences,
Trg D. Obradovića 6, 21000 Novi Sad, Serbia,
ivan@uns.ac.rs
[2] Polytechnic Institute of Bragança, Escola Superior de Tecnologia e Gestão, Campus de Santa Apolónia - Apartado 1134
5301-857 Bragança, Portugal
mjoao@ipb.pt
[3] Universisty of Minho, Department of Computer Science,
Campus de Gualtar - 4710-057 Braga, Portugal
{nunooliveira, danieladacruz, prh}@di.uminho.pt

Abstract. IIS*Case is a model driven software tool that provides information system modeling and prototype generation. It comprises visual and repository based tools for creating various platform independent model (PIM) specifications that are latter transformed into the other, platform specific specifications, and finally to executable programs. Apart from having PIMs stored as repository definitions, we need to have their equivalent representation in the form of a domain specific language. One of the main reasons for this is to allow for checking the formal correctness of PIMs being created. In the paper, we present such a meta-language, named IIS*CDesLang. IIS*CDesLang is specified by an attribute grammar (AG), created under a visual programming environment for AG specifications, named VisualLISA.

Keywords: information system modeling, model-driven approaches, domain specific languages, domain specific modelling, attribute grammars.

1. Introduction

In this paper we present a textual language aimed at modeling platform independent model (PIM) specifications of an information system (IS). Our research goals are to create such a language and couple it with Integrated Information Systems CASE Tool (IIS*Case). IIS*Case is a model driven software tool that provides IS modeling and prototype generation. At the level of PIM specifications, IIS*Case provides conceptual modeling of database schemas and business applications. Starting from such PIM models as a source, a chain of model-to-model and model-to-code transformations is performed in IIS*Case to obtain executable program code of software applications and database scripts for a selected target platform. One of the main motives for developing IIS*Case is in the following. For many years, the most favorable conceptual data model is widely-used Entity-Relationship (ER) data model. A typical scenario of a database schema design process provided by majority of existing CASE tools is to create an ER database schema first and then transform it into the relational database schema. Such a scenario has many advantages, but also there are serious disadvantages. One of them, presented in [11] is named "lack of semantic" problem. Actually, there are many examples in which the same structure of ER database schema should not be transformed into the same relational database schema structure, due to the different semantics assigned to the ER structure. In other words, the transformation process depends not only on the formal mapping rules, but also on the problem domain semantics. We overcome these disadvantages by creating an alternative approach and related techniques that are mainly based on the usage of model driven software development (MDSD) [3] and Domain Specific Language (DSL) [4, 10] paradigms. The main idea was to provide the necessary PIM meta-level concepts to IS designers, so that they can easily model semantics in an application domain. After that, they may utilize a number of formal methods and complex algorithms to produce database schema specifications and IS executable code, without any expert knowledge.

In order to provide design of various PIM models by IIS*Case, we created a number of modeling, meta-level concepts and formal rules that are used in the design process. Besides, we 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.

Apart from having created PIM models stored as repository definitions, there is a strong need to have their equivalent representation given in a form of a textual language, for the following reasons. (i) Firstly, despite that we may expect that average users prefer to use visually oriented tools for creating PIM specifications, we should provide more experienced users with a textual language and a tool for creating PIM specifications more efficiently. (ii) Secondly, we need to have PIM meta-level 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. (iii) The third, but not less important, by this we create a basis for the development of various algorithms for checking the formal correctness of the models being created, as well as for the implementation of some semantic analysis. Therefore, we need a grammatical specification to define the structure and semantics of our meta-level concepts and rules, i.e. we need an attribute grammar (AG) specification. By such a grammar, we specify a DSL [4, 15] that recognizes problem domain concepts and rules that are applied in the conceptual IS design provided by IIS*Case. In the paper, we present a specification of such meta-language, named IIS*CDesLang. IIS*CDesLang is used to create PIM project specifications that may be latter transformed into the other specifications, and finally to programs.

There are a number of meta-modeling approaches and tools suitable for the purpose of creating IIS*CDesLang. To create IIS*CDesLang, a visual programming environment (VPE) for AG specifications, named VisualLISA [19, 21] is selected. In the paper, we focus on the following application PIM concepts: project, application system, form type, component type, application, call type, and basilar concepts as attribute and domain. We applied VisualLISA Syntactic and Semantic Validators to check the correctness of the specified grammar.

A benefit of introducing IIS*CDesLang is to enable the creation of a parser aimed at checking the formal correctness of project models under development. In this way, we may help designers in raising the quality of new IS specifications. A possibility to build two translators, *IIS*Case repository-to-IIS*CDesLang specifications* and *IIS*CDesLang-to-IIS*Case repository definitions*, is another value added by this approach. The benefit of the first one is to allow the correctness checking of PIM visual models without explicitly writing IIS*CDesLang specifications; and the benefit of the second one is a possibility of generating correct PIM repository specifications from IIS*CDesLang textual specifications. Currently, we developed, using VisualLISA, an AG specifications of IIS*CDesLang. Apart from having the AG specification of IIS*CDesLang, we also need the appropriate checkers. They are still under development. Therefore, we were not able so far to test the efficiency of the concept as a whole. It remains to be one of our next research tasks. The main goal of this paper is to present a part of such VisualLISA specification and address main future research directions.

Apart from Introduction and Conclusion, the paper is organized in four sections. In Section 2 we present a related work, while in Section 3 we give a short presentation of IIS*Case. Selected IIS*CDesLang PIM concepts are briefly described in Section 4. In Section 5 we present preliminaries about VisualLISA programming environment and an AG specification of IIS*CDesLang, created by VisualLISA.

2. Related Work

Domain Specific Languages are tailored to specific application domain and offer to users more appropriate notations and abstractions. Usually DSLs are more expressive and are easier to use than GPLs for the domain in question, with gains in productivity and maintenance costs.

The design of a new DSL is usually made when it is needed to make programming more accessible to end-users, to improve correctness of the written programs, to improve the program developing time and to make maintenance easier.

There are various meta-modeling approaches and supporting tools suitable for the purpose of creating DSLs. One of them is the Meta-Object Facility

(MOF) [17] proposed by the OMG, where the meta-model is created by means of UML class diagrams and Object Constraint Language (OCL). The Generic Modeling Environment (GME) [23] is a configurable toolkit for domain-specific modeling and program synthesis. In MetaEdit+ [18] models are created through a graphical editor and a proprietary Report Definition Language is used to create code from models. The Eclipse Modeling framework (EMF) [5] is also a commonly used meta-modeling framework, where meta-meta-model named Ecore is used to create meta-models, or to import them from UML tools or textual notations like one presented in [6].

We may find a considerable number of references presenting the applications of such approaches and tools in various problem domains, as it is, for example, [8]. The same approaches can also be used for the design of IIS*CDesLang, too.

In general, our current research goals are to apply two closely related approaches to formally describe our IIS*Case environment. One of them is based on MOF and the appropriate Domain Specific Modeling (DSM) tools comprising specification language generators. The other one is applied in this paper. It is based on creating textual DSLs by means of the appropriate visually oriented tools with compiler generators. Although there is huge number of references covering many applications of both approaches in various problem domains, unfortunately, we still could not find references communicating ideas how to formally specify a CASE / MDSD tool by means of DSM and DSL approaches.

3. IIS*Case and Conceptual Modeling

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.

Apart from the tool, we also define a methodological approach to the application of IIS*Case in the software development process [12, 14]. By this approach, the software development process provided by IIS*Case is, in general, evolutive and incremental. It enables an efficient and continuous development of a software system, as well as an early delivery of software prototypes that can be easily upgraded or amended according to the new or changed users' requirements. In our approach we strictly differentiate between the specification of a system and its implementation on a particular platform. Therefore, modeling is performed at the high abstraction level, because a designer creates an IS model without specifying any implementation details. Besides, IIS*Case provides some model-to-model transformations from PIM to Platform-Specific Models (PSM) and model-to-code transformations from PSMs to the executable program code.

Detailed information about IIS*Case may be found in several authors' references and we do not intend to repeat them here. A case study illustrating main features of IIS*Case and the methodological aspects of its usage is given in [12]. The methodological approach to the application of IIS*Case is presented in more details in [14]. At the abstraction level of PIMs, IIS*Case provides conceptual modeling of database schemas that include specifications of various database constraints, such as domain, not null, key and unique constraints, as well as various kinds of inclusion dependencies. Such a model is automatically transformed into a model of relational database schema, which is still technology independent specification. It is an example of model-to-model transformations provided by IIS*Case [13].

In [1] we present basic features of SQL Generator that are already implemented into IIS*Case, and aspects of its application. We also present methods for implementation of a selected database constraint, using mechanisms provided by a relational DBMS. It is an example of model-to-code transformations provided by IIS*Case.

At the abstraction level of PIMs, IIS*Case also provides conceptual modeling of business applications that include specifications of: (i) UI, (ii) structures of transaction programs aimed to execute over a database, and (iii) basic application functionality that includes the following "standard" data operations: read, insert, update, and delete. Also, a PIM model of business applications is automatically transformed into the program code. In this way, fully executable application prototypes are generated. Such a generator is also an example of model-to-code transformations provided by IIS*Case [2].

4. PIM Concepts and IIS*CDesLang

IIS*CDesLang is a meta-language aimed at formal specification of all the concepts embedded into IIS*Case repository definitions. In this paper, we focus on the PIM concepts only. Hereby, we give a brief overview of the following concepts covered by IIS*CDesLang: project, application system, form type, component type, application, call type, as well as fundamental concepts: attribute and domain. In this section we present the PIM concepts only from the technical point of view. Additional and detailed information may be found in several authors' references, as well as in [12, 14].

A work in IIS*Case is organized through projects. Everything that exists in the IIS*Case repository is always stored in the context of a project. A designer may create as many projects as he or she likes. One project is one IS specification and has a structure represented by the project tree. Each project has its (i) name, (ii) fundamental concepts or fundamentals for short, and (iii) application systems. A designer may also define various types of application systems – application types for short, and introduce a classification of application systems by associating each application system to a selected application type. At the level of a project there is a possibility to generate various reports that present the current state of the IIS*Case repository. IIS*Case provides various types of repository reports.

Application systems are organizational parts, i.e. segments of a project. We suppose that each application system is designed by one, or possibly more than one designer. Fundamental concepts are formally independent of any application system. They are created at the level of a project and may be used in various application systems latter on. Fundamental concepts are: domains, attributes, inclusion dependencies and program units. In the paper, we focus on domains, attributes, and functions as a category of program units.

In the following text, we use a notion of domain with a meaning that is common in the area of databases. It denotes a specification of allowed values of some database attributes. We classify domains as (i) primitive and (ii) user defined. Primitive domains exist "per se", like primitive data types in various formal languages. We have a small set of primitive domains already defined, but we allow a designer to create his or her own primitive domains, according to the project needs. User defined domains are created by referencing primitive or previously created user defined domains. Domains are referenced latter from attribute specifications. A list of all project attributes created in IIS*Case belongs to fundamentals. Attributes are used in various form type specifications of an application system.

A concept of a function is used to specify any complex functionality that may be used in other project specifications. Each function has its name as a unique identifier, a description, a list of formal parameters and a return value type. Besides, it encompasses a formal specification of function body that is created by the *Function Editor* tool of IIS*Case.

4.1. Domains and Attributes

A specification of a primitive domain includes: name, description, default value, and a "length required" item specifying if a numeric length: a) not to be, b) may be or c) must be given. User defined domains are to be associated with attributes. A user defined domain specification includes: a domain name, description (like all other objects in IIS*Case repository), default value, domain type, and check condition.

We distinguish the following domain types: (i) domains created by the inheritance rule and (ii) complex domains that may be created by the: a) tuple rule, b) choice rule or c) set rule. Inheritance rule means that a domain is created by inheriting a specification of a primitive domain or a previously defined user defined domain. It inherits all the rules of a superordinated domain and may be "stronger" than the original one.

A domain created by the tuple rule is called a tuple domain. It represents a tuple (record) of values. For such a complex domain, we need to select some attributes as items of a tuple domain. Therefore, we may have a recursive usage of attributes and domains, because we need some already created attributes to use in a tuple domain specification. A domain created by the choice rule – choice domain is technically specified in the same way as tuple domain. Choice domain is the same as choice type of XML Schema Language. Each value of such a domain must correspond to exactly one attribute which is an item in the choice domain. A set domain represents sets (collections) of values over a selected domain. To create it, we only need to reference an existing domain as a set member domain. Each value of this domain will be a set of values, each of them from a set member domain.

Check condition, or the domain check expression is a regular expression that further constrains possible values of a domain. We have a formal syntax developed and the *Expression Editor* tool that assists in creating such expressions. We also have a parser for checking syntax correctness.

Currently we do not have a possibility to define allowed operators over a domain in IIS*Case repository. It is a matter of our future work.

Each attribute in an IIS*Case project is identified only by its name. Therefore, we obey to the Universal Relation Scheme Assumption (URSA) [11], well known in the relational data model for many years. The same assumption is also applicable in many other data models. Apart from the name and description, we specify if an attribute is included into database schema, derived, or renamed.

Most of the project attributes are to be included into the future database schema. However, we may have attributes that will present some calculated values in reports or screen forms that are not included into database schema. They derive their values on the basis of other attributes by some function, representing a calculation. Therefore, we classify attributes in IIS*Case as a) included or b) non-included in database schema. Also we introduce another classification of attributes, by which we may have: a) elementary or non-derived and b) derived attributes. If an attribute is specified as non-derived, it obtains its values directly by the end users. Otherwise, values are dervied by a function that may represent a calculation formula or any algorithm. Any attribute specified as non-included in database schema must be declared as derived one.

A derived attribute may reference an IIS*Case repository function as a query function. Query function is used to calculate attribute values on queries. Only a derived attribute may additionally reference three IIS*Case repository functions specifying how to calculate the attribute values on the following database operations: insert, update and delete.

In IIS*Case we have a notion of renamed attribute. A renamed attribute references a previously defined attribute and has to be included in the database schema. It has its origin in the referenced attribute, but with a slightly different semantics. Renaming is a concept that is analogous to the renaming that is applied in mapping Entity-Relationship (ER) database schemas into relational data model. If a designer specifies that an attribute A1 is renamed from A, actually he or she introduces an inclusion dependency of the form [A1] $\subseteq$ [A] at the level of a universal relation scheme.

Each attribute specification also includes: a reference to a user defined domain, default value and check condition. Check condition, or the attribute check expression is a regular expression that further constrains possible values of the attribute. It is defined and used in a similar way as it is for domain check expressions. If the attribute check expression and domain check expression are both defined, they will be connected by the logical AND.

Both user defined domain and attribute specifications also provide for specifying a number of display properties of screen items that correspond to the attributes and their domains. Such display properties are used by the IIS*Case *Application Generator* aimed at generating executable application prototypes. Display properties of an attribute may inherit display properties of the associated domain or may override them. To keep closed to the main goals of the paper, a detail technical description of display properties is omitted here. An interested reader may find it in [2, 24].

4.2. Application Systems, Form Types and Applications

Apart from name, type and description, each application system may have many child application systems. In this way, a designer may create application system hierarchies in an IIS*Case project. An application system may comprise various kinds of IIS*Case repository objects. For PIM specifications, only two kinds of objects are important: a) form types and b) business applications, or applications, for short.

A form type is the main modeling concept in IIS*Case. It generalizes document types, i.e. screen forms or reports by means of users communicate with an IS. It is a structure defined at the abstraction level of schema. Using the form type concept, a designer specifies a set of screen or report forms of transaction programs and, indirectly, specifies database schema attributes and constraints. Each particular business document is an instance of a form type.

Form types may be (i) owned, if they are created just in the application system observed, or (ii) referenced, if they are "borrowed" from another application system, regardless if it is referenced as a child application system. If a form type is referenced it is a read-only object in the application system.

Business applications are structures of form types. Each application has its name, description, and a reference to exactly one form type that is the entry form type of the application. To exist, each application must contain at least the entry form type. The execution of a generated application always starts from the entry form type. Form types in an application are related by form type calls. A form type call always relates two form types: a calling form type and a called form type. By a form type call, a designer may formally specify how values are passed between the forms during the call execution. There are also other properties specifying details of a call execution. *Business*

Application Designer is a visually oriented tool for modeling business applications in IIS*Case.

Each form type has the following properties: name, title, frequency of usage, response time and usage type or usage for short. By the usage property form types are classified as menus or programs. Menu form types are used to generate just menus without any data items. Program form types specify transaction programs with the UI. They have a complex structure and may be designated as (i) considered or (ii) not considered in database schema design. The first option is used for all form types aimed at updating database, as well as for some report form types. Only the form types that are "considered in database schema design" participate latter on in generating database schema. The former option is used for report form types only.

Each program form type is a tree structure of component types. It must have at least one component type. A component type has a name, reference to the parent component type (always empty for the root component type only), title, number of occurrences, and operations allowed. Number of occurrences may be specified as (i) 0-N or (ii) 1-N. 0-N means that for each instance of the parent component type, zero or more instances of the subordinated component type are allowed. 1-N means that for each instance of the parent component type, we require the existence of at least one instance of the subordinated component type. By operations allowed a designer may specify the following "standard" database operations over the component types: read, insert, delete, and update instances of the component type.

Each component type has a set of attributes included from IIS*Case repository. An attribute may be included in a form type at most once. Consequently, if a designer includes an attribute into a component type, it cannot be included in any other component type of the same form type. Each attribute included in a component type may be declared as: (i) mandatory or optional, and (ii) modifiable, query only or display only. Also, a set of allowed operations over an attribute in a component type is specified. It is a subset of the set of operations {query, insert, nullify, update}. A designer may also specify "List of Values" (LOV) functionality of a component type attribute by referencing a LOV form type and specifying various LOV properties. More information about LOV functionality and LOV properties may be found in [2, 24].

Each component type must have at least one key. A component type key consists of at least one component type attribute. Each component type key provides identification of each component instance, but only in the scope of its superordinated component instance. Also, a component type may have uniqueness constraints, each of them consisting of at least one component type attribute. A uniqueness constraint provides an identification of each component instance, but only if it has a non-null value. On the contrary to keys, attributes in a uniqueness constraint may be optional. Finally, a component type may have a check constraint defined. It is a logical expression constraining values of each component type instance. Like domain check expressions, they are specified and parsed by *Expression Editor*.

Both component type and form type attribute specifications provide for specifying a vast number of display properties of generated screen forms, windows, components, groups, tabs, context and overflow areas, and items that correspond to the form type attributes. There is also the *Layout Manager* tool that assists designers in specifying component type display properties, and a tool *UI*Modeler* that is aimed at designing templates of various common UI models. All of these display properties combined with a selected common UI model are used by the IIS*Case *Application Generator*. More information about display properties, *Layout Manager* and *UI*Modeler* may be found in [2, 24].

Our intention is not to present here the formal syntax rules of IIS*CDesLang in Backus-Naur (BNF) or an equivalent form, but just to illustrate them by means of a fragment of IIS*CDesLang program. A BNF specification of IIS*CDesLang is too complex and we believe that it would not contribute so much while communicate our main idea. However, apart from the selection of our references given here [1, 2, 11, 12, 13, 14, 24] there are many other references covering not only PIM concepts of IIS*Case, but also all the existing concepts of this environment, in detail. In some of them, we presented the IIS*Case concepts in a quite formal way, by means of the first order logic formulas, while in the others we presented our repository based and visually oriented tools for creating formal specifications in IIS*Case. All of such references are accessible upon request.

In the following example, we illustrate a form type created in an IIS*Case project named *FacultyIS*, and the corresponding IIS*CDesLang program. Figure 1 presents a form type defined in the child application system *Student Service* of a parent application system *Faculty Organization*. It refers to information about student's grades (STG). It has two component types: STUDENT representing instances of students, and GRADES, representing instances of grades for each student.

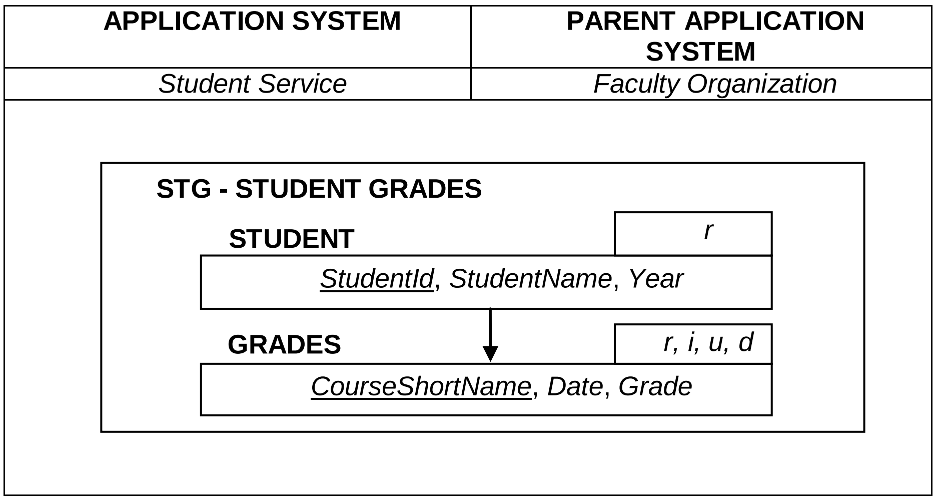


Fig. 1. A form type in the application system *Student Service*

By the form type STG, we allow having students with zero or more grades. Component type attributes are presented in italic letters. *StudentId* is the key of the component type STUDENT, while *CourseShortName* is the key of GRADES. By this, each grade is uniquely identified by *CourseShortName* within the scope of a given student. Allowed database operation for STUDENT is only *read* (shown in a small rectangle on the top of the rectangle representing the component type), while the allowed database operations for GRADES are *read*, *insert*, *update* and *delete*.

Figure 2 presents a fragment of IIS*CDesLang program that corresponds to the form type specification from Figure 1. Despite that it is just a fragment we present the program in a way to cover the specification as a whole. Just repeating segments of the specification, as well as a number of display and LOV properties are omitted. To better explain various segments of the program, we have included in-line comments tagged with the symbol //. In the following text, we give a textual explanation of the program from Figure 2.

Firstly, the project *FacultyIS* with its two application systems is specified. The first one is a specification of the *Faculty Organization* application system and then a specification of its child application system *Student Service*. After specifying the application system properties *Description* and *Type*, a list of form type specifications included in *Student Service* is given. In Figure 2 it is presented a specification of the form type *STG – Student Grades* only. Each form type specification includes properties *Title*, *UsageType* that may be *program* or *menu*, *UsageFrequency* and *ResponseTime*, and a list of component type specifications. A parent component type STUDENT and its child component type GRADES are specified in the form type *STG – Student Grades*.

The first, *Title* and *Allowed Operations* properties are specified for a component type. By this, *read* is the only allowed database operation for the component type STUDENT. After that, a list of display and other UI properties is specified. When generates UI of a transaction program of the form type *STG – Student Grades*, the component type STUDENT is to be positioned in a new window (*Position* property) and presented in a field layout style (*DataLayout* property). A window is to be centred to its parent window (*Window Position* property). Search functionality for student records is allowed (*Search Functionality* property), while multiple deletions (*Massive Delete Functionality* property) and retaining last inserted record in the screen form (*Retain Last Inserted Record* property) functionalities for student records are disabled. After the specifications of display and UI properties, it follows a list of specifications of component type attributes.

For each component type attribute we specify its name (*Name* property), title (*CTA_Title* property), if it is mandatory or optional for entering values on the screen form (*CTA_Mandatory* property), behavior (*CTA_Behavior* property) and allowed operations on the screen form (*CTA_AllowedOpera-tions* property). A set of display and LOV properties (preceded by *CTA_DisplayType* and *CTA_LOV_FormType* properties) may also be given.

```
Project: Faculty IS
  Application System: Faculty Organization
    Description: "A unit of a Faculty IS"
    Type: ProjectSubsystem
  ... // Specification of the appl. system continues...
  ...
  Application System: Student Service
                            is-child-of <<Faculty Organization>>
    Description: "A unit of Faculty Organization subsys."
    Type: ProjectSubsystem
    ...
    ... // A list of form types is specified here
    ...
    // A specification of the form type STG begins
    FormType: "STG - Student Grades"
      Title: "Catalogue of student grades"
      UsageType: Program Considered-in-db-design: Yes
      UsageFrequency: 1 Unit: seconds
      ResponseTime: 1 Unit: seconds

      // A specification of the component type begins
      ComponentType: STUDENT
        Title: "Student Records"
        Allowed Operations: read
        Position: newWindow
        DataLayout: FieldLayout
        Window Position: Center
        Search Functionality: Yes
        Massive Delete Functionality: No
        Retain Last Inserted Record: No
        Component Type Attributes:
          Name: StudentID
            CTA_Title: "Student Id."
            CTA_Mandatory: Yes
            CTA_Behavior: queryOnly
            CTA_AllowedOperations: query
            CTA_DisplayType: textbox Height: 20 ...
            // More display properties are omitted ...
            CTA_LOV_FormType: <<STD - Student>> ...
            // More LOV properties are omitted ...
          Name: StudentName
            ...
          Name: Year
            ...
        Component Type KEY: StudentID
      // A specification of the component type ends

      // A specification of the component type begins
      ComponentType: GRADES is-child-of <<Student>>
        NoOfOccurrences: (0:N)
        Allowed Operations: read, insert, update, delete
        Position: sameWindow
        Layout Relative Position: Bottom-to-parent
        DataLayout: TableLayout
        Window Position: Center
        Search Functionality: Yes
        Massive Delete Functionality: No
        Retain Last Inserted Record: Yes
        Component Type Attributes:
          Name: CourseShortName
            CTA_Title: "Course Short Name"
            CTA_Mandatory: Yes
            CTA_Behavior: modifiable
            CTA_AllowedOperations: query, insert
            CTA_DisplayType: textbox Height: 20 ...
            CTA_LOV_FormType: <<CRS - Courses>> ...
          Name: Date
            ...
          Name: Grade
            ...
        Component Type KEY: CourseShortName
      // A specification of the component type ends
    // A specification of the form type STG ends
    ...
    ... // Specification of form types continues...
  ... // Specification of the project continues...
  ...
```

Fig. 2. A fragment of IIS*CDesLang program that correspond to the form type in Fig. 1

After the list of component type attributes, the list of component type constraints is given. It may include the specifications of key, uniqueness and check constraints. In the example shown in Figure 2, only component type keys are specified for STUDENT and GRADES by the property *Component Type KEY*.

5. The Attribute Grammar Specification of IIS*CDesLang

In this section, an AG specification of IIS*CDesLang, created by VisualLISA will be described. The IIS*Case concepts, introduced along the previous section, will now be mapped into IIS*CDesLang symbols establishing a correspondence between domain concepts and non-terminal or terminal grammar symbols in the systematic way described in [9].

To provide an easier following of the rest of the paper, we firstly introduce a brief overview of the notion of AG [7]. An AG is a fvie-tuple AG = <CFG, A, R, CC, TR> where: CFG is a Context-free Grammar, also given as a four-tuple CFG = <T, N, S, P>; A is the set of attributes for all symbols in N or T; R is the set of all the attribute evaluation rules associated with each production p in P; CC is the set of contextual conditions (or predicates constraining the attribute values) associated with each production p in P; and TR is the set of all translation rules (that output attribute values) associated with each production p in P. Notice that attributes a in A(t), associated with terminal symbols, are evaluated outside the grammar rules. Their values are called "intrinsic" and are provided by the lexical analyzer. However attributes associated with an non-terminal symbol X (denoted by A(X) can be: synthesized (AS(X)), if their value is evaluated when X appears in the left-hand side of a grammar rule; or can be inherited (AI(X)), if their value is evaluated when X appears in the right-hand side of a grammar rule, using the values of parent or sibling symbols. So we can state that for each X in N, $A(X) = AI(X) \cup AS(X)$.

Although the same term "attribute" is used in this paper as a well known concept in two different contexts: (i) in Section 4, in the domain of databases and information systems and (ii) in Section 5, as a concept of AGs, it is important to notice that it is generally speaking the same concept. It is used in the sequel (associated with symbols) in the context of grammars, in the same way as it is in the context of object-oriented models/programs, or databases; in all of these contexts, the notion of attribute denotes a characteristic that gives semantic to the thing we are formally describing – a grammar symbol, a class, or even a relation scheme/entity type.

As it can be inferred from AG definition above, to write a complete attribute grammar for a real size programming language is a systematic and disciplined work. However it is time consuming and repetitive task.

Although not a complex task, in a case of real size grammar it tends to be time consuming process requiring a careful work. This inconvenience discourages language designers to use AGs. Such an attitude prevents them of resorting to systematic ways to implement the languages and their supporting tools [22].

To overcome this drawback, for modeling the new DSL we use a Visual Language (VL) and its respective VPE called VisualLISA, as it is proposed in [21], and conceived in [19]. The idea of introducing VL is not only about having a nice visual depiction that will be translated into a target notation latter on, but also having a possibility of checking syntactic and semantic consistency.

VisualLISA environment offers a visually oriented and non-errorprone way for AG modeling and an easy translation of AG models into a target language. Three main features of VisualLISA are: (i) syntax validation, (ii) semantics verification and (iii) code generation. The syntax validation restricts some spatial combinations among the icons of the language. In order to avoid syntactic mistakes, the model edition is syntax-directed. The semantics verification copes with the static and dynamic semantics of the AG meta-language. Finally, the code generation produces code from the drawings sketched up. The target code would be LISA specification language (LISAsl), the meta-language for AG description under LISA generator [19]. LISAsl specification is passed to the LISA system [16, 20] in a straightforward step.

In this section, we discuss how VisualLISA is used to create IIS*CDesLang. We only present a small set of productions and semantic calculations, to show how we use the visual editor to model the language. Before that we present a short description of VisualLISA look and feel, and main usage.

Figures 3-6 show the editor look and feel; it exhibits its main screen with four sub-windows. To specify an AG a user starts by declaring the productions in *rootView* – sub-window presented in Figure 3, and rigging them up by dragging the symbols from the dock to the editing area in *prodsView* – sub-window presented in Figure 4, as commonly done in VPEs. The composition of the symbols is almost automatic, since the editing is syntax-directed. When the production is specified, and the attributes are attached to the symbols, the next step is to define the attribute evaluation rules. Once again, the user drags the symbols from the dock, in *rulesView* – sub-window presented in Figure 5, to the editing area. To draw the computations links should connect some of the (input) attributes to an (output) attribute using functions. Functions can be pre-defined, but sometimes it is necessary to resort to user-defined functions that should be described in *defsView* – sub-window presented in Figure 6. In this sub-window it is also possible to import packages, define new data-types or define global lexemes.

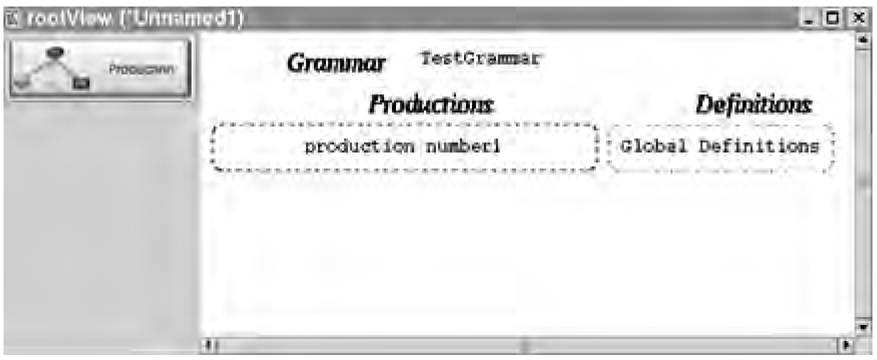


Fig. 3. VisualLISA subwindow for declaring productions

In this example, presenting the development of the IIS*CDesLang formal specification with VisualLISA, we will show how the following condition is formalized and verified using the visual editor: *"The application types associated to application systems should be previously defined"*.

For a thorough understanding of the upcoming example, here follows a brief overview of the visual symbols semantics. The cloud-shaped symbol is the left-hand side (LHS) of a production; the squares and ellipses are the terminals and the non-terminals at the right-hand side (RHS) of a production, respectively. The triangles represent the attributes: inherited attributes are inverted triangles, while the other triangles are synthesized attributes. The explosion-shaped symbol represents a function to compute the attributes value. Concerning the lines and the arrows: the simple lines represent the connection between the LHS and the RHS symbols; the dashed lines represent the connections between the symbols and the synthesized and the inherited attributes; the full arrow means the copy of a value from an attribute to another; the dashed arrow with a number over it represents an ordered argument of a function and, finally, the full arrow from an explosion-shaped symbol stands for the output of the function.

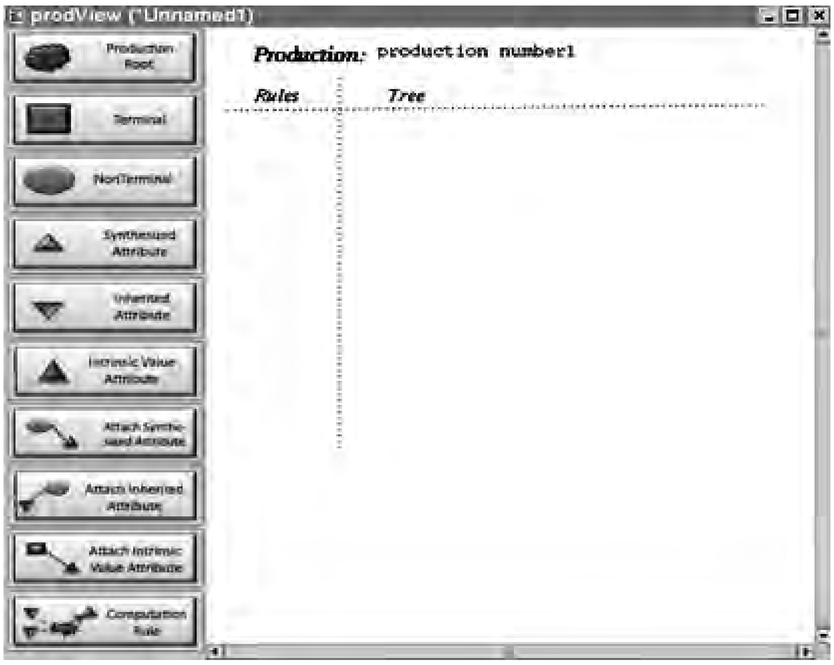

prodView (*Unnamed1)
Production Root
Terminal
NonTerminal
Synthesized Attribute
Inherited Attribute
Intrinsic Value Attribute
Attach Synthesized Attribute
Attach Inherited Attribute
Attach Intrinsic Value Attribute
Computation Rule
Production: production number1
Rules
Tree

Fig. 4. VisualLISA subwindow for selecting symbols

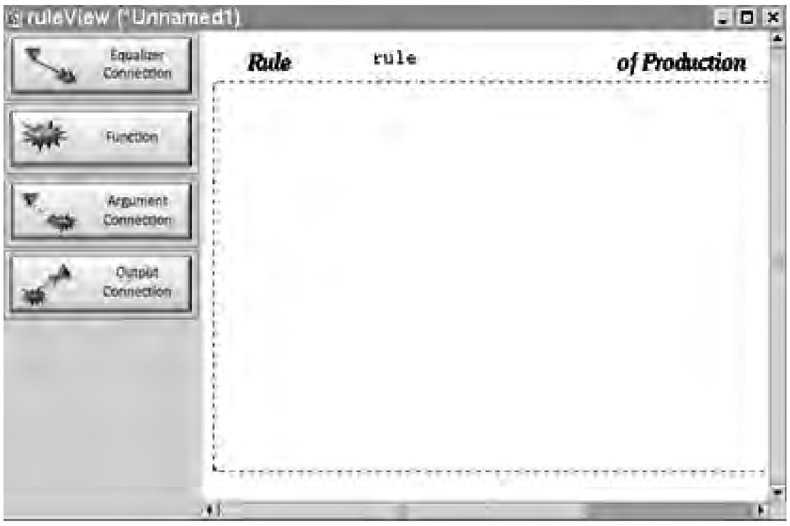
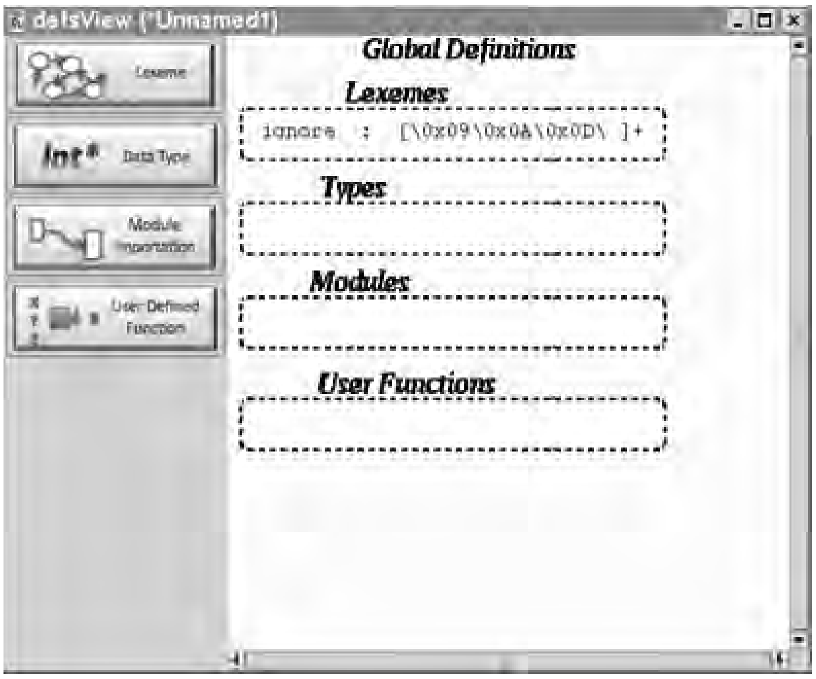


Fig. 5. VisualLISA editing area subwindow



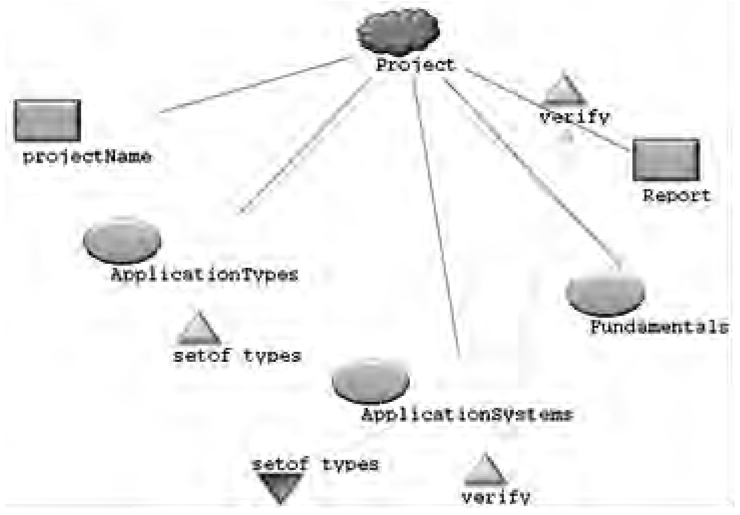
```
ignore  :  [\0x09\0x0A\0x0D\ ]+
```

Fig. 6. VisualLISA subwindow for creating user defined functions, importing packages, defining new data-types and global lexemes

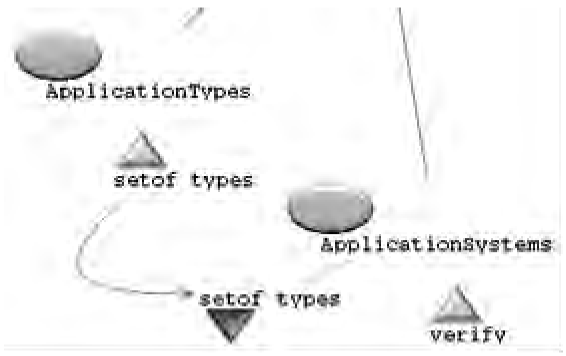
Figure 7 shows the first production of IIS*CDesLang – the one having the grammar axiom as the tree root. The root *Project* (see Figure 7.a) derives in three other non-terminal symbols (*ApplicationTypes*, *ApplicationSystems*, and *Fundamentals*) and two terminals. Apart from that structural description, the production shown in Figure 7.a states that the attribute *verify* of the root symbol has the same value as the synthesized attribute *verify* (triangle) of the non-terminal *ApplicationSystems*. In Figure 7.b it is presented a detail of the same production, specifying that the inherited attribute *setof_types* (inverted-triangle) of non-terminal *ApplicationSystems,* inherits the value of the attribute *setof_types* of the non-terminal *ApplicationTypes*.

In Figure 8, we present how the attribute *setof_types* of the non-terminal *AplicationTypes*, is computed. First notice that the production for this non-terminal has two options: (i) a non-recursive one, where *AplicationTypes* derives only one *AplicationType* (Figure 8.a) and (ii) a recursive case, where the left-hand side non-terminal derives into an *AplicationType* and recursively calls itself.

In this production, we are interested in collecting the application type names that can be associated to the application systems, as explained before. To describe this in VisualLISA we created a function that adds a string to a list, and this function is used to collect the types that are synthesized from each non-terminal *ApplicationType*. The explosion symbol denotes the function, the dashed-arrows define the arguments of these functions, and the straight arrows denote to which attribute the output of the function is assigned. The numbers in the dashed-arrows indicate the order of the arguments in the function, which are then used as '$i' in the function body, where $1 is the first function argument and $2 the second; in general, $i represents the value of the i-th argument.



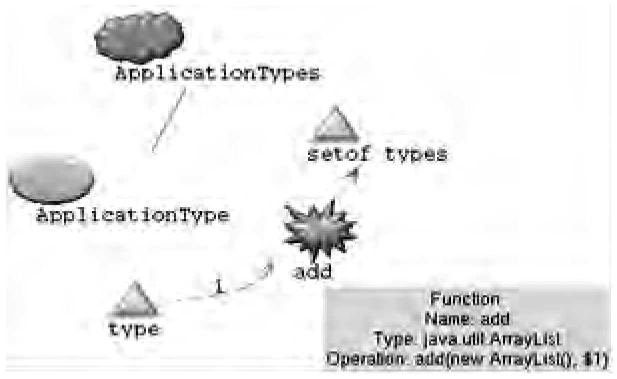
(a)



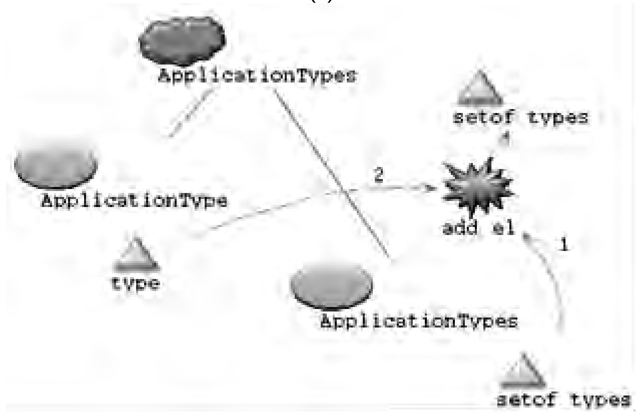
(b)

Fig. 7. Production structure and computation rules for non-terminal Project. (a) computation rule for attribute *verify*; (b) computation rule for inherited attribute *setof_types*

Recall Figure 7.b, where an inherited attribute is assigned the value of the attribute we just compute in Figure 8. The reason why we need to inherit this attribute is in the fact that we must check whether the type of each application system is in this list. Otherwise the language is not correct according to the contextual condition that we try to verify in this example. Figure 9 presents the recursive option of the production with the *ApplicationSystems* as LHS symbol.



(a)



(b)

Fig. 8. Production structure and computation of attribute *setof_types* of the element *ApplicationTypes*. (a) non-recursive case; (b) recursive case.

From each application system we synthesize its application type (attribute *app_type*). Then, we use the inherited attribute *setof_types* and the value that results from applying this computation to the rest of the application systems in the language, to inject these three arguments in a function that tests if the *setof_types* ($1 in the operation description of Figure 9) contains the value of the synthesized attribute *app_type* ($2 in the operation description). As this operation returns a boolean value, we check using the logic *and* operation, if this value and the value of the attribute *verify* ($3 in the operation description) are both true. The output of the function is also a Boolean and is assigned to attribute *verify* of the LHS symbol.

The non-recursive option of this production is similar, but the computation of the final attribute is only based on the list of types and the type that comes from the *ApplicationType* symbol.

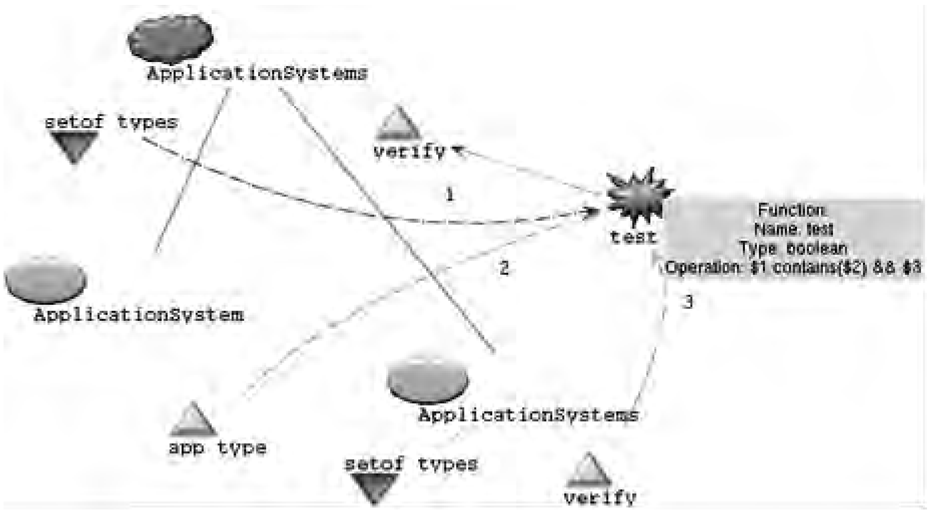


Fig. 9. Recursive case for production of the symbol *ApplicationSystems* and computation of the attribute *verify*.

Although the drawings presented in Figures 7 to 9 have been formally constructed, for those that read the visual grammar it is not necessary to know if attributes are synthesized or inherited, neither the way evaluation rules are built – it is enough to understand the way they are connected to understand the new language semantics. The remaining parts of the formalization follows the same structure as the one presented in this section.

To develop incrementally a DSL using VisualLISA is very easy. Just define a new set of attributes (corresponding to the next semantic step) and the respective evaluation rules and draw this new semantic specification over a syntax tree (a production) previously created. VisualLISA environment will automatically add this new component to the ones existing for the same symbols. However VisualLISA does not include any operator for grammar inheritance or symbol/production extension in LISA style.

With VisuaLISA we defined a model of IIS*CDesLang PIM concepts. The IIS*CDesLang productions were visually modelled, checked and translated to LISA specifications. This model can be turned into a valid AG, and in a straightforward step, we have not only a new language, but also a compiler for the language.

We list below the textual format for the most important IIS*CDesLang productions of the AG outputted by VisualLISA environment. Those are the productions that in general cover the concepts of: project, application system, form type and component type. Notice that we transcribe them in a neutral AG-format to avoid that the reader must learn LISA syntax.

The first production is:

Project → *ProjectName* ApplicationType+ **ApplicationSystem**+
Fundamentals Reports

It defines a project specifying a name (*ProjectName*), a set of possible types of application systems (ApplicationType), a set of application systems created in the scope of a project (ApplicationSystem), fundamental concepts (Fundamentals) and category of a repository report (Reports).

The production defining the application system is:

ApplicationSystem → *AppSystemName* *AppSystemDescription*
ApplicationTypeName **FormTypes**
BusinessApplication+ ChildAppSystem+
RelationScheme+ JoinDependency+
ClosureGraph Reports

It specifies a name (*AppSystemName*), a description (*AppSystem Description*), a type of application system (ApplicationTypeName), a category of a form type (FormTypes), a set of business applications (BusinessApplication), a set of child application systems (ChildAppSystem), a set of generated relation schemes (RelationScheme), a set of created join dependencies (JoinDependency), a closure graph (ClosureGraph) and category of application system specific reports (Reports).

At this point, it is needed to verify if the application system type specified for an application system belongs to the set of possible types:

ApplicationSystem.ApplicationTypeName.value belong_to
{set_of(ApplicationType.ApplicationTypeName .value)}

Just as an illustration, we give here selected productions covering the form type and component type concepts:

FormTypes → OwnedFormType+ ReferencedFormType+

OwnedFormType → FormTypeName FormTypeTitle FTFrequency
FTResponseTime FTParameter+ CalledFormType+
FTUsage

FTUsage → *Menu* | Program

Program → ComponentTypeTreeStructure *ConsideredInDBSchDesign*

ComponentTypeTreeStructure → **ComponentType**+

ComponentType → *CTName* *CTParent* *NoOfOcurrences*
*CTTitle*AllowedOperations ComponentDisplay
ItemGroup+ComponentTypeAttribute+
ComponentTypeKey+ ComponentTypeUnique+
ComponentTypeCheckConstraint

These productions also have a set of semantic conditions that must be verified.

6. Conclusion

AGs are widely used to specify the syntax (by the underlying Context Free Grammar) and the semantics (by the set of attributes and theirs computation rules and contextual conditions) of computer languages. This formalism is well defined and so its usage is completely disciplined; but, more than that, it has the unique property of supporting the specification of syntax and semantics under the same framework. Moreover, an AG can be automatically transformed into a program to process the sentences of the language it defines.

The research presented in this paper resulted from the collaborative research project between Serbia and Portugal. To formally describe the Integrated Information Systems CASE Tool (IIS*Case) – a model driven software tool that provides IS modeling and prototype generation developed at University of Novi Sad – we define a DSL, named IIS*CDesLang, that encompasses problem domain concepts and rules that are applied in the conceptual IS design provided by IIS*Case. In the paper, we present such a meta-language resorting to a VPE for attribute grammar specifications, named VisualLISA, developed at University of Minho. VisualLISA makes the process of AG development easier and safer; it allows the drawing of the AG productions (grammar rules) in the form of attributed trees decorated with attribute evaluation rules. These visual productions are syntactically and semantically checked for correctness.

Currently, we are completing the IIS*CDesLang AG specification to cover all the IIS*Case. After that, we will resort to the compiler generator system LISA to produce a compiler for IIS*CDesLang.

On the basis of the problem domain knowledge embedded in the AG, the generated compiler will also provide semantic analyses of the designed specifications and further assist designers in raising the quality of their work. Two characteristic examples are domain compatibility analysis and check constraint equivalence analysis. We plan to include a textual editor for IIS*CDesLang into IIS*Case, and integrate into it the generated compiler to couple IIS*Case repository with the formal IIS*CDesLang descriptions.

Moreover, as future work we plan to build a translator from IIS*Case Visual PIM specifications into textual IIS*CDesLang descriptions. This will allow to verify the specifications correctness without writing them manually in IIS*CDesLang. Also, it will be possible and interesting to implement the automatic generation of PIM specifications from IIS*CDesLang descriptions.

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

References

1. Aleksić, S., Luković, I., Mogin, P., Govedarica, M.: A Generator of SQL Schema Specifications. Computer Science and Information Systems (ComSIS), Consortium of Faculties of Serbia and Montenegro, Novi Sad, Serbia, ISSN:1820-0214, Vol.4, No. 2, 79--98. (2007)
2. Banović J.: An Approach to Generating Executable Software Specifications of an Information System. Ph.D. Thesis. University of Novi Sad, Faculty of Technical Sciences in Novi Sad. (2010)
3. Bézivin J., On the unification power of models, Software and Systems Modeling, Vol. 4, No. 2, 171--188. (2005)
4. Deursen van, A., Klint, P. Visser, J.: Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices, Association for Computing Machinery, USA, Vol. 35, No. 6, 26--36. (2000)
5. Eclipse Modeling Framework [Online] Available: http://www.eclipse.org/modeling/emf/ (current April, 2011)
6. Jouault F., Bézivin J.: KM3: a DSL for Metamodel Specification, In: Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy, Springer LNCS 4037, 171--185. (2006)
7. Knuth, D. E.: Semantics of Context-free Languages. Theory of Computing Systems, Vol 2, No. 2, 127--145. (1968)
8. Krahn H., Rumpe B., Völkel S.: Roles in Software Development using Domain Specific Modelling Languages, In: Proceedings of 6th OOPSLA Workshop on Domain-Specific Modeling, Portland, USA, 150--158. (2006)
9. Kosar T., Mernik M., Henriques P.R., Varanda Pereira M.J, Žumer V.: Software development with grammatical approach. Informatica, ISSN: 1854-3871, Vol. 28, No. 4, 39--404. (2004)
10. Kosar T., Oliveira N., Mernik M., Varanda Pereira M.J., Črepinšek M., da Cruz D., Henriques P.R.: Comparing General-Purpose and Domain-Specific Languages: An Empirical Study. Computer Science and Information Systems (ComSIS), Consortium of Faculties of Serbia and Montenegro, Novi Sad, Serbia, ISSN:1820-0214, Vol. 7, No. 2, 247--264. (2010)
11. Luković I.: From the Synthesis Algorithm to the Model Driven Transformations in Database Design, In: Proceedings of 10th International Scientific Conference on Informatics (Informatics 2009), Herlany, Slovakia, ISBN 978-80-8086-126-1, 9--18. (2009)
12. Luković, I., Mogin, P., Pavićević, J., Ristić, S.: An Approach to Developing Complex Database Schemas Using Form Types. Software: Practice and Experience, John Wiley & Sons Inc, Hoboken, USA, DOI: 10.1002/spe.820, Vol. 37, No. 15, 1621--1656. (2007)
13. Luković, I., Ristić, S., Aleksić, S., Popović, A.: An Application of the MDSE Principles in IIS*Case. In: Proceedings of III Workshop on Model Driven Software Engineering (MDSE 2008), Berlin, Germany, TFH, University of Applied Sciences Berlin, 53--62. (2008)
14. Luković, I., Ristić, S., Mogin, P., Pavićević, J.: Database Schema Integration Process – A Methodology and Aspects of Its Applying. Novi Sad Journal of Mathematics, Serbia, ISSN: 1450-5444, Vol. 36, No. 1, 115--150. (2006)
15. Mernik, M., Heering, J., Sloane, M. A.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys (CSUR), Association for Computing Machinery, USA, Vol. 37, No. 4, 316--344. (2005)

16. Mernik, M., Lenič, M., Avdičaušević, E., Žumer, V.: LISA: An Interactive Environment for Programming Language Development. In: Proceedings of Compiler Contruction, LNCS Vol. 2304, 1--4. (2002)
17. Meta-Object Facilty [Online] Available: http://www.omg.org/mof/ (Current: April, 2011)
18. MetaCase Metaedit+ [Online] Available: http://www.metacase.com/ (Current: April, 2011)
19. Oliveira, N. Varanda Pereira, M.J., Henriques, P.R., Cruz, D., Cramer, B.: VisualLISA: A Visual Environment to Develop Attribute Grammars. Computer Science an Information Systems Journal, Special Issue on Compilers, Related Technologies and Applications (ComSIS), Lukovic, I. and Leitão A, Slivnik B. (Guest Eds.), ISSN:1820-0214, Vol. 7, No. 2, 265--289. (2010)
20. Varanda Pereira, M.J., Mernik, M., Cruz, D., Henriques, P.R.: Program Comprehension for Domain-Specific Languages. Computer Science and Information Systems (ComSIS), ISSN:1820-0214, Vol. 5, No. 2, 1--17. (2008)
21. Varanda Pereira, M.J., Mernik, M., Cruz, D., Henriques, P.R.: VisualLISA: a Visual Interface for an Attribute Grammar based Compiler-Compiler, In: Proceedings of 2nd Conference on Compilers, Related Technologies and Applications (CoRTA08), IPB, Bragança, Portugal, 265--289. (2008)
22. Henriques P.R., Pereira Varanda M.J., Mernik M., Lenič M., Gray J., Wu H.: Automatic Generation of Language-based Tools using LISA. IEE Proceedings – Software, Vol. 152, No. 2, pp. 54--69. (2005)
23. The Generic Modeling Environment [Online] Available: http://www.isis.vanderbilt.edu/Projects/gme/ (Current April, 2011)
24. Popović A.: 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 in Novi Sad. (2008)

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 75 papers, 4 books, and 30 industry projects and software solutions in the area.

Maria João Varanda Pereira, received the M.Sc. and Ph.D. degrees in computer science from the University of Minho in 1996 and 2003 respectively. She is a member of the Language Processing group in the Computer Science and Technology Center , at the University of Minho. She is currently an adjunct professor at the Technology and Management School of the Polytechnic Institute of Bragança,on the Informatics and Communications Department and vice-president of the same school. She usually teaches courses under the broader area of programming: programming languages, algorithms and language processing. But also some courses about project management. As a researcher of gEPL, she is working with the development of compilers based on attribute grammars, automatic generation tools, visual languages, domain specific languages and program comprehension. She is author or coauthor of 12 journal papers and over 36 international conference papers. She was also responsible for PCVIA project (Program Comprehension by Visual Inspection and Animation), a FCT funded national research project; She was involved in several bilateral cooperation projects with University of Maribor (Slovenia) since 2000. Now, the bilateral project underdevelopment is about ``Program Comprehension for Domain Specific Languages''.

Nuno Oliveira received, from University of Minho, a degree in Computer Science (2007) and a M.Sc. in Informatics (2009), for his thesis “Improving Program Comprehension Tools for Domain Specific Languages”. He is a member of the Language Processing group at CCTC (Computer Science and Technology Center) , University of Minho. He participated in several projects with focus on Visual Languages and Program Comprehension. Currently, he is starting his PhD studies on Architectural Reconfiguration of Interacting Services, under a research grant funded by FCT.

Daniela da Cruz received a degree in “Mathematics and Computer Science”, at University of Minho (UM), and now she is a Ph.D. student of ”Computer Science” also at University of Minho, under the MAPi doctoral program. She joined the research and teaching team of “gEPL, the Language Processing group” in 2005. She is teaching assistant in different courses in the area of Compilers and Formal Development of Language Processors; and Programming Languages and Paradigms (Procedural, Logic, and OO). She was also involved in several research projects (CROSS, DSLpc, PCVIA).

Pedro Rangel Henriques got a degree in “Electrotechnical/Electronics Engineering”, at FEUP (Porto University), and finished a Ph.D. thesis in “Formal Languages and Attribute Grammars” at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the “Language Processing group” at CCTC (Computer Science and Technologies Center). He teaches many different courses under the broader area of programming: Programming Languages and Paradigms; Compilers, Grammar Engineering and Software Analysis and Transformation; etc. Pedro Rangel Henriques has supervised Ph.D. (11), and M.Sc. (13) thesis, and more than 50 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; code analysis, program visulaization/animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He is co-author of the “XML & XSL: da teoria a pr´ atica” book, published by FCA in 2002; and has published 3 chapters in books, and 20 journal papers.

Received: December 29, 2010; Accepted: May 19, 2011.

DOI: 10.2298/CSIS110112010P

UML Profile for Specifying User Interfaces of Business Applications

Branko Perišić, Gordana Milosavljević, Igor Dejanović, and Branko Milosavljević

University of Novi Sad
Faculty of Technical Sciences
{perisic,grist,igord,mbranko}@uns.ac.rs

Abstract. This paper presents an approach to automatic user interface code generation that is based on our own HCI standard that defines layout and behaviour of coarse-grained objects for enterprise business applications. A domain-specific language (in the form of a UML profile) based on the concepts introduced by the HCI standard facilitates efficient modeling and generation of fully-functional UIs. Being a regular UML extension, this language can be used in any general-purpose UML modelling tool and can easily be integrated with other UML-based models of the application.

Key words: user interface, code generation, MDA, UML profile

1. Introduction

Various aspects of model-based development of user interfaces (UIs) are the subject of intensive research efforts. However, the majority of presented solutions is hardly applicable to development of real-world information systems because too much time and effort is spent on developing and synchronising different types of user interface models (for example, presentation model, content model, navigation model, interaction model), the complexity of sharing the knowledge embedded in different models, the lack of support in development tools, and the lack of consensus over which types of models best describe UIs [23].

Most tools for modelling user interfaces use its own set of notations, thus impeding the integration with other application models [26]. This is especially the problem in developing business applications that require tight integration of UI models with models that specify business logic.

In order to overcome the problem of integration and to facilitate the exchange of information among different tools, UML can be used to model all aspects of an application, including the user interface [3]. Although very powerful, UML without extensions is not suitable for modelling UIs [26, 3, 19].

This paper presents a UML extension in the form of a UML profile for specifying UIs of business applications named EUIS (*Enterprise User Interface Specification* profile). EUIS is developed in order to enable rapid user interface modelling at a high level of abstraction. EUIS is based on our own HCI (human-computer interaction) standard of a business application that defines functional and presentational features of coarse-grained building blocks thus enabling the generation of a fully functional UI, without the need for defining a multitude of models used for developing UIs in the general case.

It is important to note that the UI model *is not* a model of an application (from the implementation standpoint); it defines the structure of the application using building blocks at a high abstraction level (different types of screen forms, reports, procedures) and their relationships. Depending on the development platform, the intended application architecture, and the implementation of a code generator, one class from the UI model may be mapped to one or more classes or modules of an application, or may even be not mapped to the program code at all but to application repository data instead, if a data-driven application architecture is used (for example, see [13, 14]).

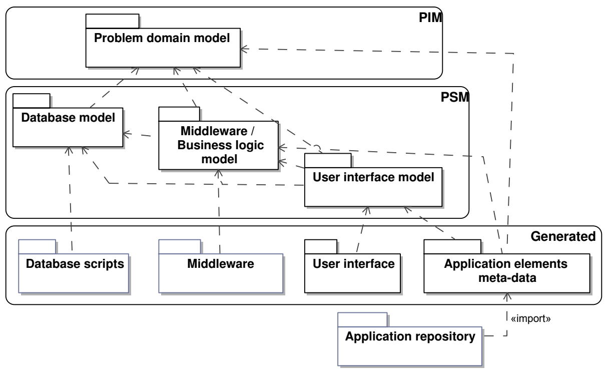


Fig. 1. Model transformations

The development of a whole business application using the EUIS profile comprises the following activities (see Figure 1):

- The development of PIM (platform independent model) of a problem domain by means of class diagrams in a general-purpose UML modelling tool.
- The automatic transformation of a PIM to PSMs (platform-specific model): database schema model, user interface model, and the middle-tier model (in the case a three-tier architecture is chosen).
- Automatic generation of artifacts needed for implementation based on PSMs: database schema creation or alteration scripts, middle-tier implementation artifacts (such as EJBs), fully functional application UI (depending on the target architecture of the client application), and atomic “CRUD” transactions implementing creation, retrieval, update, and deletion for every entity in the persistence layer.

The rest of the paper is structured as follows. Section 2 describes the basics of the HCI standard. Section 3 presents the EUIS profile. Section 4 reviews the related work. The last section concludes the paper.

2. The HCI Standard

Our human-computer interaction (HCI) standard is aimed at defining functional and visual features of course-grained application components. Its goals include the following: simplicity of use, quick user training, and the automation of user interface construction.

The papers [16, 17] define a number of types of screen forms. For this discussion, the following types are relevant:

– standard data management form,
– standard panel,
– parent-child form, and
– many-to-many form.

Standard form is designed to display data and all available operations so the user can choose a data item and invoke an operation on it without memorising commands (the object-action approach [24]). Standard operations common to all entities are represented by buttons/icons at the top of the form, while specific operations (if they exist) are represented by links/buttons at the right hand side. The standard form layout is presented in Figure 2.

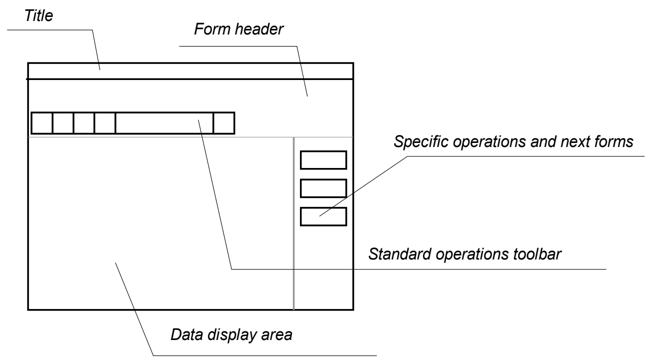

Title
Form header
Specific operations and next forms
Standard operations toolbar
Data display area

Fig. 2. Standard form layout

Operations common to all entities include search (query by form), display, addition, update, removal, copying, data navigation and view mode toggle (grid view or single record view). Specific operations include complex data processing procedures associated with the given entity (transactions), invocation of related (*next*) screen forms, and invocation of reports. The standard mandates that the specific operations always use the currently selected (viewed) record.

Standard panel has the appearance and the behaviour of the standard form but, instead being shown in its own window, it is used as an element of a complex form. Standard panels are regularly used for parent-child and many-to-many forms.

A **parent-child form** is used for data that have hierarchical structure, where each element in the hierarchy is modelled as an entity in the persistence layer. Each element in the hierarchy is represented by a standard panel, where a panel at the n-th hierarchy level filters its content according to the selected data item at the level $n-1$.

The **many-to-many form** is used for intensive management of data belonging to entities connected by "many-to-many" relationships, with or without associate classes. Its layout is presented in Figure 3. This screen form is used as follows:

– A number of desired records are selected in the upper panel. These records are "dragged" to the lower panel by clicking the button with the downwards arrow. If a record is dragged by mistake, it can be revoked back by clicking the upwards arrow button.
– The values of non-key attributes of a record selected in the lower panel may be changed.

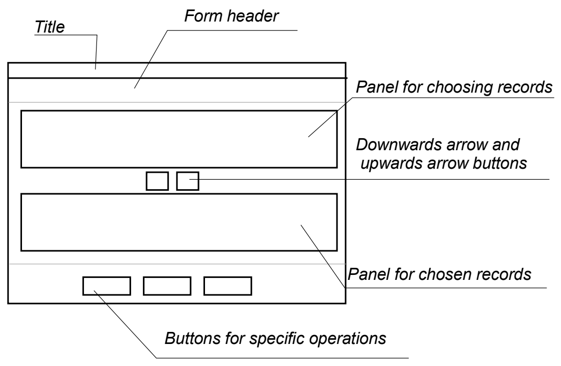

Title
Form header
Panel for choosing records
Downwards arrow and upwards arrow buttons
Panel for chosen records
Buttons for specific operations

Fig. 3. Many-to-many form layout

Relationships among screen forms are represented by three mechanisms: *zoom*, *next*, and *activate*. The **zoom** mechanism represents the invocation of the form associated with the given entity where the user can choose a data item and "drag" it (pick its values) to the fields of the previously viewed form.

The **next** mechanism, invoked from the form associated with the current entity, displays the form associated with the child entity with its data filtered so that only connected objects are displayed. The key or a representation of the parent entity is displayed in the form header, so the user easily recognises the current context. A *next* can be invoked by menu items, buttons, or links.

The **activate** mechanism enables direct invocation of a form by another form, without restrictions on the data displayed. The invoked form does not need to be related to the current one.

3. The EUIS Profile

The EUIS profile extends the following metaclasses from the *UML::Kernel* package: *Element*, *Class*, *Property*, *Operation*, *Parameter*, *Constraint*, and *Package*. It is complementary to the profile for modelling persistent data that is available in a majority of modelling tools (see Figure 4). Therefore, EUIS is independent on modelling tools, persistence layer and the database of choice. Profile that models persistent data comprises only the stereotypes present in the majority of modelling tools (possibly under a different name): persistent class, persistent property, persistent data type, and persistent operations (methods implemented in the persistence layer) – see Figure 5. When using the EUIS profile, these stereotypes are replaced with concrete stereotypes of the chosen modelling tool.

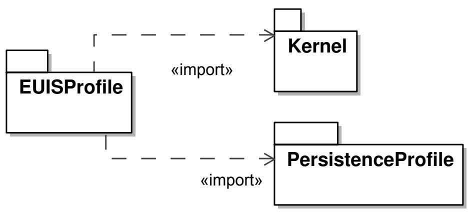


Fig. 4. Profile structure

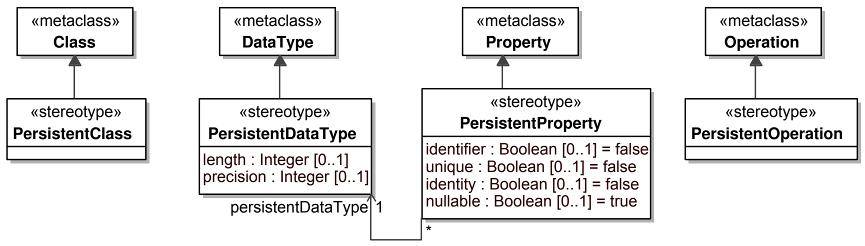

«metaclass» Class
«metaclass» DataType
«metaclass» Property
«metaclass» Operation
«stereotype» PersistentClass
«stereotype» PersistentDataType
length : Integer [0..1]
precision : Integer [0..1]
persistentDataType 1
«stereotype» PersistentProperty
identifier : Boolean [0..1] = false
unique : Boolean [0..1] = false
identity : Boolean [0..1] = false
nullable : Boolean [0..1] = true
*
«stereotype» PersistentOperation

Fig. 5. Persistence profile

In order to specify additional information needed for transforming a problem domain model to a user interface model, another profile is developed (see Figure 6) that provides the following: defining a set of one or more properties as a business key – the *BusinessKey* stereotype [4], designation of a method as a complex business procedure – the *Transaction* stereotype, and the designation of a method as a report – the *Report* stereotype.

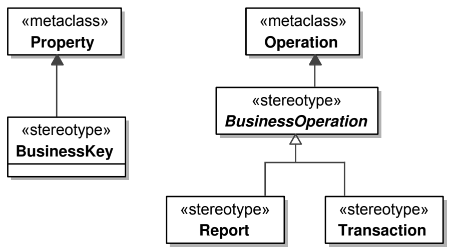


Fig. 6. Profile used in the problem domain model

Stereotypes and enumerated types of the EUIS profile are organised in the following categories:

- a visible element: extension of *Element* metaclass
- visible classes (panels): extensions of *Class* metaclass
- visible properties: extensions of *Property* metaclass
- visible methods: extensions of *Operation* metaclass
- visible parameter: extension of *Parameter* metaclass
- a group of elements: extension of *Property* metaclass
- visible association ends: extensions of *Property* metaclass
- validators: extension of *Constraint* metaclass
- a business subsystem: extension of *Package* metaclass

Due to space constraints, the rest of the section presents only the most important stereotypes and tags. Formal OCL constraints are not presented.

3.1. Visible Elements

Stereotype *VisibleElement* (see Figure 7) represents a model element that is mapped to a user interface element in the generated application. Since *Element* metaclass is a common superclass of all UML metaclasses, this facilitates the representation of all model elements with an UI component and a label, where applicable.

The enumerated type *ComponentType* defines a set of available UI component types. The set of components is designed to be platform-independent. Mapping these values to particular UI components of the chosen development platform is performed in the application generator.

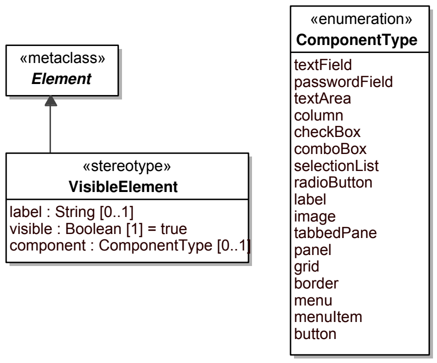


Fig. 7. Visible element

3.2. Visible Classes

Stereotype *VisibleClass* (see Figure 8) represents a class that is mapped to a panel (a regular or a tabulated panel) in the application UI. If a panel is associated to an empty window or a web page, it becomes a screen form that can be independently activated (opened).

Stereotype *VisibleClass* is not abstract because of the possibility of modelling specific panels that are not comprised by the HCI standard, but still occur rarely enough that there is no need to extend the HCI standard with a new element.

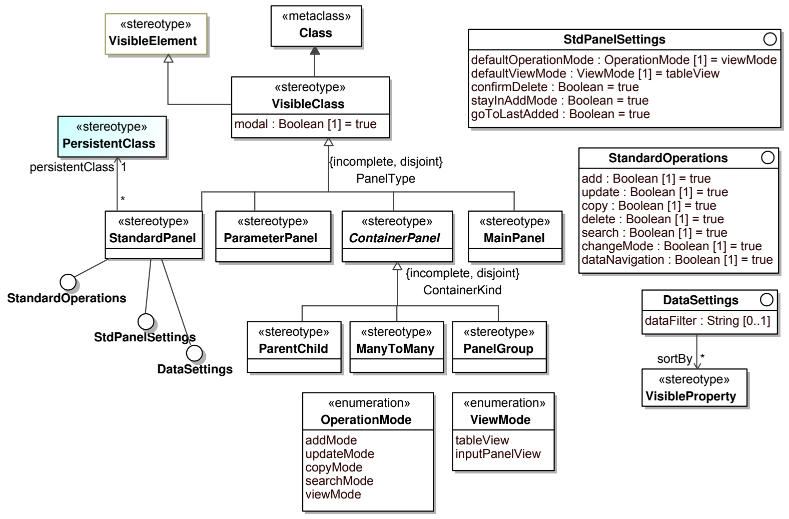


Fig. 8. Visible classes

The inherited tag *label* is used as a window title or a label that explains the purpose of the panel if displayed within a complex panel.

Stereotype *StandardPanel* denotes that the given persistent class is associated with a standard panel whose layout and behaviour are defined by the HCI standard. The standard panel implements three interfaces: *StandardOperations* – operations defined by the HCI standard: add, update, copy, delete, search, change mode, navigate data; *StdPanelSettings* – settings that define panel's runtime behaviour; and *DataSettings* – defines data filtering and sorting.

Stereotype *ParameterPanel* represents a class that is mapped to a panel for entering parameters for a visible method (see *VisibleOperation* stereotype) that is invoked by a button or a menu item. Since the majority of parameter panels in an application is created implicitly, as a result of a visible method and its parameters, classes with this stereotype rarely occur. It can be used in situations where a user successively invokes a number of methods with the same set of parameter values.

The *ContainerPanel* is an abstract stereotype that represents a complex panel that can contain other panels (simple or complex), as well as a number of properties and methods. It defines additional attributes, methods, and constraints for its descendants (*ParentChild*, *ManyToMany*, and *PanelGroup*). The layout and behaviour of *ParentChild* and *ManyToMany* panels is defined by the HCI standard, while their relationship to the contained panels is defined by hierarchical relationships (associations with ends having the *Hierarchy* stereotype). For details on associating panels, see section 3.7.

The layout and behaviour of a *PanelGroup* is not defined by the HCI standard. It is used for modelling special-purpose complex panels. The class with a *PanelGroup* stereotype defines only the contained elements, while their relationship is implemented in application code.

Classes with the *MainPanel* stereotype are used for modelling the main form of a business subsystem (see section 3.9).

3.3. Visible Properties

Stereotype *VisibleProperty* (see Figure 9) is a property of a "visible" class and is mapped to a UI component contained in the panel associated to the class. Its tags provide customisation of appearance and behaviour of the UI component, or the table column in the case of tabular display of data (*label*, *columnLabel*, *dataFormat*, *disabled*), default values in the UI component (*default*, *defaultValueGetter*), and automatic focus traversal (*autoGo*). Tag *default* contains an OCL expression that defines the initial value, while *defaultValueGetter* contains the reference to the method used for fetching the default value (in cases when OCL expression cannot be used). Tag *representative* indicates that the given property can be used to represent the whole class from the users' point of view (for example, company name, first name + " " + last name).

Aggregated represents an aggregated property, whose value is calculated using one of the aggregation functions (min, max, sum, avg, count) over the

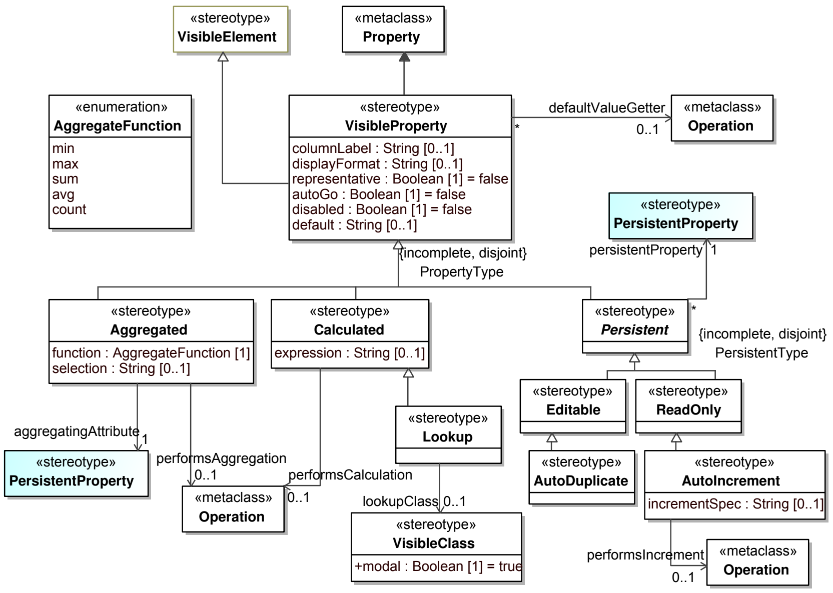


Fig. 9. Visible properties

selected property (*aggregatingAttribute*). The set of values being aggregated is specified by an OCL expression (*selection*) or by a method (*performsSelection*).

Stereotype *Calculated* represents a property whose value is calculated according to the given formula over the values in objects of this or some other class. Calculation method can be specified by an OCL expression (*expression*) or by a method (*performsCalculation*).

Abstract stereotype *Persistent* represents a property that is mapped to a persistent property in the problem domain model. Its descendants include *Editable* (enables editing the value of the persistent property in the UI component) and *ReadOnly* (disables editing). Editing values is allowed if the user has appropriate permissions.

Editable has an *AutoDuplicate* descendant that represents a persistent property where the value entered in the UI component is kept as default when entering a new record. It is usually applied to properties whose values are repeated across many records, so the user is spared some effort while entering data.

ReadOnly has an *AutoIncrement* descendant that denotes a persistent property whose value is automatically incremented with each new record entered. Contrary to identity columns or database sequences, this property allows the counter value to be reset if a condition is met (using an OCL expression in *incrementSpec* or a method in *performsIncrement*).

Stereotype *Lookup* describes a property whose value is formed from property values of referenced objects, directly or indirectly. Direct reference means that there is an association with the class that provides the data; indirect reference means that such class can be reached by traversing a series of associations. Properties forming a lookup can be specified as an OCL expression (*expression*) or by specifying the class that provides the data. In the latter case, the representative property of that class is used.

3.4. Visible Parameters

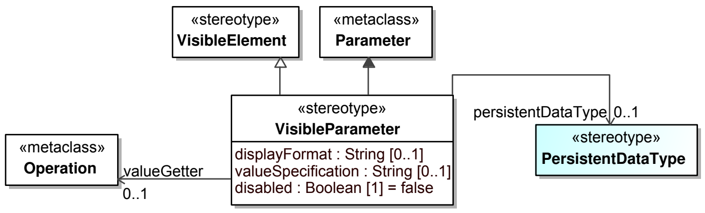


Fig. 10. Visible parameters

Stereotype *VisibleParameter* (see Figure 10) denotes a parameter of a visible method (having the *VisibleOperation* stereotype) that behaves as follows. If it is an input or an input/output parameter, then

– it enables entering parameter values by means of a UI component contained in the parameter panel associated with a visible method, or
– it defines the way of fetching the parameter values in the case when the user is not supposed to enter its value (using tag *valueSpec* contains an OCL expression that calculates the value, or tag *valueGetter* that specifies the method for calculating the value).

If it is an output parameter or a method result, it enables the display of its value by means of a UI component contained in the parameter panel associated with a visible method.

3.5. Groups of Elements

Stereotype *ElementsGroup* (see Figure 11) represents an attribute of a class with the *VisibleClass* stereotype used for grouping its elements (properties, methods, associations), thus forming semantic groups that map to groups of UI components in a panel associated with the class. Each group can define the following: an ordered collection of contained elements (tag *element*), the UI element orientation in layout (*orientation*), the location of the group in the panel (*location*), and the alignment of elements in the group (*alignment*).

The inherited tag *label* represents a label displayed in a UI component associated with the group (frame title, panel title, name of the menu item that opens a submenu).

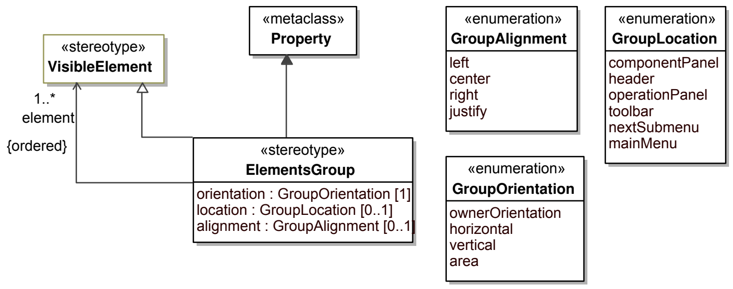


Fig. 11. Groups of elements

3.6. Visible Methods

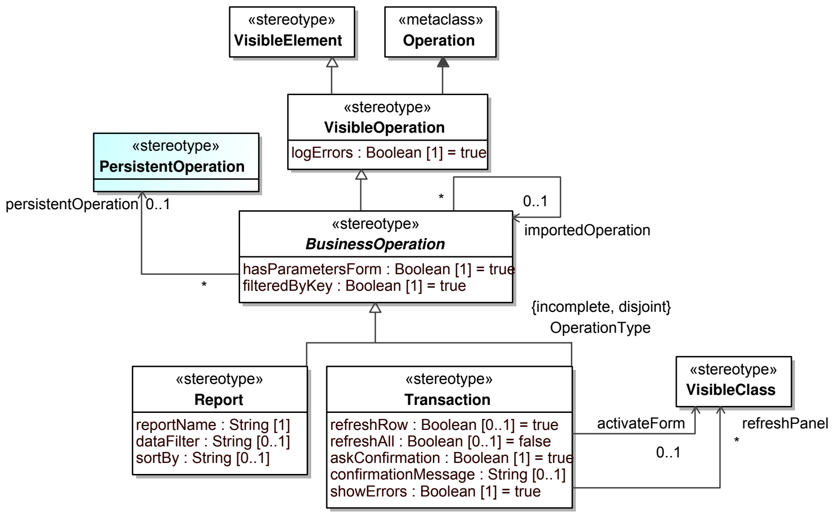


Fig. 12. Visible methods

Stereotype *VisibleOperation* (see Figure 12) denotes the method of a visible class that has an associated UI component (a button or a menu item) that enables its invocation by the user. If the method has input parameters, they must have the *VisibleParameter* stereotype (see section 3.4).

Abstract stereotype *BusinessOperation* represents a method that is mapped to an activity in the problem domain. Its descendants are *Report* and *Transaction*. *Report* describes a method that invokes a report created by one of the reporting tools. *Report*'s tags enable specifying the report name, and the filtering and sorting criteria. *Transaction* represents a complex business transaction that is implemented as a stored procedure in the database or a method in a middle tier. Its tags enable specifying the UI behaviour immediately before and after its invocation (requesting the confirmation from the user, display refresh mode, error display mode, etc).

3.7. Visible Association Ends

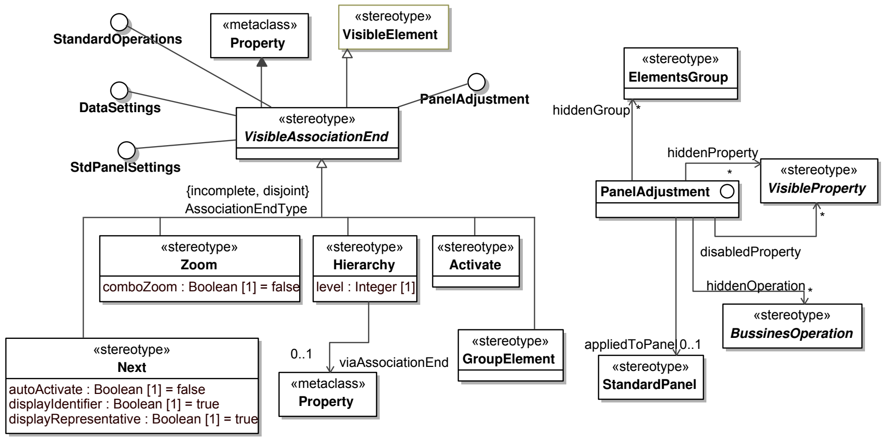

StandardOperations
«metaclass» Property
«stereotype» VisibleElement
PanelAdjustment
DataSettings
«stereotype» VisibleAssociationEnd
StdPanelSettings
{incomplete, disjoint} AssociationEndType
«stereotype» Zoom
comboZoom : Boolean [1] = false
«stereotype» Hierarchy
level : Integer [1]
«stereotype» Activate
«stereotype» Next
autoActivate : Boolean [1] = false
displayIdentifier : Boolean [1] = true
displayRepresentative : Boolean [1] = true
0..1
viaAssociationEnd
«metaclass» Property
«stereotype» GroupElement
«stereotype» ElementsGroup
hiddenGroup
*
hiddenProperty
*
«stereotype» VisibleProperty
PanelAdjustment
*
disabledProperty
hiddenOperation
*
«stereotype» BussinesOperation
appliedToPanel
0..1
«stereotype» StandardPanel

Fig. 13. Visible association ends

Abstract stereotype *VisibleAssociationEnd* (see Figure 13) is applied to a property belonging to a binary association between two visible classes. It defines the relationship between the panel belonging to the class that owns the property (activation panel) and the panel belonging to the class at the other end (destination panel). The nature of the relationship is determined by this stereotype's descendants. *VisibleAssociationEnd* only introduces common properties and constraints that enable destination panel to adjust its layout and behaviour to the context it is used in. For this purpose, *VisibleAssociationEnd* implements the following interfaces: *StandardOperations*, *DataSettings*, *StdPanelSettings*, and *PanelAdjustment*. Tag values specified by *PanelAdjustment* can be set for all types of panels (*VisibleClass* and its descendants), while tag values specified by *StandardOperations*, *DataSettings*, and *StdPanelSettings* can be applied to standard panels only (stereotype *StandardPanel*, see section 3.2).

If tag values are not defined at the association end, values defined at the standard panel are used. If values of tags *add*, *update*, *copy*, *delete*, *search*, and *changeMode* are set to false in the standard panel, the value set at the association end is ignored. This helps adhering to rules that are usually consequences of problem domain constraints independent of the usage context.

Stereotypes *Zoom*, *Next*, and *Activation* model the corresponding type of activation as defined by the HCI standard. Stereotype *Hierarchy* denotes that the destination panel has the role of an element in the parent-child or many-to-many panel. Role of the destination panel is set by the value of the *level* tag. For many-to-many complex panels, $level = 1$ is the panel that represents the header, $level = 2$ is the panel for choosing data, and $level = 3$ is the panel that contains the transferred data (for example, see class *PickAuthors* in Figure 17). For parent-child complex panels, $level = 1$ is the standard panel being the root of the tree, $level = 2$ is the child panel, $level = 3$ is the child of the child panel, and so forth: for $n > 2$, $level = n$ is a panel that is the child for panel at $level = n - 1$ (for example, see class *JournalPaperComposite* in Figure 17).

Composing parent-child and many-to-many complex panels requires defining only levels of hierarchy for each contained panel; runtime association of panels is performed by analysing their associations. If two or more associations exist between two panels, or there is a recursive association, association end to be used must be explicitly stated in the *viaAssociationEnd* tag.

Stereotype *GroupElement* denotes that the destination panel is an element of a complex panel, where its role and behaviour are defined in the application code and/or using values of tags inherited from *VisibleAssociationEnd*.

3.8. Validator

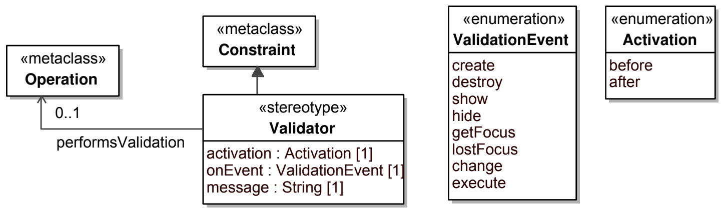


Fig. 14. Validator

Stereotype *Validator* (see Figure 14) is used to model constraints imposed by problem domain rules. Constraints are defined either as OCL expressions (tag *specification*), or as methods (*performsValidation*). The *activation* tag specifies whether the validation should be performed before or after the occurence of the selected event (*onEvent*). Tag *message* contains a human-readable message displayed in the case the constraint is not met.

3.9. Subsystem

Stereotype *BusinessSubsystem* (see Figure 15) represents an extension of the *Package* metaclass used for defining business subsystems. Every business

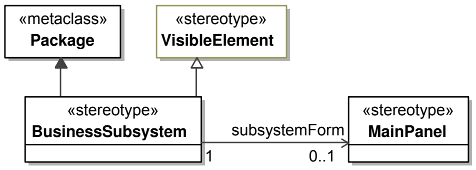


Fig. 15. Business subsystem

subsystem can have a main form that contains a menu structure for the given subsystem.

3.10. Example

Figure 16 presents a domain model of a part of a CERIF-compliant research management system presented in [17]. All classes and attributes in this model are persistent, but their stereotypes are not displayed for the sake of brevity.

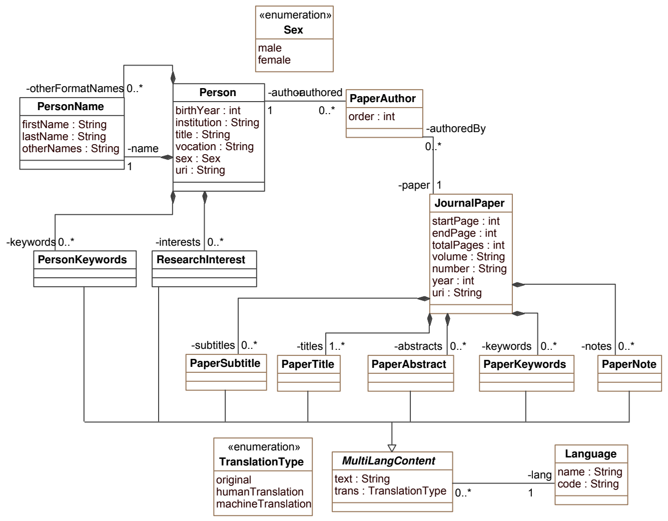


Fig. 16. A domain model of a part of CERIF-compatible system

The problem domain model in Figure 16 is automatically transformed into the UI model presented in Figure 17. Persistent classes from the domain model were mapped to UI classes with *StandardPanel* stereotype, persistent properties to UI properties with *Editable* stereotype, association ends with cardinality 0..* to UI association ends with *Next* stereotype, and association ends with cardinality 0..1 or 1 to UI association ends with *Zoom* stereotype. This was an initial version of the UI model.

The application developer manually changed this version to meet the users' requirements. The diagram in Figure 17 shows manually added classes *Pick-Authors* (a many-to-many form for choosing paper authors) and *JournalPaper-Composite* (a parent-child form for managing journal papers) with corresponding associations. Properties with *Lookup* and *ElementsGroup* stereotypes in all classes are also manually added.

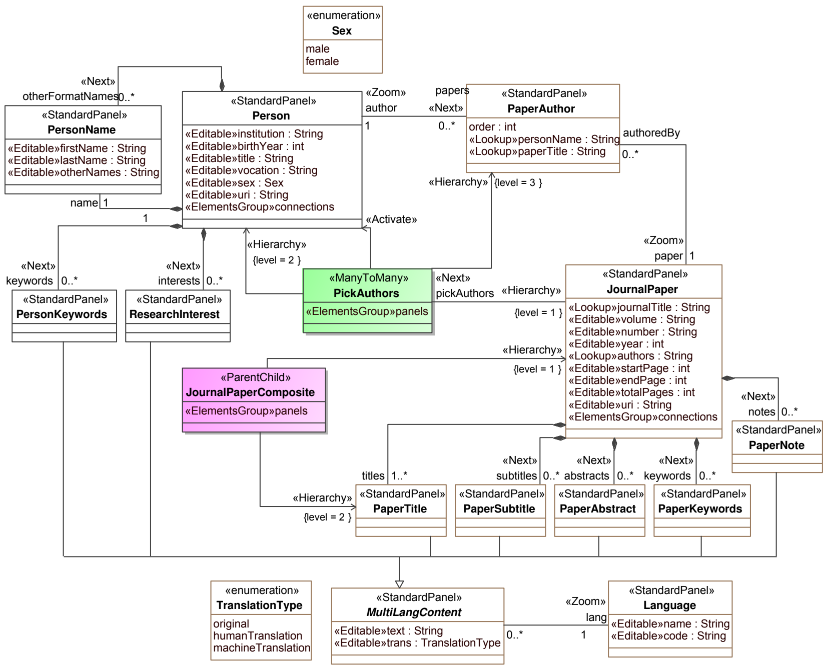
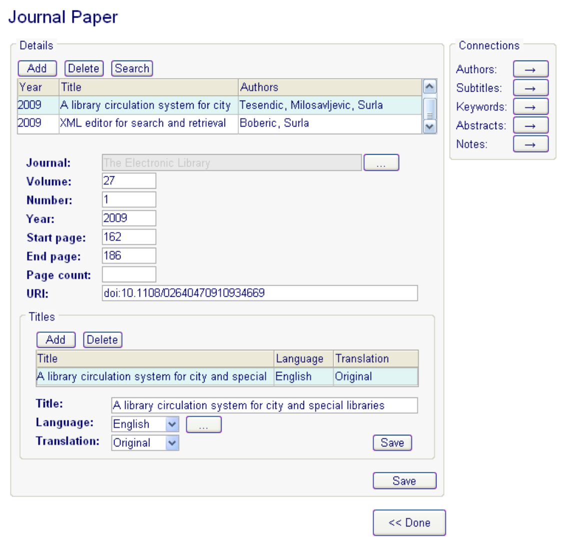

«enumeration» Sex
male
female
«Next» otherFormatNames 0..*
«StandardPanel» PersonName
«Editable»firstName : String
«Editable»lastName : String
«Editable»otherNames : String
name 1
1
«StandardPanel» Person
«Editable»institution : String
«Editable»birthYear : int
«Editable»title : String
«Editable»vocation : String
«Editable»sex : Sex
«Editable»uri : String
«ElementsGroup»connections
«Zoom» author 1
papers «Next» 0..*
«StandardPanel» PaperAuthor
order : int
«Lookup»personName : String
«Lookup»paperTitle : String
authoredBy 0..*
«Hierarchy» {level = 3 }
«Activate»
«Hierarchy» {level = 2 }
«Next» keywords 0..*
«Next» interests 0..*
«StandardPanel» PersonKeywords
«StandardPanel» ResearchInterest
«ManyToMany» PickAuthors
«ElementsGroup»panels
«Next» pickAuthors
«Hierarchy» {level = 1 }
«Zoom» paper 1
«StandardPanel» JournalPaper
«Lookup»journalTitle : String
«Editable»volume : String
«Editable»number : String
«Editable»year : int
«Lookup»authors : String
«Editable»startPage : int
«Editable»endPage : int
«Editable»totalPages : int
«Editable»uri : String
«ElementsGroup»connections
«Hierarchy» {level = 1 }
«ParentChild» JournalPaperComposite
«ElementsGroup»panels
«Next» notes 0..*
«StandardPanel» PaperNote
titles 1..*
«Next» subtitles 0..*
«Next» abstracts 0..*
«Next» keywords 0..*
«Hierarchy» {level = 2 }
«StandardPanel» PaperTitle
«StandardPanel» PaperSubtitle
«StandardPanel» PaperAbstract
«StandardPanel» PaperKeywords
«enumeration» TranslationType
original
humanTranslation
machineTranslation
«StandardPanel» MultiLangContent
«Editable»text : String
«Editable»trans : TranslationType
0..*
«Zoom» lang 1
«StandardPanel» Language
«Editable»name : String
«Editable»code : String

Fig. 17. A UI model of a part of CERIF-compatible system

An example of a web-based form generated for *JournalPaperComposite* is presented in Figure 18.

4. Related Work

In order to compare the EUIS profile with other profiles presented in the literature, this section reviews recent papers ranging in subject from modelling user interfaces of business applications to complete methodologies and tools for information system development, including its presentation aspects. Papers that deal with developing user interfaces in general are not discussed here.


Journal Paper

Details

Year	Title	Authors
2009	A library circulation system for city	Tesendic, Milosavljevic, Surla
2009	XML editor for search and retrieval	Boberic, Surla

Journal:	The Electronic Library
Volume:	27
Number:	1
Year:	2009
Start page:	162
End page:	186
Page count:	
URI:	doi:10.1108/02640470910934669

Titles

Title	Language	Translation
A library circulation system for city and special	English	Original

Title:	A library circulation system for city and special libraries
Language:	English
Translation:	Original

Fig. 18. Managing journal papers data

The papers [21, 22], like this paper, propose the use of the problem domain model as a starting point that is transformed into a model of the user interface. The approach to modelling of application views based on the available classes (various complex panels, navigation among them) is also similar. However, [21, 22] introduces more transformation levels in order to achieve portability across different implementation platforms while not restricting to a particular fixed set of components (we deal with portability as well, but with a limited set of components). Besides, [21, 22] require the development of an information retrieval model in order to implement fetching of data used in the user interface, while we opt for implicit mapping of the user interface model to the persistence layer (the cases where implicit mapping is insufficient are defined by OCL constraints or an associated method). The authors in [21, 22] only plan the development of a tool prototype that will provide transformations of models and the application generation, while their approach is tested by manual application of transformation rules.

The papers [10, 5] present a way of automatic user interface generation based on the following: a business logic model (UML activity diagrams) and a user interface model (UML class diagrams). An activity diagram is supplied with elements of the UML profile for defining system and user actions with the specified inputs and outputs, while the class diagram that is produced from the activity diagram is supplied with elements of the UML profile for user interface specification (e.g., *ContainerElement*, *GuiElement*, *ActionElement*). The profile does not support modelling the relationships between forms (navigation is omitted). Furthermore, obtaining classes that provide management of data from the problem domain model is not specified, although their presence is assumed (the *dataProvider* attribute in the *ChoiceElement* stereotype, and the *methodURL* attribute in the *ActionElement* stereotype).

In [23], the authors propose the use of patterns for accelerating the user interface development. Those patterns are at a higher abstraction level and may be used in task models, presentation models, and component layout models. This paper also proposes the use of a number of tools that guide the designer in the choice and the application of patterns during modelling, assist in model synchronisation, and generate the user interface.

Compared to elements of the standard presented in Section 2, the patterns used in [23] represent finer-grained application components. The paper [23] does not specify neither the relationship between the problem domain model and the user interface model, nor if there is a mapping of the generated user interface to the data persistence layer (whether the generated user interface is immediately testable in the real users' environment).

The paper [27] presents a method for developing web-based information systems based on problem domain models, applications and navigations that are directly mapped to existing development frameworks. The mapping is provided by the UML profile named *FrameWeb* whose stereotypes correspond with the categories of the framework used, so that the development team can deliver the implementation in a straightforward way (the implementation is manual, there are no code generators used). The majority of stereotypes in the presented UML profile are aimed at the development of the problem domain model and its mapping to the persistence layer, while support for specifying the user interface is relatively modest (there are only four stereotypes that specify the type of the web page).

The series of papers [2, 8, 7, 18] presents a methodology for developing web-based information systems UWE (*UML-based Web Engineering*) that uses a UML profile for modelling hypermedia and the ArgoUWE tool that provides for definition of different application models, their transformation, and semi-automatic code generation. The UML profile provides for the creation of navigation models, navigation structure models, and presentation models. A navigation model is a class diagram that is extracted as a problem domain model subgraph and defines which web pages assigned to problem domain model classes are linked (associations among problem domain model classes are the link candidates). A navigation structure model is a consequence of the navigation model and defines the nature of links and additional elements needed to specify navigation (menus, indices, navigational contexts). The presentation model is a composition diagram that provides for sketching the layout of application elements although these sketches are not obligatory – the user interface layout is finalised during implementation.

The concept of modelling an application in UWE methodology is the closest to the proposition in this paper – in both cases, the starting point is the problem domain model expressed as a class diagram that is automatically mapped to the application model, data model, and other models needed. Thanks to this approach, there is a direct mapping of application elements to the layer that implements business logic, a feature missing in the majority of reviewed solutions. The most notable differences between UWE and EUIS approaches are the following:

– The UWE methodology and profile are focused solely on developing web-based systems, while the methods presented here can be applied to both web and “classical” information systems.
– The UWE method does not rely on an HCI standard (there is only one type of forms).
– Our approach proposes a single user interface model that defines coarse-grained application building elements, their structure and layout (using the *ElementsGroup* stereotype), and navigation among them. Sketches of forms need not be made thanks to the mechanism for intelligent component layout that forms a usable user interface according to rules and groups, and which can be further adapted during implementation.

Although not based on a UML profile, the concept of specifying GUI forms and generating the database schema and the functional prototype of the application using the IIS*Case tool [20, 11, 6] is similar to the solution presented here, apart from the order in which artifacts are implemented. Using IIS*Case, the modelling starts with specifying form types, while database schema and the prototype application are generated. Here we start with the model of the problem domain, that is used to generate the user interface model, database schema model, and the middle-tier model (in the case of three-tier architectures). After manual changes applied to these automatically obtained models, the application is generated.

Our previously implemented tools for generating UIs of business applications for various platforms are presented in [12–16, 9]. All tools are based on the HCI standard presented in Section 2, but the difference is that UI model was not generated from the domain model, but was kept as metadata in the application repository. Metadata was further customised by the Form Generator tool, which utilised this information to generate source code. Metadata in the application repository, although stored in the database or an XML file and edited by a special-purpose tool, can be considered to be a DSL (domain specific language) for the description of UIs. The UI model enriched with EUIS stereotypes is based on the same metadata, but this UML-based form is more suitable for team work of experts from different fields (developers, UI design specialists, problem domain specialists, users) during application development.

5. Conclusions

Automatic generation of UIs in the general case requires development of a number of UI models and thus needs much time and effort, often with unsatisfactory results. Synchronisation and integration among different models, is another big problem, especially in developing business applications that require tight integration of UI models with models that specify business logic.

In order to overcome the problem of integration and to facilitate the exchange of information among different tools, UML was used to model all aspects of an application. This paper presented EUIS profile, an UML profile for specifying UIs of business applications. Being a regular UML extension, this language can be used in any general-purpose UML modelling tool and can easily be integrated with other UML-based models of the application.

EUIS profile is based on our HCI standard of a business application that defines functional and presentational features of coarse-grained building blocks. Relying on this standard has enabled the rapid development of UIs for this particular type of applications at a high abstraction level, without need to develop a number of different UI models. Automatic transformation from domain to UI model additionally speed up this process.

Our previous tools developed to support the presented concepts [12–16, 9] are used for the implementation of more than 70 projects of business information systems by several different development teams. The percentage of the generated code in the overall code base (database, middle tier, UI) ranged from 81.8% to 98.2%, depending on the type of application.

The code generation tool that relies on the presented EUIS profile is implemented as a MagicDraw plugin. Although this tool is still in development, initial results show that the percentage of the generated code will increase when all elements are implemented. The current version does not support parsing OCL constraints. Since we have already implemented a dynamic general-purpose parser Arpeggio [1], the support for OCL expressions is soon to be finalised.

Acknowledgments. Research presented in this paper was supported by Ministry of Science and Technological Development of Republic of Serbia, Grant III-44010, Title: Intelligent Systems for Software Product Development and Business Support based on Models.

References

1. Arpeggio Parser, `http://code.google.com/p/arpeggio/`
2. Baumeister, H., Koch, N., Mandel, L.: Towards a UML Extension for Hypermedia Design, In: Proceedings of The Unified Modelling Language Conference: Beyond the Standard (UML 1999), France R. and Rumpe B., Eds, LNCS vol. 1723, pp. 614–629, Springer Heidelberg (1999)
3. van den Bergh, J., Coninx, K.: Using UML 2.0 and Profiles for Modelling ContextSensitive User Interfaces, In: Model Driven Development of Advanced User Interfaces, Montego Bay, Jamaica (2005)
4. Dejanović, I., Milosavljević, G., Perišić, B., Tumbas, M.: A Domain-Specific Language for Defining Static Structure of Database Applications, Computer Science and Information Systems 7(3), (2010) (in print)
5. Funk, M., Hoyer, P., Link, S.: Model-driven Instrumentation of Graphical User Interfaces, In: Second International Conference on Advances in Computer-Human Interaction, Cancun, Mexico (2009)

6. Govedarica, M., Luković, I., Mogin, P.: Generating XML Based Specifications of Information Systems, Computer Science And Information Systems 1(1), pp. 117–140 (2004)
7. Knapp, A., Koch, N., Zhang, G.: Modelling the Structure of Web Applications with ArgoUWE, LNCS vol. 3140, Springer Heidelberg (2004)
8. Koch, N., Kraus, A.: The Expressive Power of UML-based Web Engineering, In: Proc. 2nd International Workshop on Web Oriented Software Technology, pp. 105–119 (2002)
9. Komazec, S., Milosavljević, B., Konjović, Z.: XML Schema-Driven GUI Forms Environment, In: 11th IASTED Intl. Conf. Software Engineering and Applications, pp. 342–348, Cambridge, MA (2007)
10. Link, S., Schuster, T., Hoyer, P., Abeck, S.: Focusing Graphical User Interfaces in Model-Driven Software Development, In: First International Conference on Advances in Computer-Human Interaction, Saint Luce, Martinique (2008)
11. Luković, I., Mogin, P., Pavievi, J., Risti, S.: An Approach to Developing Complex Database Schemas Using Form Types, Software: Practice and Experience 37(15), pp. 1621-1656 (2007)
12. Milosavljević, B., Vidaković, M., Milosavljević, G.: Automatic Code Generation for Database-Oriented Web Applications, In: Power, J., Waldron, J. (eds): Recent Advances in Java Technology: Theory, Application, Implementation. pp. 89–97, Trinity College Dublin (2003) ISBN 0954414500
13. Milosavljević, B., Vidaković, M., Komazec, S., Milosavljević, G.: User Interface Code Generation for EJB-Based Data Models Using Intermediate Form Representations, In: Principles and Practice of Programming in Java, pp. 125–132, Kilkenny, Ireland (2003)
14. Milosavljević, B., Vidaković, M., Komazec, S., Milosavljević, G.: User Interface Code Generation for Data-Intensive Applications with EJB-Based Data Models, In: Software Engineering Research and Practice (SERP'03), pp. 23–27, Las Vegas, NV (2003)
15. Milosavljević, G., Perišić, B.: Really Rapid Prototyping of Large-Scale Business Information Systems, In: IEEE Intl. Workshop on Rapid System Prototyping, pp. 100–106, San Diego, CA (2003)
16. Milosavljević, G., Perišić, B.: A Method and a Tool for Rapid Prototyping of Large-Scale Business Information Systems, Computer Science And Information Systems 2(1), pp. 57–82 (2004)
17. Milosavljević, G., Ivanović, D., Surla, D., Milosavljević, B.: Automated Construction of the User Interface for a CERIF-Compliant Research Management System, The Electronic Library (in print)
18. Moreno, N., Melia, S., Koch, N., Vallecillo, A.: Addresing New Concerns in Model-Driven Web Engineering Approaches, In: Proc. Web Information Systems Engineering (WISE), LNCS vol. 5175, pp. 426–442, Springer Heidelberg (2008).
19. Paterno, F.: Towards a UML for Interactive Systems, In: Proc. Engineering for Human-Computer Interaction, pp. 7–18, Toronto, Canada, (2001)
20. Pavićević, J., Luković, I., Mogin, P., Govedarica, M.: Information System Design And Prototyping Using Form Types, In: International Conference on Software and Data Technologies, pp.157–160, Setubal, Portugal (2006)
21. Schattkowsky, T., Lohmann, M.: Towards Employing UML Model Mappings for Platform Independent User Interface Design, In: Model Driven Development of Advanced User Interfaces, Montego Bay, Jamaica (2005)

22. Schattkowsky, T., Lohmann, M., UML Model Mappings for Platform Independent User Interface Design, In: MoDELS 2005 Workshops, LNCS 3844, pp. 201-209, Springer, Heidelberg (2006)
23. Seffah, A., Gaffar, A.: Model-Based User Interface Engineering with Design Patterns, Journal of Systems and Software 80(8), pp. 1408–1422 (2007)
24. Shneiderman, B.: Designing the User Interface: Strategies for Effective HumanComputer Interaction, Addison-Wesley, Third Edition (1998)
25. da Silva, P.P.: User Interface Declarative Models and Development Environments: A Survey, In: Proc. Design, Specification and Verification of Interactive Systems, LNCS vol. 1946, pp. 207–226, Limerick, Ireland (2000)
26. da Silva, P.P., Paton, N.W.: Improving UML Support for User Interface Design: A Metric Assessment of UMLi, In: Workshop on Bridging the Gaps Between Software Engineering and Human-Computer Interaction at International Conference on Software Engineering (ICSE 03), pp. 76–83, Portland, Oregon, USA (2003)
27. Estêvão Silva Souza, V., Almeida Falbo, R., Guizzardi, G.: A UML Profile for Modelling Framework-based Web Information Systems, In: Workshop on Exploring Modelling Methods for Systems Analysis and Design (EMMSAD'07), pp. 149–158, (2007)

Branko Perišić is an associated professor at University of Novi Sad, Faculty of Technical Sciences. He has received his engineer diploma from University of Sarajevo, Faculty for electrical engineering, M.Sc. and PhD diplomas from University of Novi Sad, Faculty of Technical Sciences. He is currently a Computer center manager and head of Software development team at Faculty of Technical Sciences. As a teaching professor he has developed and teached a variety of Computer Engineering, Software Engineering and Information System Design courses at different Universities. His major research interests are related to Model Driven Software Development, Business Information Systems Design, Software Configuration Management and Secure Software Design.

Gordana Milosavljević is an assistant professor at University of Novi Sad, Faculty of Technical Sciences. She teaches courses in Business Information Systems and Model Driven Software Development. Her research interests focus on software engineering methodologies, rapid development tools and enterprise information systems design.

Igor Dejanović received his M.Sc. (5 years, former Diploma) degree from the Faculty of Technical Sciences in Novi Sad. He completed his Mr (2 year) degree at the University of Novi Sad, Faculty of Technical Sciences. Currently, he works as a teaching assistant at the Faculty of Technical Sciences at the University of Novi Sad, where he assists in teaching several Computer Science and Software Engineering courses. His research interests are related to Domain-Specific Languages, Model-Driven Engineering and Software Configuration Management.

Branko Milosavljević is an associate professor at University of Novi Sad, Faculty of Technical Sciences. He teaches courses in Net-Centric Computing, XML and Web Services, and Security in E-Business Systems. His research interests include information retrieval, digital libraries, document management and information security.

Received: January 12, 2011; Accepted: May 5, 2011.

DOI: 10.2298/CSIS110111015S

Formalizing Business Process Specifications

Andreas Speck[1], Sven Feja[1], Sören Witt[1], Elke Pulvermüller[2], and Marcel Schulz[3]

[1] Christian-Albrechts-University Kiel
Olshausenstrasse 40, 24098 Kiel, Germany
{aspe,svfe,swi}@informatik.uni-kiel.de
[2] University of Osnabrueck
Albrechtstr. 28, 49076 Osnabrueck, Germany
elke.pulvermueller@informatik.uni-osnabrueck.de
[3] Intershop Communications AG
Intershop Tower, 07740 Jena, Germany
marcel.schulz@intershop.com

Abstract. The behavior of commercial systems is described with business process models. There are different notations and formalism to express business processes. Many of these notations such as BPMN or ARIS EPC models are widely used in commercial projects.
In the paper we focus on formalisms to express rules and specifications for the business processes. Temporal logic in general is a suitable formalism to express rules for dynamic processes. CTL is one kind of temporal logic focusing on branches and paths in particular. With CTL it is possible to formulate rules about different paths in business processes. Since the textual formulae of CTL are not very suitable in the development of commercial systems we introduce a graphical notation (G-CTL) based on the business process notation ARIS EPC. Moreover, we add to the CTL semantics specializers to differentiate between the element types in business process models and provide wildcards which allow the user to check for unknown elements or elements with only partially known properties.

Keywords: formal business process rules, temporal logic, model checking, extended graphical-CTL.

1. Introduction

Business process models are used to describe the behavior of commercial systems. There are different notations of business process models. Especially the formal business process models may be the base of an automated checking. In order to reach this goal we need also formalisms to express the rules or specifications for the business processes.

Temporal logic in general and the **C**omputational **T**ree **L**ogic (CTL) [7] in particular are promising in order to express business rules and temporal dependencies of business processes. Since CTL focuses on the branches and paths of processes it allows to distinguish between elements in different paths. Moreover, there are well established checking tools like the **S**ymbolic **M**odel **V**erifier (SMV) [39] which may be applied or at least serve as base for extensions [43].

In the following section 2 we examine business process characteristics and the business process modeling. In the following section 3 first CTL as basic notation is introduced and the more suitable graphical notation G-CTL is presented. In section 4 the extensions of the G-CTL (specializers and wildcards) are motivated by checking examples and then presented. Section 5 gives an overview about the related work.

2. Business Process Models

2.1. Business Process Modeling

There are different types of business process models. For instance, BPMN (**B**usiness **P**rocess **M**odel and **N**otation), ARIS (**Ar**chitecture of integrated **I**nformation **S**ystems) or UML Activity Diagrams are well-known approaches supporting the modeling of business systems in general. However, the basic expressiveness of these model types is quite similar. They provide functions (or activities) and events, various types of connections (like control or sequence flows or associations), splits and joins (mostly combined with logic operators such as `AND`, `OR` or `XOR`) and different elements for further information. This convergence of business process model types allows reducing the models to a formal nucleus: an automaton model. Business process models may be transformed or reduced to states and transitions between the states [38]. Furthermore, such automaton models may be subject of automated checking. Two typical approaches for such transformations may be found in [3] and [42]. [3] transforms the business processes to Petri nets, followed by a transformation into Kripe structures which are then checked. [42] transforms the business processes directly into Kripke structures. The Kripke structures are a base for model checking.

In this paper we focus on the application domain of e-commerce systems in general and in particular on one of the largest standard e-commerce systems: *Intershop Enfinity*. *Intershop Enfinity* is modeled with an ARIS profile *ARIS for Enfinity* [11].

Although, Enfinity-based e-commerce systems are modeled using various model types, we concentrate on the model type mainly used to describe business processes: the **E**vent-driven **P**rocess **C**hains (EPCs). [4]

The EPCs are used to model the business processes on a specific detail level (cf. model elements description in figure 1). EPCs are more concrete than value added chains and present the business aspects of the processes very well. However, they are no concrete implementation models e.g. like UML sequence charts. The EPC models are ideal for the communication between the domain experts (economists) and the computer scientists, since they are still understood by both groups.

[4] Further model types are value added chain models or function hierarchies. These model types are not issue of the paper.

Based on the EPC models the design of the implementation may start. In the case of Intershop Enfinity, executable workflow models (called *Pipelines*) represent the design and are executed by the system's application server.

When the domain experts want to check the business process descriptions of an e-commerce system, this is generally performed on the level of EPC models. Therefore, business rules, regulations and system specific requirements which have to be implemented by the system should be verified on this business process (EPC) level, too. If the EPC models do not represent the required rules and regulations correctly then the resulting system will hardly meet the needs. Therefore it is essential to verify the EPC models.

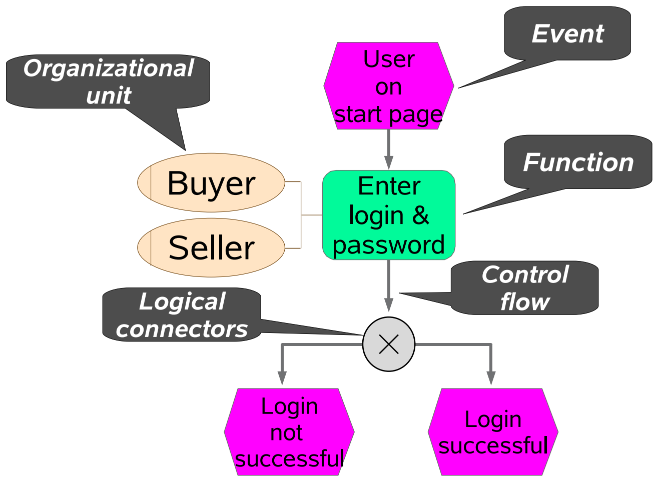


Fig. 1. Basic Elements of Event-driven Process Chains (EPCs).

The main elements in the EPC models are (cf. figure 1):

– **Functions** are considered as active elements in EPCs. They describe functionality such as tasks or activities. Functions represent transformations from one state to another, follow-up state. If different follow-up states can occur, the selection of the respective follow-up state can be modeled explicitly by *logical connectors* (as described below). Functions may be refined into another EPC (hierarchical functions). In the EPC model rounded rectangles represent functions.
– **Events** are passive elements which describe the conditions or circumstances which result from *functions* or are triggering the execution of *functions*. An *event* is represented by a hexagon.
– The **control flow** connects *events*, *functions* or *logical connectors* creating a chronological sequence and depicts the logical interdependencies between them. *Control flows* are represented by arrows.

– **Logical connectors** express the logical relationships between elements in the *control flow* (*events* and *functions*). The relationships correspond to the logical operations AND, OR and XOR. Figure 1 depicts the graphical representation of an XOR relationship. The notation elements for AND or OR are similar and with the corresponding Boolean symbol within the circle.

  An XOR in a *control flow* defines a branching point or branch, respectively. There, a decision is required which follow-up state (or path, respectively) is to be taken exclusively. The counterpart of a branch is a merge which means that different branches are merged into one. Branches as well as splits use the same symbol.

  An AND may represent the split or join in the *control flow*. A split activates the outgoing *control flows* in parallel. The join synchronizes incoming *control flows*.

  OR is the weakest relation. An opening OR connector activates one or more control flows and deactivates the rest of them. The counterpart of this is the closing OR connector which activates the *control flow* when at least one of the incoming *control flows* is activated.

Besides these EPC model elements there are several others. A further remarkable element is the **organizational unit** and its assignment which describes the connection between an organizational unit (a person or an organization responsible for a specific function) and the *function* it is responsible for.

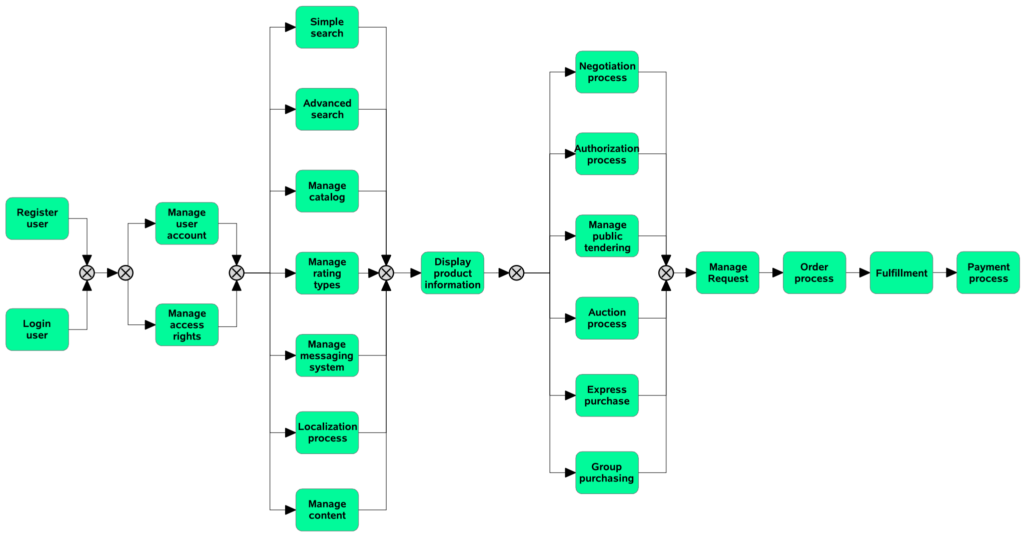


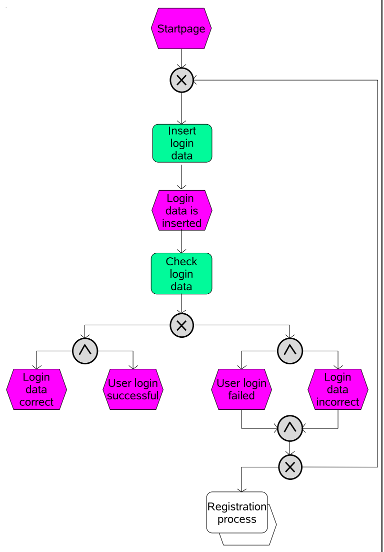
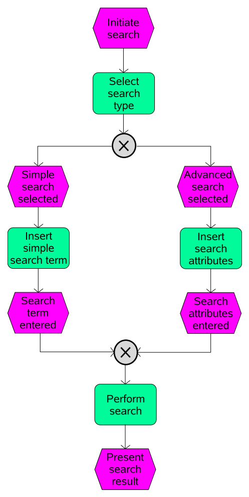
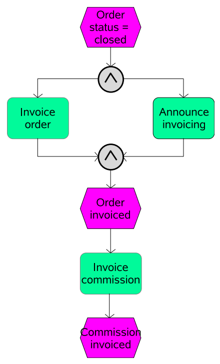
Fig. 2. Example of a Function Flow in an eProcurement System.

2.2. Commercial Information System Models

As all comparatively large systems commercial information systems may be modeled on several levels of abstraction. This approach is backed by the modeling concepts such as ARIS. Overview models are used to express the major parts of the system on a very abstract level. Figure 2 depicts an abstract function flow. Here, the details of the complete process are hidden.

These comparatively abstract process models may be further detailed and transformed into EPC models with different sub-processes. Finally, we will end up in very detailed sub-process models. In table 1 three examples of detailed sub-processes are presented which are typical for the e-commerce domain: *Login Sub-Process*, *Search Sub-Process* and *Invoice Sub-Process*.

Table 1. Detailed Business Process Patterns for eCommerce Systems.

		
Login Sub-Process	*Search Sub-Process*	*Invoice Sub-Process*

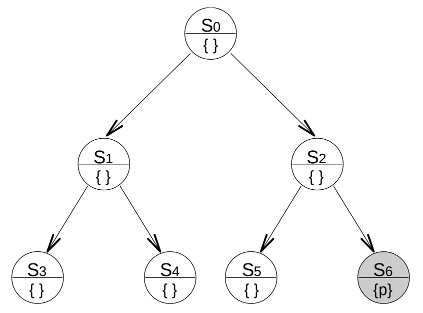
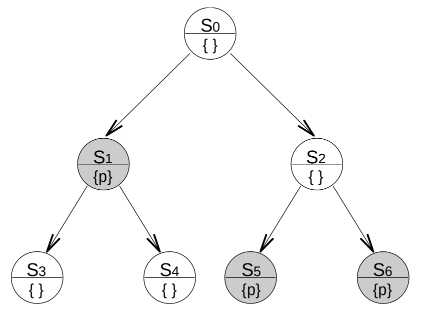
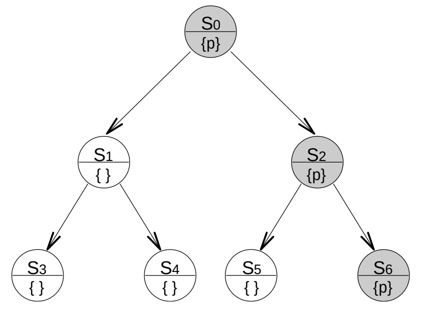
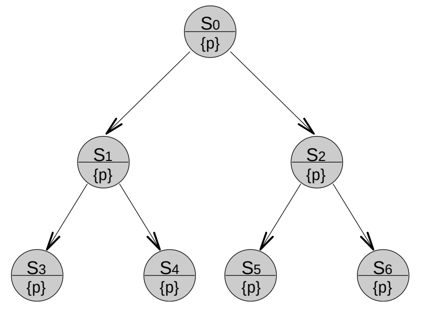
Looking at the representation of processes in a commercial information system then there are mainly two alternatives to present the model: either a large printout as a wall paper or as comparatively small models of sub-processes in a modeling tool (such as ARIS). When we look at the wall paper we have to search manually.

However, in each case it is difficult to keep the overview as well as the knowledge of the (important) details and the interactions between these details. The verification or checking of such systems is quite hard for human beings and requests automated assistance. An assistance which is able to read rules which have to be fulfilled by the commercial information system and to check the model automatically.

3. Formal Specifications

Formal specifications express the rules to be checked. In the following subsections we present temporal logic as base to represent the business rules.

Table 2. Examples of some basic CTL Operators `EF`, `EG`, `AF` and `AG` [18].

	
$s_0 \models EF\ p$	$s_0 \models AF\ p$
	
$s_0 \models EG\ p$	$s_0 \models AG\ p$

3.1. Computational Tree Logic (CTL)

Temporal logic as extension of Boolean logic may be used as formal language to express the rules. **C**omputational **T**ree **L**ogic (CTL) is the logic we use in our research.

As already mentioned CTL is based on Boolean logic:

$$\Phi ::= \bot \mid \top \mid p \mid (\neg\Phi) \mid (\Phi \wedge \Psi) \mid (\Phi \vee \Psi) \mid (\Phi \rightarrow \Psi)$$ [5]

Additionally, there are temporal operators in CTL. These operators are called *Temporal Connectives* and are used pairwise:

$$\Phi ::= AX\Phi \mid EX\Phi \mid A\,[\Phi\; U\; \Psi] \mid E\,[\Phi\; U\; \Psi] \mid AG\Phi \mid EG\Phi \mid AF\Phi \mid EF\Phi$$

A means *Always*
E means *Eventually*
G means *Globally*
X means *Next*
U means *Until*

Table 2 shows four general CTL operators. $EF\Phi$ means that Φ potentially holds. $AF\Phi$ means that Φ will occur on each potential path. $EG\Phi$ expresses that Φ is true in each states of one complete path. $AG\Phi$ is an invariant: Φ is true in each state [16].

Table 2 omits the operator pairs with *Next* and *Until* operators. Examples for these operator pairs are the following:
$EX\Phi$ means that there is a path in which in the next state Φ holds.
$AX\Phi$ means that in all paths in the next state Φ becomes true.
$E(\Phi U \Psi)$ means that there is a path in which Φ is true until Ψ holds.
$A(\Phi U \Psi)$ means that in all paths Φ is true until Ψ becomes true.

When we apply CTL for expressing rules for business processes these may look like the examples below:

- *Customer* may always *Catalog Browse*
  `AF Catalog_Browse`
  (**A**lways in the **F**uture *Catalog Browse*)
- There is a path to *Product Search*
  `EF Product_Search`
  (it **E**xists in the **F**uture *Product Search*)
- *Centralized Buyer* will get a *Personal Content* and *Personalized Offer*
  `AG (Customer_is_Centralized_Buyer ->`
  `AF (Personal_Content ∧ Personalized_Offer))`
  (**A**lways **G**lobally if *Customer is Centralized Buyer* is true implies that **A**lways in the **F**uture *Personal Content* and *Personalized Offer* will be true)
- *User not logged in* until *User login successful*
  `AG (¬User_logged_in U User_login_successful)`
  (**A**lways **G**lobally *User logged in* is false **U**ntil *User login successful*)

[5] $\Phi \rightarrow \Psi$ is a logic implication.

- *Login Data is inserted* next *Login Data Check*
  `AG (Login_Data_is_inserted -> AX Login_Data_Check)`
  (**A**lways **G**lobally *Login Data is inserted* **A**lways ne**X**t state will be *Login Data Check*)

3.2. Visualization of Graphical Specifications

Considering the examples in the previous section the CTL formula does not look very convenient. The likelihood that business modelers will accept the temporal CTL formulae is rather low. This leads us to a more suitable notation for the business modeling community: the graphical representation of CTL on base of EPC models: the *Temporal Logics Visualization Framework* (TLVF). TLVF describes how graphical specifications of the rules are derived from the business process models [23]. Our business process models are EPCs. However, the graphical notation may also be applied on other notations such as BPMN.

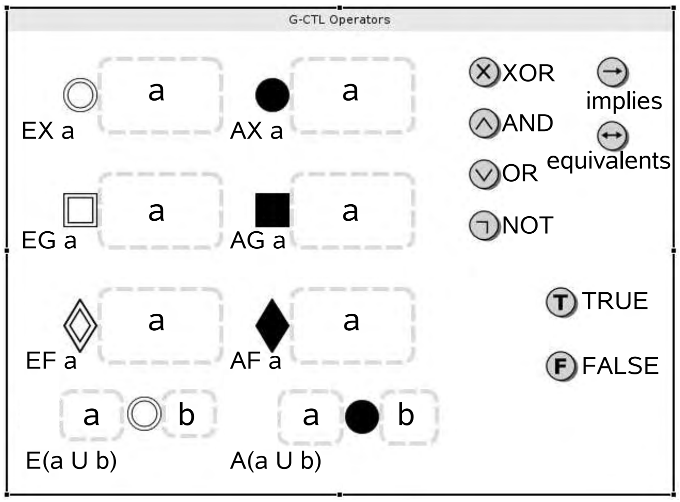
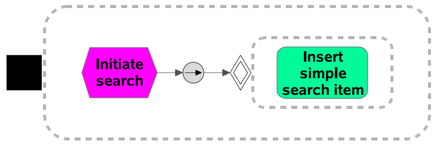
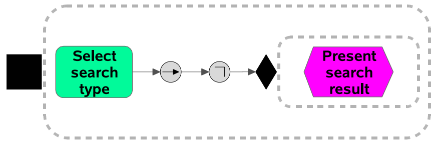


Fig. 3. G-CTL Operators.

The EPC-based definition of the TLVF language elements of **G**raphical **CTL** (G-CTL) is shown in figure 3. G-CTL operators are based on CTL operators [17], [7]. Like CTL (introduced in the previous section 3.1) there are two types of operators which are combined pairwise: Path quantifiers always (**A**) and exists (**E**) which indicate the occurrence within a path. The temporal operators determine the temporal order. The G-CTL temporal operators are: in the future (**F**), globally (**G**), next (**X**) and until (**U**). Examples for pairwise combinations are: **AG** always globally or **EX** exists next.

These operators are represented by graphical symbols which may be combined with EPC notation elements in order to describe a specification.

Table 3. Examples of G-CTL Specification Patterns in the Search Sub-process.


CTL format: $AG\ Initiate\ search\ \rightarrow\ EF\ Insert\ simple\ search\ item$

CTL format: $AG\ Select\ search\ type\ \rightarrow\ !\ AF\ Present\ search\ result$

An example of two simple rules formatted in G-CTL graphical notation are depicted in table 3. Boolean logic operators (the implication in this case) are represented by symbols which may be connected with other operators (logical or temporal logical operators) and model elements such as events or functions. Temporal logical operators like the AG (always globally) and EF (exists in the future) are realized as containers since they embrace a sub-formula or element as operand.
The informal semantics of the two rules is:

1. Rule (top of table 3): Always globally it has to be true that when the event *Initiate search* has occurred (became true) there exists in the future the function *Insert simple search item* (logical implication).
   Or in other words: When the event *Initiate search* occurred in the following of the process there must be at least one branch with the function *Insert simple search item*.
2. Rule (below): Always globally it has to be true that if the function *Select search type* has occurred (became true) always in the future the event *Present search result* becomes not true (logic implication).

Or in other words: When the function *Select search type* occurred (which means that the search function is activated by the user) in the following of the process there is a branch in which the event *Present search result* does not occur. This means that we are looking for counterexamples of actually desired behavior: We want to assure that at least an empty search result is presented to the user when s/he has initiated a search.

4. Enhanced Checking

The visualization of rules is part of the user front end. The checking algorithms of the checking system are another issue. Basically, we rely on the CTL model checking algorithms e.g. like realized in the SMV model checker [39].

However, before we are able to apply model checkers (or any other checking concepts) we have to transform the models (the EPC business process models in our case) and the specifications to the formal representations used by the checkers.

The result of a checking is either that the specification is fulfilled or violated. If the model is correct according to the specification only the notification of ”true” is reported. In an error case the model checker answers with a textual description of a counter example. The result may be presented in the format of a textual description as in [26] .

4.1. Extended Model Checking

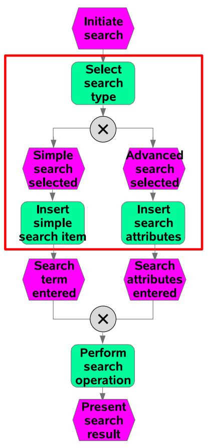
In general, model checkers need automata models as input for the checking procedure. Actually, the automata models are represented in a specific structure – the Kripke structure. Kripke structures may be considered as a specific expression of ordinary automata representations [7].

If we consider a direct transformation of the EPC models we might transform the elements **event** and **function** directly into states which are connected according to the control flow. This straight-forward approach is not necessarily wrong. In some cases it is sufficient and temporal specifications may be checked in a correct manner in compliance to the semantics.

However, there may be cases in which we would like to distinguish between the different model elements when we develop a specification. We propose an extension of the CTL notation with specializers which characterize the specific model elements.

Two examples in which a specification without distinction between the element types leads to an error are presented in figure 4. Both examples are supplemented with textual specifications. In both examples the upper specification is without additional specializers and the lower specification makes use of the two additional specializers *F* and *E* [6].

[6] These additional specializers are not to be confused with the operators *Future* and *Exists* which are always used in a pairwise manner.

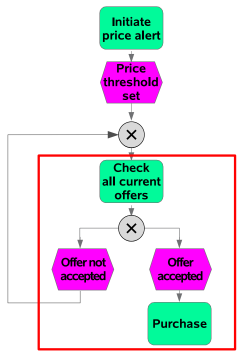


Specification without Specializer:

Select search type ->
AX (*Insert simple search item* ∨ *Insert serach attributes*) false

Spezification with Specializer:

Select search type ->
AX F[(*Insert simple search item* ∨ *Insert serach attributes*)] ✓



Specification without Specializers:

A (*Check all current offers* U *Offer accepted*) false

Specification with Specializers:

A (F[*Check all current offers*] U E[*Offer accepted*]) ✓

Fig. 4. Specializers in Temporal Logic.

1. In the example on the left upper specification (without specializers) of the model at the left requires that directly after the function *Select search type* the functions *Insert simple search item* or *Insert search attributes* have to follow. The model on which this rule is applied is the already known search example. Such a specification or rule may be defined in the situation when we would like to keep the denomination of the events after *Select search type* open (e.g. for customizing at design time) and are only interested that they are followed by standard functions (such as *Insert simple search item* or *Insert complex search item*). The second specification contains the specializer *F* (for function) directly after the *Always neXt* (AX). This indicates that only function elements have to be considered in the checking. An event (or an element of another type) is ignored. This specification is true (as we would expect it in the domain semantics).
2. The example on the right side of figure 4 contains a loop. The process is a price alert process. If a price falls below (or rises upon) a certain threshold there is a price alert and the system purchases.
   It may be of interest if the process *Check(s) all current offers* is performed until the price threshold is met and the *Offer (is) accepted*. The first specification without specializers turns out to be false although the model meets the requirement. When we use the specializers then the specification is correct. In our example *Always* the function *Check(s) all current offers* (due to

the specializer only functions are considered) *Until* the event *Offer accepted* becomes true. The event *Offer not accepted* is in the loop back to the function *Check all current offers*. Since *Offer not accepted* is an event and not a function as indicated by the specializer the checker does not care about it.

With this specializer concept in temporal logic specifications we are able to select specific element types and focus on these. This is an extension of the temporal logic CTL we call ECTL1 (Extended CTL). In order to handle the specializers the algorithm of the model checker has to be modified. A more detailed description of the modified checking algorithm may be found in [43].

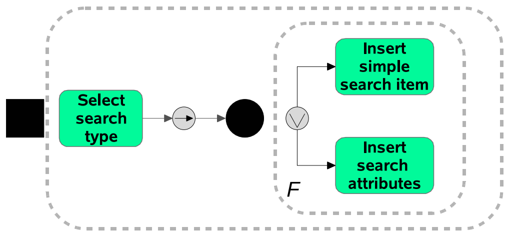


Fig. 5. Example of graphical Representation of Specializers (EG-CTL).

An example of a graphical representation of **E**xtended **G**raphical-CTL (EG-CTL) specification is depicted in figure 5. In the figure the rule of the left example in figure 4 is a little improved: An AG operator is added:
$AG\,(Select\ search\ type\ \rightarrow\ AX\ F[\,(Insert\ simple\ search\ item \vee Insert\ search\ attributes)])$

Due to the specializers the specification may be more precise. The expressiveness of the temporal logic is extended and captures different types of elements. This leads to the question to introduce uncertainty in the way of using wildcards in specifications.

4.2. Wildcards

The previous specifications of rules require that we know there must be a certain function or an event occurs at a specific moment. In other cases the explicit expected element is not clear. Figure 6 depicts an example: the payment process with some alternative payment functions. This is one possible implementation of the payment process. However, it is up to the wishes of the later shop owner which payment function is realized. E.g. in our example the payment via Pay Pal is not considered.

If we use wildcards we are able to specify rules which expect specific element types of business process models not knowing the explicit element. It is most likely that all web shops use a specific payment function or a set of

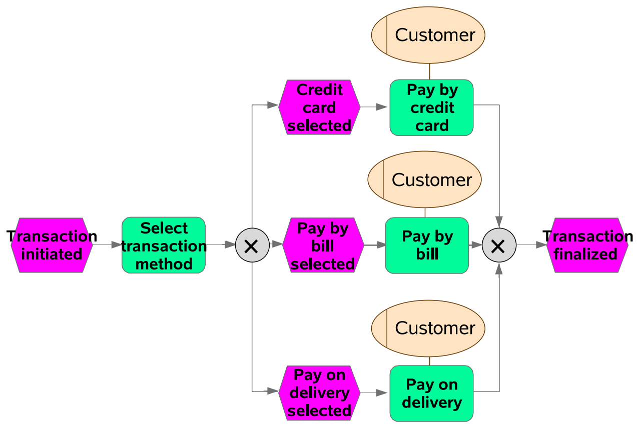


Fig. 6. Payment Process with alternative Payment Functions.

payment functions or at least in case of a free download a specific interaction function with the customer. We may express this function which is at the time of rule specification unknown by a wildcard. Of course a completely open wildcard may be critical since it may be meaningless. However, in our example we know that it is important to interact with the customer, that the function is customer driven. The relation of the unknown function to the organization customer expresses this.

The wildcard functionality may be expressed as:
$AG\ (\ E[\,Transition\ initiated\,]\ \rightarrow\ AF\ F[\ *\ \wedge\ O[Customer]\,])$

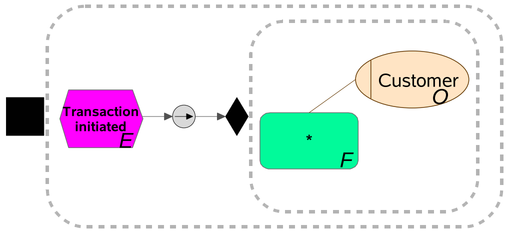


Fig. 7. Wildcard in Payment Function Rule.

The graphical representation of the rule is shown in figure 7. The specializers are indicated by a capital *E*, *F* and *O* [7]. The asterisk character symbolizes the wildcard.

[7] The *O* specializer represents an organizational unit which is by default connected to its function by a logical `AND`.

5. Related Work

Software models have been issue of verification like model checking for a rather long period. Examples for early approaches based on model checking are [21] or [39].

However, base of all the checking is the formalization of business process models like [32] (graph grammar based approach) and [8] which enable to apply formal methods for business processes. The transformation of EPC models to Petri nets also formalize them ([35] and [36]). In this case the semantics are restricted. The formalization proposed in [30] uses a fix-point-semantics-based definition of the semantics of EPCs which is also used for model checking.

The mapping of business process models on ASM (Abstract State Machines) allows to operate the business processes on an abstract level [9]. Many approaches (also the majority of the here referenced approaches) consider BPMN. However, executable models like BPEL are object of formalization as well [13]. Further formalization approaches are based on the pi calculus [37]. An issue of research to be addressed by formalization approaches are the joins after branching [8]. In the EPC model [45] the semantics as base of the formalization have been analyzed e.g. by [2] and [40]. An example for the BPMN analysis may be found at [28].

Examples for approaches employing model checking on business processes are [22], [33], [5] or [6]. [33] evaluates different checking technologies for being applied on business processes. [5] and [6] focus on the aspect of transactions in e-commerce systems. In [22] a large number of business processes have been investigated and different checking concepts are applied. One important conclusion is that several concepts could be combined in order to improve their effect.

An example of an approach for the verification of business process systems based on Petri nets is [19] using BPMN. With Petri nets the business processes are mapped to the Petri net elements similar to Kripke structure mapping. An alternative approach for Petri net based verification is based on bi-simulation and algebraic solutions (e.g. [41]).

The approach presented in this paper relies directly on the push-button model checking technology and temporal logic requirement specifications. Most approaches applying formal methods to business process models for the purpose of checking use straight-forward model transformations. These transformations result in a loss of information and, therefore, verification precision. The reason is the incompatible semantics of the business process models and the verification models which causes several problems resulting in different alternative approaches to tackle them [20]. Moreover, additional information (such as organizational units in EPC models) is lost during the transformation due to a surjective mapping. Two approaches transforming business process models to verification models are [42] (SMV Kripke structures) or [1] (Petri nets).

An approach which proposes a graphical representation of models and specifications is [26]. In this approach the business process notation are UML activity diagrams and the result of the LTL-based checking is presented in a textual manner.

In the domain of formal methods approaches may be found which concentrate on an increase of semantic expressiveness of the specification languages (e.g. the μ-calculus [10] and [34] or in the multi-valued logic research as in [15]). Extensions to the temporal logic for LTL have been proposed in [14] or [29], for instance. In these approaches a link to software models or business process models is missing and the general idea of a specialization on different model elements is not considered. In contrast to [14], [29], [27] and [31] we are able to explicitly distinguish and mix specializers (and thus views) for different model elements.

6. Conclusions and Future Work

Most business process models are very large and consist of a large number of elements and flows. The checking of these large models by hand is time consuming and still not satisfying in all cases [8].

Formal verification methods may help to automate at least some kinds of checking (e.g. routine checks). The formal method we apply in our work is model checking. The rules to be verified must be able to express the temporal relations between the process elements (e.g. the control flow).

Due to this we use the temporal logic CTL (Computational Tree Logic) as a base. However, the pure CTL has some drawbacks.
It does not support a graphical notation and the semantics of CTL may be extended. We present G-CTL as a graphical notation of CTL supporting a similar element notation than EPC models and combining them with the temporal operators of CTL.
Due to the lack of expressiveness of CTL we extend the semantics of CTL by specializers (Extended Graphical-CTL, EG-CTL). These specializers allow distinguishing between different model element types (in our case EPC events, functions, organizational units and others). With the help of the specializers it is possible to check on the existence of a specific type in the process not considering elements of other types. In order to support the specializers the checking algorithm has been modified. Moreover, we introduced wildcard which allows defining a rule in a moment when the concrete modeling of a process is not clear. The wildcards keep the position in the process open. Nevertheless, by some additional information, e.g. the knowledge which concrete organizational unit will be related to the unknown element represented by a wildcard it is possible to complete the rule.

[8] The checking of large models may result in state explosion problems. However, there are different approaches to deal with this problem. One used by most model checking tools is to optimize the model structure by applying **O**rdered **B**inary **D**ecision **D**iagrams (OBDD) [12]. Other approaches are abstraction or partial evaluation and compositional model checking [25]. These different approaches are known to the authors and taken into account. Although, these approaches are not issue of the paper.

The different techniques of the graphical representation of specifications and the extension of CTL in order to represent different model element types as well as the introduction of wildcards are integrated in the Business Application Modeler (BAM) in order to improve the usability of this business process model checking concept.

At the moment we are developing a presentation tool on the base of Eclipse the **B**usiness **A**pplication **M**odeler (BAM) [24]. In detail, BAM is based on the Eclipse Graphical Editing Framework (GEF) [4]. GEF has been chosen since it supports the required presentation functions and is comparatively portable which means that the BAM editor runs on different operating system platforms. The goal of this Eclipse-based implementation is a high degree of portability and the ability to integrate transformation and checking systems as simple as possible.

In our future work we intend to support further notations (e.g. BPMN). Already now we are working on an i* model support [44]. The interoperability between our BAM and other modeling tools like ARIS or ViFlow has not yet been realized. An intermediate (probably XML-based) data format may be useful.

References

1. van der Aalst, W.M.P.: Formalization and Verification of Event-driven Process Chains. Information and Software Technology 41(10), 639–650 (1999)
2. van der Aalst, W.M.P., Desel, J., Kindler, E.: On the semantics of EPCs: A vicious circle. In: EPK 2002 - Geschäftsprozessmanagement mit Ereignisgesteuerten Prozessketten, Proceedings des GI-Workshops und Arbeitskreistreffens (Trier, November 2002). pp. 71–79 (2002)
3. van der Aalst, W.M.P.: Formalization and Verification of Event-driven Process Chains. Information and Software Technology 41(10), 639–650 (1999)
4. Anders, E.: Modellierung und Validierung von Prozessmodellen auf Basis variabler Modellierungsnotationen und Validierungsmethoden als Erweiterung für Eclipse, Diploma Thesis (2010)
5. Anderson, B.B., Hansen, J.V., Lowry, P.B., Summers, S.L.: Model checking for design and assurance of e-Business processes. Decision Support Systems 39(3), 333–344 (2005)
6. Anderson, B.B., Hansen, J.V., Lowry, P.B., Summers, S.L.: The application of model checking for securing e-commerce transactions. Communications of the ACM 49(6), 97–101 (2006)
7. Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification – Model-Checking Techniques and Tools. Springer, Berlin, Germany (2001)
8. Börger, E., Sörensen, O., Thalheim, B.: On Defining the Behavior of OR-joins in Business Process Models. The Journal of Universal Computer Science (J. UCS) 15(1), 3–32 (2009)
9. Börger, E., Thalheim, B.: Modeling Workflows, Interaction Patterns, Web Services and Business Processes: The ASM-Based Approach. In: Proceedings of Abstract State Machines, B and Z, First International Conference (ABZ 2008). pp. 24–38. Springer LNCS 5238 (2008)

10. Bradfield, J., Stirling, C.: Modal logics and mu-calculi: an introduction. In: Handbook of Process Algebra, pp. 293–33. Elsevier Science Publishers (2001)
11. Breitling, M.: Business Consulting, Service Packages & Benefits. Tech. rep., Intershop Customer Services, Jena (2002)
12. Bryant, E., R.: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Transactions on Computers C-35(8), 677–691 (1986)
13. Cámara, J., Canal, C., Cubo, J., Vallecillo, A.: Formalizing WSBPEL Business Processes Using Process Algebra. Electronic Notes in Theoretical Computer Science 154(1), 159–173 (2006)
14. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/Event-Based Software Model Checking. In: Proceedings of the 4th International Conference on Integrated Formal Methods (IFM). pp. 128–147. Springer, LNCS 2999 (2004)
15. Chechik, M., Devereux, B., Easterbrook, S., Gurfinkel, A.: Multi-Valued Symbolic Model-Checking. ACM Transactions on Software Engineering Methodology 12(4), 371–408 (October 2003)
16. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. ACM Transactions on Programming Languages and Systems 8(2), 244 – 263 (April 1986)
17. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation. pp. 427–432 (1995)
18. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts; London, England, 3 edn. (2001)
19. De Backer, M., Snoeck, M.: Business Process Verification: a Petri Net Approach. Tech. rep., Catholic University of Leuven, Belgium (2008)
20. van Dongen, B.F., Jansen-Vullers, M., Verbeekm, H.H.M.W., van der Aalst, W.M.P.: Verification of the SAP reference models using EPC reduction, state-space analysis, and invariants. Computers in Industry 58(6), 578–601 (2007)
21. Emerson, E.A., Clarke, E.M.: Characterizing Correctness Properties of Parallel Programs Using Fixpoints. In: ICALP 1980, Automata, Languages and Programming, 7th Colloquium. pp. 169–181. Springer LNCS 85 (1980)
22. Fahland, D., Favre, C., Jobstmann, B., Köhler, J., Lohmann, N., Völzer, H., Wolf, K.: Instantaneous Soundness Checking of Industrial Business Process Models. In: Proceedings of the 7th International Conference on Business Process Management (BPM 2009). pp. 278–293. Springer, LNSC 5701 (2009)
23. Feja, S., Fötsch, D.: Model Checking with Graphical Validation Rules. In: Proceedings of the 15th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2008). pp. 117–125. IEEE (2008)
24. Feja, S., Speck, A., Witt, S., Schulz, M.: Checkable Graphical Business Process Representation. In: Proceedings of the 14th East-European Conference on Advances in Databases and Information Systems (ADBIS 2010,). pp. 176–189. Springer, LNCS 6295 (2010)
25. Fisler, K., Krishnamurthi, S.: Decomposing Verification by Features. In: IFIP Working Conference on Verified Software: Theories, Tools, Experiments (October 2005)
26. Förster, A., Engels, G., Schattkowsky, T., Van Der Straeten, R.: Verification of Business Process Quality Constraints Based on Visual Process Patterns. In: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE '07). pp. 197–208 (2007)
27. Giannakopoulou, D., Magee, J.: Fluent Model Checking for Event-based Systems. In: Proceedings of the 9th European Software Engineering Conference (ESEC) held

jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). pp. 257–266. ACM Press (2003)

28. Grosskopf, A.: xBPMN. Formal control flow specification of a BPMN based process execution language, Master's thesis (2007)
29. Jonsson, B., Khan, A.H., Parrow, J.: Implementing a Model Checking Algorithm by Adapting Existing Automated Tools. In: Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems. pp. 179–188. Springer, LNCS 407 (1989)
30. Kindler, E.: On the Semantics of EPCs: A Framework for Resolving the Vicious Circle. In: Proceedings fo Business Process Management: Second International Conference, (BPM 2004). pp. 82–97. Springer LNCS 3080 (2004)
31. Kindler, E., Vesper, T.: ESTL: A Temporal Logic for Events and States. In: Proceedings of the 19th International Conference on Application and Theory of Petri Nets (ICATPN). pp. 365–384. Springer LNCS 1420 (1998)
32. Klauck, C., Müller, H.J.: Formal business process engineering based on graph grammars. International Journal on Production Economics 50, 129–140 (1999)
33. Köhler, J., Tirenni, G., Kumaran, S.: From Business Process Model to Consistent Implementation: A Case for Formal Verification Methods. In: 6th International Enterprise Distributed Object Computing Conference (EDOC 2002). pp. 96–106 (2002)
34. Kozen, D.: Results on the propositional mu-calculus. Theoretical Computer Science 3(27), 333–354 (December 1983)
35. Langner, P., Schneider, C., Wehler, J.: Prozeßmodellierung mit ereignisgesteuerten Prozeßketten (EPKs) und Petri-Netzen. Wirtschaftsinformatik 39(5), 479–489 (1997)
36. Langner, P., Schneider, C., Wehler, J.: Petri net based certification of event-driven process chains. In: Proceedings of Application and Theory of Petri Nets 1998, 19th International Conference (ICATPN '98). pp. 286–305. Springer, LNI 1420 (1998)
37. Ma, S., Zhang, L., He, J.: Towards Formalization and Verification of Unified Business Process Model Based on Pi Calculus. In: Proceedings of the 6th ACIS International Conference on Software Engineering Research, Management and Applications (SERA). pp. 93–101. IEEE Computer Society (2008)
38. Mahleko, B., Wombacher, A.: Indexing Business Processes based on Annotated Finite State Automata. In: IEEE International Conference on Web Services (ICWS 2006). pp. 303–311. IEEE Computer Society, Los Alamitos, CA, USA (2006)
39. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
40. Mendling, J., Neumann, G., van der Aalst, W.M.P.: Understanding the Occurrence of Errors in Process Models Based on Metrics. In: On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007. pp. 113–130 (2007)
41. Morimoto, S.: A Survey of Formal Verification for Business Process Modeling. In: ICCS 2008, 8th International Conference. pp. 514–522. Springer LNCS 5102 (2008)
42. Pulvermüller, E.: Composition and correctness. Electronic Notes in Theoretical Computer Science (ENTCS) 65(4) (2002)
43. Pulvermüller, E.: Reducing the Gap between Verification Models and Software Development Models. In: The 8th International Conference on Software Methodologies, Tools and Techniques (SoMeT 2009). pp. 297–313. IOS Press (2009)
44. Rusnjak, A., El Kharbili, M., Hristov, H., Speck, A.: Managing the Dynamics of E/mCommerce with a Hierarchical Overlapping Business-Value-Framework. In: 24th

IEEE International Conference on Advanced Information Networking and Applications Workshops (AINA Workshops), WAINA 2010. pp. 461–466. IEEE Computer Society (2010)

45. Scheer, A.W.: ARIS - Modellierungsmethoden, Metamodelle, Awendungen. Springer, Berlin, Germany (1998)

Prof. Dr. Andreas Speck is head of the ”Business Information Technology” research group at Christian-Albrechts-University of Kiel, Germany. Previously he headed the research group ”Application Systems (Software Engineering and eCommerce)” at the Friedrich-Schiller-University of Jena and led the research group of the Intershop Communications AG at Jena. His main research interests are the modeling and verification of commercial application systems and electronic and mobile commerce systems. Andreas Speck is member of the German Computer Society (GI).

Sven Feja studied Business Informatics at the Friedrich-Schiller-University of Jena, Germany, and received his degree in Business Informatics in 2006. After his graduation he joined the research group ”Application Systems (Software Engineering and eCommerce)” at the Friedrich-Schiller-University of Jena. Currently he is a research assistant at the Christian-Albrechts-University of Kiel, Germany, researching in the field of business process modeling and validation and verification of correctness of process models (including functional and non-functional aspects). Sven Feja is member a member of the German Computer Society (GI).

Sören Witt studied Computer Engineering at the Christian-Albrechts-University of Kiel, Germany. He received his degree as Computer Engineer in 2009. After his graduation he joined the ”Business Information Technology” research group at Christian-Albrechts-University of Kiel as research assistant. Soeren Witt is researching in the area of business process model verification and validation on basis of graphically represented specifications.

Prof. Dr.-Ing. Elke Pulvermüller is a professor (Jun.Prof.) in the Department of Mathematics & Computer Science at the University of Osnabrueck, Germany. There, she is head of the Software Engineering research group. Between September 2009 and March 2010 she has been temporarily appointed as an Acting Full Professor of the Institute of Software Technology and Programming Languages at the University Luebeck, Germany. Previous to her appointments at Luebeck and Osnabrueck she has been a senior researcher / research assistant at the University of Luxembourg (2006 - 2007), at the Friedrich Schiller-University of Jena (Germany) and at the Universitaet Karlsruhe (Germany). She received her doctoral degree from the Friedrich Schiller-University of Jena in 2006. Her research focuses on new approaches in software and quality engineering. Elke Pulvermueller is a member of the German Computer Society (GI) and the ACM.

Marcel Schulz studied Business Informatics at the Friedrich-Schiller-University of Jena, Germany, and received his degree in Business Informatics in 2009. He gained interactional experience as project manager for commercial information systems in Shanghai, China working for American customers from 2007 till 2009. Currently he is member of the Intershop Commuincations research group. His research interests are business intelligence, data mining and simulation.

Received: January 11, 2011; Accepted: May 5, 2011.

DOI:10.2298/CSIS101231014M

An Approach to Assess and Compare Quality of Security Models

Raimundas Matulevičius[1], Henri Lakk[1], and Marion Lepmets[2]

[1] Institute of Computer Science, University of Tartu,
J. Liivi 2, 50409 Tartu, Estonia
rma@ut.ee, henri.lakk@gmail.com
[2] Centre for Public Research Henri Tudor – SSI
29 Av. John F. Kennedy, L-1855 Luxembourg,
Marion.Lepmets@tudor.lu

Abstract. System security is an important artefact. However security is typically considered only at implementation stage nowadays in industry. This makes it difficult to communicate security solutions to the stakeholders earlier and raises the system development cost, especially if security implementation errors are detected. On the one hand practitioners might not be aware of the approaches that help represent security concerns at the early system development stages. On the other hand a part of the problem might be that there exists only limited support to compare different security development languages and especially their resulting security models. In this paper we propose a systematic approach to assess quality of the security models. To illustrate validity of our proposal we investigate three security models, which present a solution to an industrial problem. One model is created using PL/SQL, a procedural extension language for SQL; another two models are prepared with SecureUML and UMLsec, both characterised as approaches for model-driven security. The study results in a higher quality for the later security models. These contain higher semantic completeness and correctness, they are easier to modify, understand, and facilitate a better communication of security solutions to the system stakeholders than the PL/SQL model. We conclude our paper with a discussion on the requirements needed to adapt the model-driven security approaches to the industrial security analysis.

Keywords: model-driven security development, modelling quality, PL/SQL, secureUML, UMLsec.

1. Introduction

Nowadays, computer software and systems play an important role in different areas of everyday life. They deal with different type of information including the one (e.g., bank, educational qualification, and health records) that must be secured from the unintended audience. Thus, ensuring system security is a necessity rather than an option. Security analysis should be performed throughout the whole system development cycle starting from the early stages (e.g., requirements engineering and system design) and leading to the late stages (e.g., implementation and testing). However this is not the case in practice [13], [32] where security is considered only when the system is about to be implemented (e.g., at implementation stage) or deployed (e.g., at installation stage). This is a serious limitation to the secure system development, since it is the early stages where security requirements should be discovered and communicated among stakeholders, security trade-offs should be considered, and security concerns should be clearly differentiated among different system aspects (e.g., data, functionality, and etc).

One possible suggestion to solve the above problem is an approach called model driven architecture (MDA). MDA provides a solution for the system development process based on models [5] that are the simplified representations of reality. Although MDA is certainly useful for the general-purpose system and software development [14], [20], [33], [34], the current state of the art gives little evidence (we identified only one study – [3]) on how model driven security (MDS) could help developers to improve the security definition and implementation process.

A part of the problem could be a lack of the systematic support to assess the security development languages both at the systems modelling and system implementation stages. In this paper we have proposed a systematic approach to evaluate quality the security models following the instantiation of the Semiotic Quality (SEQUAL) framework [15] [16]. To validate our proposal we have performed a case study (carried on at the Software Technology and Application Centre in Estonia), where we compare quality of the security model prepared using PL/SQL [9] (a procedural programming language), and quality of the security model prepared using MDS approaches, namely SecureUML [2], [19] and UMLsec [11]. All the security models define a role-based access control [8] on the *data model* provided to us by our industrial partner. Our case study results in a higher quality for the security models, created at the requirements engineering and design stages of the systems development. However we also highlight a set of requirements that are necessary to fulfil in order the MDS approaches were applicable in practice.

The structure of the remaining paper is as follows: in Section 2 we introduce the background of our research. We present the general RBAC model, the quality framework, and the approaches that help express system security concerns. In Section 3 we introduce an approach to assess quality of the security models. Next in Section 4 we illustrate the application of our proposal to evaluate quality of three languages, namely PL/SQL, SecureUML and UMLsec. Hence, we list our observations regarding model semantic, syntactic and pragmatic quality types. Finally, in Section 5 we discuss the results against the related work, and we also conclude our study.

2. Background

In this section we provide the background for our study. Firstly, we discuss the principles of the role-based access control. Secondly, we survey an evaluation framework that helps to assess model quality. Finally, we discuss development languages to represent system security.

2.1. Role-based Access Control

In this work we adapt the core role-based access control (RBAC) model [8]. This model defines a minimum set of concepts and relationships in order to define a role-based access control system. The basic concept of RBAC is that *users* are assigned to *roles*, *permissions* are assigned to *roles*, and *users* acquire *permissions* by being members of *roles*. The same *user* can be assigned to many *roles* and a single *role* can have many *users*. Similarly, for permissions, a single *permission* can be assigned to many *roles* and a single *role* can be assigned to many *permissions*.

The basic concepts of the RBAC model are illustrated in Fig. 1. The main elements of this model are *Users*, *Roles*, *Objects*, *Operations*, and *Permissions*. A *User* is typically defined as a human being or a software agent. A *Role* is a job function within the context of an organisation. Role refers to authority and responsibility conferred on the user assigned to this role. *Permissions* are approvals to perform one or more *Operations* on one or more protected *Objects*. An *Operation* is an executable sequence of actions that can be initiated by the system entities. An *Object* is a protected system resource (or a set of resources). Two major relationships in this model are *User assignment* and *Permission assignment*. *User assignment* relationship describes how users are assigned to their roles. *Permission assignment* relationship characterises the set of privileges assigned to a *Role*.

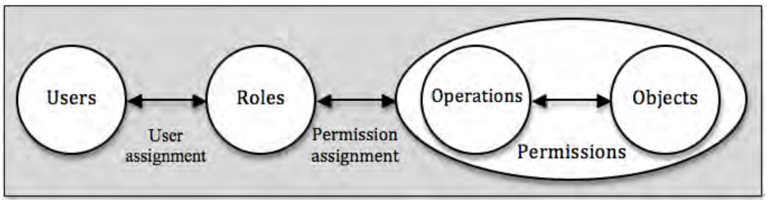


Fig. 1. Role-based Access Control Model (adapted form [8])

In Section 3 we propose an assessment of the quality for security models. There, the RBAC model suggests the criteria that help to judge about the model semantic properties as we illustrate in Section 4.

2.2. Modelling Quality

Evaluations of a model quality [30] could be performed (*i*) using detailed qualitative properties or (*ii*) through general quality frameworks. A systematic survey of these approaches could be found in [28]. In this study we combine both approaches: firstly, we follow guidelines of the *semiotic quality* (SEQUAL) framework [15], [16] to select the quality types of interest. Secondly, we identify a set of qualitative properties that are used to compare two security models.

The SEQUAL framework (Fig. 2) is an extension of the Lindland *et al*, (1994) quality framework [18], which includes discussion on syntax, semantics and pragmatics. It adheres to a constructivistic world-view that recognises model creation as part of a dialog between the participants whose knowledge changes as the process takes place. The framework distinguishes between quality goals and means to achieve these goals. *Physical quality* pursues two basic goals: externalisation, meaning that the explicit knowledge ***K*** of a participant has to be externalised in the model ***M*** by the use of a modelling language ***L***; and internalisability, meaning that the externalised model ***M*** can be made persistent and available, enabling the stakeholders to make sense of it. *Empirical quality* deals with error frequencies when reading or writing ***M***, as well as coding and ergonomics when using modelling tools. *Syntactic quality* is the correspondence between ***M*** and the language ***L*** in which ***M*** is written. *Semantic quality* examines the correspondence between ***M*** and the domain ***D***. *Pragmatic quality* assesses the correspondence between ***M*** and its social as well as its technical audiences' interpretations, respectively, ***I*** and ***T***. *Perceived semantic quality* is the correspondence between the participants' interpretation ***I*** of ***M*** and the participants' current explicit knowledge $\boldsymbol{K_S}$. *Social quality* seeks agreement among the participants' interpretations ***I***. Finally, *organisational quality* looks at how the modelling goals ***G*** are fulfilled by ***M***. In the second case the major quality types include physical, empirical, syntactic, semantic, pragmatic, social and organisational quality.

2.3. System Security

In order to define the system security policy in a systematic way it is important to understand the need for security within an organisation. One of the possible ways is to apply the security risk management process [26]. This process begins with the identification of the secure assets and the determination of the security objectives (in terms of *confidentiality*, *integrity*, and *availability*). During the next step security risks and their harm to the secured assets and their security objectives, are identified. Once the risk assessment is performed, risk treatment decisions (e.g., risk avoidance, risk reduction, risk transfer or risk retention) are taken. Following these decisions, the developers formulate the security requirements in order to mitigate the identified risks. Security requirements are, finally implemented into the security controls.

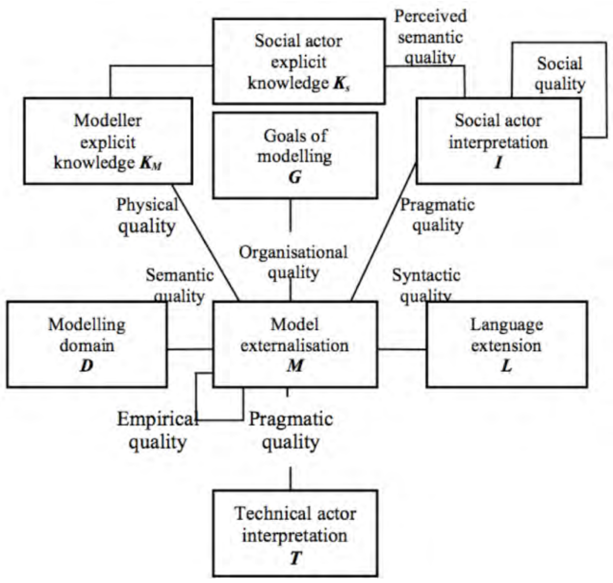


Fig. 2. The SEQUAL framework (adapted from [15], [16])

In order to support security modelling various research groups have proposed a variety of different approaches. For instance abuse frames [17] suggest means to consider security during early requirements engineering stage. Secure *i** [6] addresses security trade-offs. KAOS' extension to security [35] was augmented with anti-goal models designed to elicit attackers' rationales. Tropos has been extended with the notions of ownership, permission and trust [10]. Another version of Secure Tropos [29] defines security through the security constraints. Abuse cases [27], misuse cases [32] and mal-activity diagrams [31] are the extensions for the modelling languages from the UML family. Another UML extension (through the stereotypes, tagged values and constraints) towards security is UMLsec [13]. This language is, basically, used to address the security concerns during the system design stage. Although the majority of those approaches contribute to a proper definition of the security requirements, but they discuss little on how these security requirements should be implemented into the security controls.

Furthermore there is little support to assess these languages before their actual application to solve problems of system and software development. Thus, in this paper we propose a systematic approach, which could guide evaluation of the security languages through the hands-on testing. To illustrate application of the approach we have executed a case study where we have selected three languages – PL/SQL [9], SecureUML [2], [19], UMLsec [13]. We have investigated how these languages could contribute to the implementation of the security controls. More specifically we use these three approaches to define a role based access control (RBAC) policy for the data that needs to be secured.

3. An Assessment of Quality for Security Models

In this paper we introduce a systematic and hands-on-based approach to assess and compare quality of the security models. Our proposal consists of six steps as illustrated in Fig. 3. During the first step one needs to define the evaluation goal. With respect to the security models, the assessment goal could be understanding of the nature of the security needs, learning about the scope of the security models, learning about the quality of the security models, comparing different security models according to the quality criteria identified in the second step and similar.

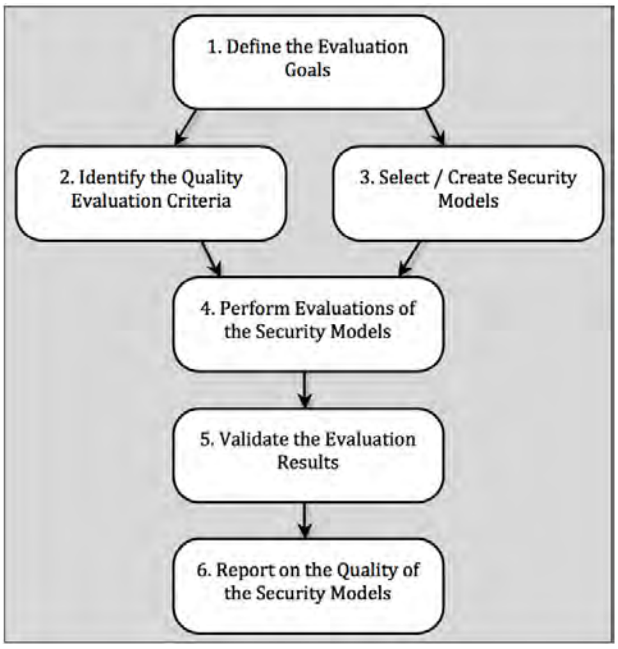

1. Define the Evaluation Goals
2. Identify the Quality Evaluation Criteria
3. Select / Create Security Models
4. Perform Evaluations of the Security Models
5. Validate the Evaluation Results
6. Report on the Quality of the Security Models

Fig. 3. An Assessment of Quality for Security Models

The second and the third steps of our proposal could be executed in parallel. The second step is identification of the quality evaluation criteria. Although, as illustrated in Section 2.2, the SEQUAL framework provides fundamental principles to evaluate model quality, firstly, it remains abstract, and, secondly, it is dedicated to the models of the general purpose, but not to the security models. As we show in Section 4.2, we select a set of qualitative properties that instantiates SEQUAL for the *security model* assessment based on the literature [4], [15] and on our experience of assessing the requirements engineering tools [21], development guidelines [11], goal modelling languages and models [24].

As discussed in Section 2.3, the security concerns could be represented using different languages. Thus, depending on the goal defined in the first step, one needs to select or to create security models, which quality will be executed assessed in the subsequent steps.

The fourth step is about performance of the evaluation of the selected/created (in step 3) security models. This includes the investigation of the models and assignment of the subjective and objective values to the predefined (in step 2) model measures.

Expressing security quality is not an easy task. Thus we introduce the fifth step where evaluators have to validate the quality evaluation results. This typically means consultation of the received measures to the experts or to the model developers (see for instance Section 4.5.2). The final step of the security model assessment is the summary and report on the evaluation results.

In Section 4 we are reporting on a case study where we use our proposal to assess quality of three security models, created using PL/SQL [9], SecureUML [2] [19] and UMLsec [13].

4. A Case Study

Two researchers have followed the steps of the assessment of the quality for security models. They have defined the evaluation goals, identified the quality evaluation criteria and created the security models for evaluation. The model assessment results were communicated to the model developers in order to validate their correctness. The overall application of the method is illustrated in the following subsections.

4.1. Defining the Evaluation Goals

The goal of this case study is twofold:

- Firstly, we are interested in learning about the quality of the security models created using different languages. More specifically we will compare the models created at the software system design stage and software system implementation stage. In both cases our model will be defining the role-based access control polity for the system data.
- Secondly, we are interested in performance and feasibility of the method introduced in Section 3. Through the case study we will record our observations on the method application.

4.2. Identifying the Quality Evaluation Criteria

Although being influenced by the overall theoretical background of the SEQUAL framework, in our study we specifically focus only on three quality types, namely *semantics*, *pragmatics*, and *syntax*. Hence we will introduce a set of measures in order to understand the quality of the security models. In fact in [25] we have already defined a set of subjective measures that helped us to address the model quality by its relative level (there we applied the ordinal scale consisting of *Low*, *Partial*, and *High* values). In this work we extend the quality model by introducing measures that allow developers to estimate quality quantitatively. The instantiation of the SEQUAL framework for the security model is illustrated in Fig. 4 and presented below.

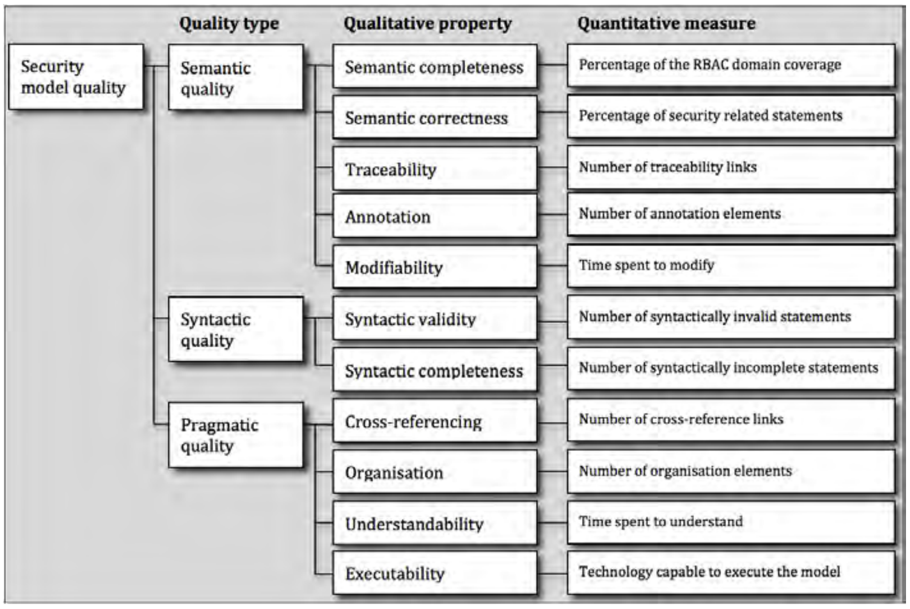


Fig. 4. Instantiation of the SEQUAL framework

Semantic quality is a correspondence between a model and its semantic domain. We assess semantic quality through the following qualitative properties and their measures:

Semantic completeness. It means that everything that the software is supposed to do is included in the model. With respect to the security domain, we say that the security model should include concepts corresponding to the RBAC domain, which is presented in Section 2.1. The *Percentage of the RBAC domain coverage* is calculated as a division between the number of RBAC concepts presented in the model and the number of RBAC concepts.

Semantic correctness. It means that a model should represent something that is required to be developed. With respect to the security domain this qualitative property requires separation between data- and security-related concerns – only the security-related knowledge is required in the security model. *Percentage of security related statements* describe the degree of security statements with respect to the overall model is.

Traceability. It requires that the origin of the model and its content should be identifiable. The security model should clearly present the rationale why different security solutions are included in the model. We define a measure *Number of traceability links*, which characterise a count of links traced to the origin of the model.

Annotation. It means that a reader is easily able to determine which elements are most likely to change. This is especially important in the security model because system security policy might be changed often. A measure of *Number of annotation elements* gives the count of annotations used in the model.

Modifiability. It means that the structure and the content are easy to change. When security policies change it should be easy to change the security concerns quickly in the model. To estimate modifiability we define a measure of *Time spent to modify*. It indicates how long it takes to change security policy in the system.

The last two qualitative properties are important when the new system security policies are introduced. Knowing the place and being able to implement the new security concerns quickly might substantially reduce the maintenance cost of overall system.

Syntactic quality is a correspondence between a model and a modelling language. The major goal of the syntactic quality is syntactic correctness. The following qualitative properties and their measures are defined:

Syntactic validity. It means that the grammatical expressions used to create a model should be a part of the modelling language. The measure defined for this qualitative property is a *Number of syntactically invalid statements*. If the value for this measure is higher the syntactical validity of the model is worse.

Syntactic completeness. It means that all grammar constructs and their parts are present in the model. We define a measure *Number of syntactically incomplete statements*. Similarly to syntactic validity measure, the syntactic completeness estimates high if *Number of syntactically incomplete statements* results in null.

To test the syntactic correctness of the security models we need to investigate the concrete syntax of the languages used to create these models.

Pragmatic quality is a correspondence between a model and an interpretation of social and technical audience. The social audience of security model is typically security engineer, but it also includes the system analysts, the software developers, the stakeholders (actors who pay for the development of the secure system), and even the direct users, who should also be involved in the security requirements definition process. With respect to the social actors we define the following qualitative properties and their measures:

Understandability. It means that a reader is able to understand the model with minimum explanations. To estimate the understandability of the security model we can count number of the explanations needed for the social audience. On the other hand here we define a measure *Time spent to understand* the model.

Cross-referencing. It means that the different pieces of model content are linked together. A measure of *Number of cross-reference links* provides a count of cross-referenced links between model components.

Organisation. It means that the model content should be arranged so that a reader could easily locate information and logical relationships among the related information. This could be done by the table of content, division of the model to different sections/chapters, inclusion of the glossary and similar. A measure of *Number of organisation elements* returns a count for the elements, which could help in arrangement of the logical information.

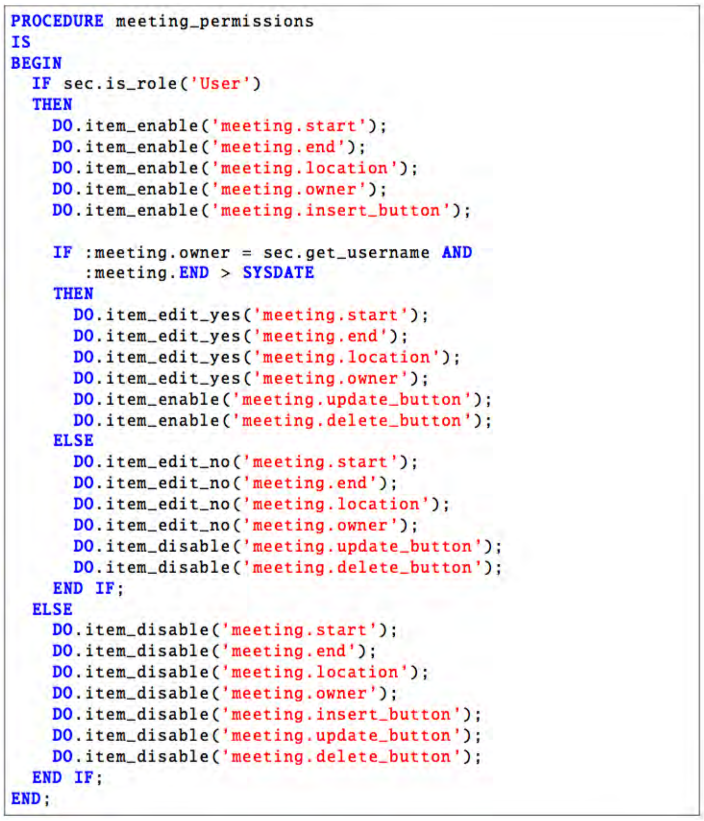
For the technical model interpretation we define that the model should be estimated according to *executability* property, meaning that there should exist technology capable of inputting the model and resulting in its implementation. The existence of technology is characterised by a measure *Technology capable to execute the model*.

4.3. Selecting / Creating Security Models

In order to understand the quality of the security models we have selected three languages: PL/SQL [9], SecureUML [2] [19], and UMLsec [13]. We have applied these languages to create the models following the RBAC policy. In fact in our models we were solving the industrial problem; however the actual data and security models could not be presented here due to the privacy concerns of our industrial partner. But here we include an extract of a *meeting scheduling system* [7]. This example closely corresponds to the industry models used in the assessment. Our observations are the same for the industrial problem and for the meeting scheduler system.

Security problem. *Meeting scheduling system* [7] is described as follows: there is a need to organise a *top-secret* meeting in the way that only intended users would know when the meeting starts and ends, what meeting owner and location are. In our example users are allowed adding information about new meetings and viewing information about all existing meetings. But one can delete or change meeting information if and only if he/she is an owner (e.g., meeting initiator) of the meeting. We will present solutions to this problem in the PL/SQL, SecureUML and UMLsec security models.

PL/SQL. Oracle PL/SQL is a procedural language extension [9] to the standard query language (SQL). PL/SQL was introduced by Oracle Corporation to overcome some limitations of SQL and to provide a more complete implementation solution to develop the mission-critical applications, which run on the Oracle database. PL/SQL is an embedded language and could not be used as a standalone language. The language ensures that the programs can stay entirely within the operating-system independent Oracle environment. One of the important aspects of the language is its tight integration with SQL. This means the programs do not rely on intermediate software (e.g. Open Database Connectivity (ODBC) or Java Database Connectivity (JDBC)) in order to run SQL statements. Among other features, PL/SQL deals with control flows, exception handling, and advanced data types.



```
PROCEDURE meeting_permissions
IS
BEGIN
  IF sec.is_role('User')
  THEN
    DO.item_enable('meeting.start');
    DO.item_enable('meeting.end');
    DO.item_enable('meeting.location');
    DO.item_enable('meeting.owner');
    DO.item_enable('meeting.insert_button');

    IF :meeting.owner = sec.get_username AND
       :meeting.END > SYSDATE
    THEN
      DO.item_edit_yes('meeting.start');
      DO.item_edit_yes('meeting.end');
      DO.item_edit_yes('meeting.location');
      DO.item_edit_yes('meeting.owner');
      DO.item_enable('meeting.update_button');
      DO.item_enable('meeting.delete_button');
    ELSE
      DO.item_edit_no('meeting.start');
      DO.item_edit_no('meeting.end');
      DO.item_edit_no('meeting.location');
      DO.item_edit_no('meeting.owner');
      DO.item_disable('meeting.update_button');
      DO.item_disable('meeting.delete_button');
    END IF;
  ELSE
    DO.item_disable('meeting.start');
    DO.item_disable('meeting.end');
    DO.item_disable('meeting.location');
    DO.item_disable('meeting.owner');
    DO.item_disable('meeting.insert_button');
    DO.item_disable('meeting.update_button');
    DO.item_disable('meeting.delete_button');
  END IF;
END;
```

Fig. 5. Excerpt of the PL/SQL security model

The PL/SQL security model is prepared using the *EditPlus*[1] tool. In general the security model consists of the library that accumulates different security procedures written in PL/SQL. In our example this library contains three procedures that define different security policies for three RBAC roles – *Admin*, *SuperUser*, and *User*. For example in Fig. 5 we illustrate a procedure of *meeting_permissions* that describes a set of permissions, which are defined on the meeting for one RBAC role, called *User* (e.g., the role is checked through the condition *if sec.is_role('User')*). Here we see that if a certain condition (e.g., a user is a meeting *owner* and the meeting end date has not yet passed) holds, it is possible to edit meeting attributes (e.g., *start*, *end*, *location*, and *owner*); otherwise editing is not allowed. In order to receive a running application one needs to compile the PL/SQL source code.

[1] http://www.editplus.com/

SecureUML. The SecureUML modelling language [2] [19] adapts the RBAC model. At the concrete syntax level SecureUML is a "lightweight extensions" of the UML, namely through stereotypes, tagged values and constraints. It introduces the concepts and the stereotypes for *User*, *Role*, and *Permission* as well as the relationships between them (*RoleAssignment* and *PermissionAssignment*). Here the secured objects and the operations are expressed through the protected objects, which are modelled using the standard UML elements.

The semantics of *Permission* is defined through *ActionType* elements used to classify permissions. Here every *ActionType* represents a class of security-relevant operations (e.g., specific security actions: *select*, *change*, *insert*, and *delete*) on a particular type of protected resource. An *AuthorisationConstraint* is a part of the access control policy. It expresses a precondition imposed to every call to an operation of a particular resource. This precondition usually depends on the dynamic state of the resource, the current call, or the environment. The authorisation constraint is attached either directly or indirectly, via permissions, to a particular model element representing a protected resource.

The SecureUML security model was prepared using *MagicDraw*[2]. The overall model consists of five diagrams. A top-level diagram is a content diagram as shown in Fig. 6. Other four diagrams present four aspects of the security model. For instance, diagram *SecurityResource-Views* describes the data, which need to be secured, diagrams *RolePermissions-Admin*, *RolePermissions-SuperUser*, and *RolePermissions-User* present the security permissions with respect to the roles Admin, SuperUser, and User.

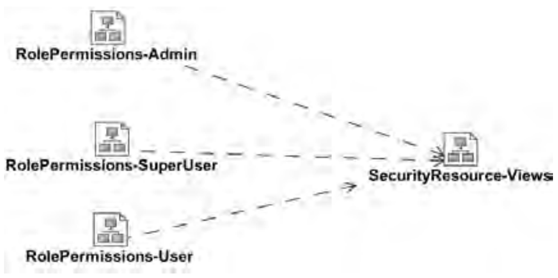


Fig. 6. SecureUML content diagram

In Fig. 7 we present an excerpt of the *Meeting Scheduling* system (*User permissions*). Here two security permissions (e.g., *UserSelectAllMeetings* and *UserUpdateOwnMeeting*) are defined for the role *User* over the resource *Meeting*. Similarly like in the PL/SQL model, an authorisation constraint *UserOwnDataConstraint* defines that only an *owner* is allowed to *update* or *delete* meeting information if the meeting date has not yet passed.

[2] http://www.magicdraw.com/

In order to receive an executable application, the SecureUML model is automatically transformed to the PL/SQL code (see illustration in the Appendix of this paper). The transformed PL/SQL code is then compiled to a running application.

In our case study we have selected to analyse the model created using SecureUML, but not its PL/SQL transformation. The reason is that we intend to analyse the model, which is editable by the system developers directly.

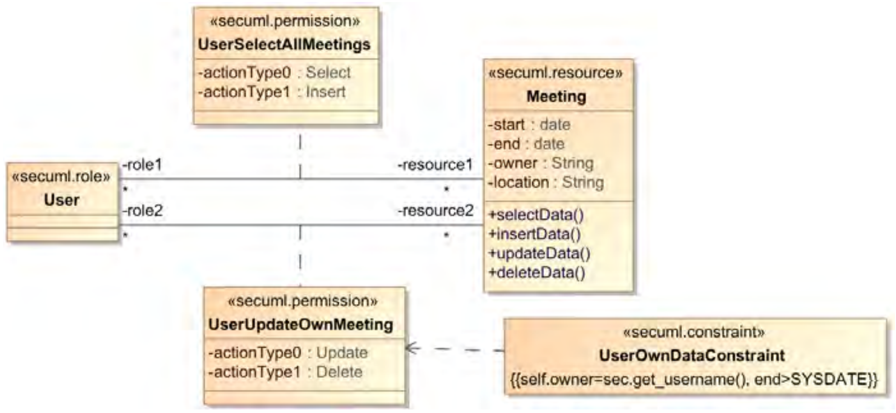


Fig. 7. Excerpt of the SecureUML security model

UMLsec. The UMLsec modelling language [13] is defined as a UML profile extension using stereotypes, tagged values and constraints. Constraints specify security requirements. Threat specifications correspond to actions taken by the adversary. Thus, different threat scenarios can be specified based on adversary strengths.

A subset of UMLsec that is directly relevant to this study is the role-based access control stereotype – <<*rbac*>> – its tagged values and constraints. This stereotype enforces RBAC in the business process specified in the activity diagram. It has three associated tags *{protected}*, *{role}*, and *{right}*. The tag *{protected}* describes the states in the activity diagram where the access to the activities should be protected. The *{role}* tag may have a list of pairs (*actor*, *role*) as its value, where *actor* is an actor in the activity diagram, and *role* is a role. The tag *{right}* has a list of pairs (*role*, *right*) as its value, where *role* is a role and *right* represents the right to access a protected resource. The associated constraint requires that the actors in the activity diagram only perform actions for which they have the appropriate rights.

In Fig. 8 we define an activity diagram, which describes an interaction between *User* and *Meeting*. The diagram specifies that *User* can *Insert data* (e.g., meeting start- and end-dates, meeting owner, and meeting location). Next, *User* is able to *Select data* in order to check if data are correct. If these are not OK *User* is able to *Update* data. After the meeting is over, *User* is able to *Delete data* about this meeting.

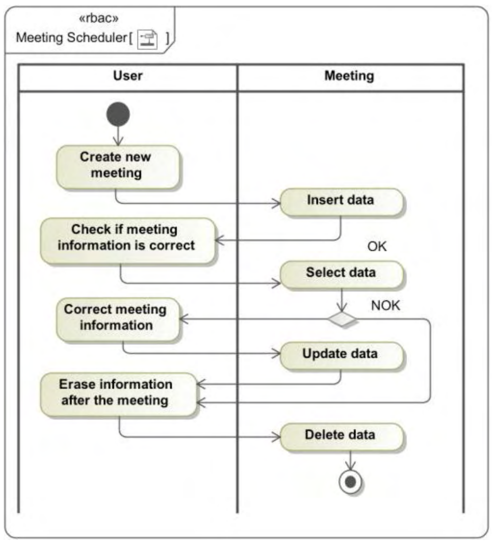

«rbac»
Meeting Scheduler
User
Meeting
Create new meeting
Insert data
Check if meeting information is correct
OK
Select data
NOK
Correct meeting information
Update data
Erase information after the meeting
Delete data

Fig. 8. Meeting Scheduler with UMLsec

This diagram carries an <<*rbac*>> stereotype, meaning that the security policy needs to be applied to the protected actions. For instance, the *User*'s actions lead to the secured actions executed by the *Meeting*. For example*,* *Insert data* is executed if and only if there exists an associated tag that defines the following: (*i*) *Insert data* is a protected action, (*ii*) there exists a user (e.g., *Bob*) who plays role *User*, and (*iii*) *User* enforces the action *Insert data*. In the activity diagram this associated tag is defined as follows:

```
{protected = Insert data}
{role = (Bob, User)}
{right = (User, Insert data)}
```

Similarly, the sets of associated tags are defined for other three protected actions *Select data*, *Update data*, and *Delete data*. Like in the SecureUML model, using UMLsec we need to define activity diagrams (with the models <<*rbac*>> stereotype) for other two actors – *Admin* and *SuperUser*.

4.4. Performing Evaluation of the Security Models

In this section we will subsequently discuss the results of our assessment of the security models. We will see the results on *semantic*, *syntactic* and *pragmatic* quality types.

4.4.1. Assessment of Semantic Quality

Our analysis of the *semantic quality* for the security models is summarised in Table 1. As defined in Section 4.2 we considered semantic quality according to *semantic completeness*, *semantic correctness*, *traceability*, *annotation*, and *modifiability*.

Table 1. Semantic quality of the security models

Qualitative property	Measure	PL/SQL security model	SecureUML security model	UMLsec security model
Semantic completeness	Percentage of the RBAC domain coverage	42,86%	71,43% (100%)	85,71%
Semantic correctness	Percentage of security related statements	7,69%	100%	33%
Traceability	Number of traced links	0	0	0
Annotation	Number of annotation elements	0	5	1
Modifiability	Time spent to modify	Not-known	5-10 minutes	5-10 minutes

PL/SQL security model. *Semantic completeness* is assessed through a model correspondence to the RBAC domain (see Section 2.1). In the first condition the PL/SQL model explicitly defines the role (e.g., *User* in Fig. 5) for which security permission is defined. Next the PL/SQL model focuses partially on the presentation of the security *permissions* (e.g., see the second condition expression in Fig. 5), which are defined for the attributes of secured *objects* (e.g., statements like *meeting.start*, *meeting.end*, and others shown in Fig. 5). However it does not define on which *operations* the security permissions are placed. Also the PL/SQL model does not express *users* and *user assignment* relationships. We estimate 42.86% (expresses 3 RBAC concepts out of 7) of the RBAC domain coverage.

The *semantic correctness* of the PL/SQL model is low, because it does not separate the data and programmable concerns from the security concerns. In PL/SQL diagram we found only two statements that are defining security concerns (see two conditions defined in Fig. 5). All other 24 statements are defining different programmable variables or user interface components (e.g., *DO.item_enable('meeting.new_meeting')* is enabling the item of the user interface). We estimate only 7,69% (2 statements out of 26) of the security related statement in the diagram presented in Fig. 5.

The PL/SQL model is not *traced*. This means that origin and rationale for the security decisions are not provided in the model and we did not observe any traceable links in this model. The PL/SQL model is not *annotated*, thus it is difficult to determine which elements are most likely to change.

Modifiability is estimated by the time used to modify different aspects of the model. To estimate this characteristic it was rather difficult because it directly correlates to the *understandability* property (see discussion below). However we acknowledge that, once the model is understood, time spent to modify the model might depend on the scope of the changes and skills of the developer.

SecureUML security model. SecureUML is developed to design the RBAC-based solutions. This means that SecureUML could fully correspond to the semantic domain, thus resulting in high *semantic completeness*. However in our analysed diagram (see Fig. 7) we did not identify RBAC concept of *User* and relationship *User assignment*. Thus we result in 71,43% of the RBAC domain coverage (however we should note that definition of *User* and *User assignment* is not a problem using SecureUML, thus possibly resulting in 100% of semantic completeness).

We identify high *semantic correctness*, because only security solutions are presented in the SecureUML model. We assess percentage of security related statements as 100%.

Like in the PL/SQL security model, in the SecureUML model we did not observe any rationale for security decisions, thus it results in a low *traced* property.

The Secure UML model is partially *annotated*. This annotation is achieved through SecureUML stereotypes (e.g., <<secuml.permission>>, <<secuml.role>>, etc.) and class names given to the *permissions* (e.g., *UserSelectAllMeetings* and *UserUpdateOwnMeeting*) and the *authorisation constraints* (e.g., *UserOwnDataConstraint*). These class names are not directly used in the transformation of the model to code, but they provide additional information to the model reader. They also identify the places in the model where security policy is most likely to be changed. We counted 5 annotation examples in the SecureUML model.

The SecureUML model is *modifiable.* The model implies a certain presentation pattern – *Role-Permission-Resource*, which facilitates the changing of the model. Like for the PL/SQL model we acknowledge that modifiability much depends on the change requirements and on the skills of the developer, but we also observe that the average time of one change might vary from 5 to 10 minutes.

UMLsec security model. The RBAC principles are expressed through the activity diagram using UMLsec. Using UMLsec the majority of the RBAC concepts are defined in the associated tags. For example, *User* and *Roles* are associated in the {*role*} tag, thus, expressing the RBAC *user association* link), *Roles* and *Operations* are combined in the {*right*} tag, thus, defining the RBAC *Permission association* link. The only RBAC *concept* that is not expressed in the UMLsec model is *Permission*, i.e., what the *Roles* are allowed to do with the secure *Objects*. We result in 85,71% (6 concepts out of 7) of the RBAC domain coverage.

Regarding semantic correctness, in the UMLsec diagram we can observe actions related to business/work description (e.g., *Create new meeting*, *Check if meeting information is correct*, *Correct meeting information*, and *Erase information after the meeting*) and actions that needs to support the business/work actions (e.g., ones executed by *Meeting – Insert data*, *Select data*, *Update data*, and *Delete data*). The later ones each needs security-related treatment defined through the association tags. Thus we result in 33% of security related statements (actions and association tags) in the UMLsec model.

In the UMLsec model we find only one annotation element, i.e., the <<*rbac*>> (see Fig. 8) stereotype that the modelled security aspect. Similar like in the SecureUML model, we observed no traceability from/to the UMLsec model. In addition, we identify, that depending on the needs for changes, we can modify the UMLsec model in 5-10 minutes.

4.4.2. Assessment of Syntactic Quality

Syntactic quality is expressed through *syntactic validity* and *syntactic completeness*, as defined in Section 4.2. We summarise our analysis of the security models in Table 2.

Table 2. Syntactic quality of the security models

Qualitative property	Measure	PL/SQL security model	SecureUML security model	UMLsec security model
Syntactic validity	Number of syntactically invalid statements	0	1	0
Syntactic completeness	Number of syntactically incomplete statements	0	0	0

PL/SQL security model. The PL/SQL model is of high *syntactic validity* and *syntactic completeness*, because the model is created using the PL/SQL language, a programmable language. We did not observe any syntactically invalid or syntactically incomplete statements. Syntactically this model is also correct because otherwise it would not be possible to compile it to the application.

SecureUML security model. In the current model of the SecureUML we can identify a case of *syntactic invalidity*. For instance the SecureUML documentation [2] [19] identify that *authorisation constraints* need to be written in OCL (Object Constraint Language). However in our model (see Fig. 7) the SQL-based authorisation constraints are used (e.g., see class *UserOwnDataConstraint* constraint *{owner=sec.get_username*(), *end>SYSDATE}*). On the other hand the model is *syntactically complete* – it includes only UML extensions and their relationships proposed by the authors of SecureUML, thus we did not observe any syntactically incomplete statements.

UMLsec security model. We did not observe any syntactically invalid or syntactically incomplete statements in the UMLsec model. However we should note that this model was checked only manually. For the UMLsec model investigated by us, we were not running any transformations to the application code (like we did with the PL/SQL or SecureUML models).

4.4.3. Assessment of Pragmatic Quality

We summarise the analysis of the pragmatic quality for the security models in Table 3. Pragmatic quality is defined in terms of *understandability*, *organisation*, *cross-referencing*, and *executability*, as presented in Section 4.2.

Table 3. Pragmatic quality of the security models

Qualitative property	Measure	PL/SQL security model	SecureUML security model	UMLsec security model
Understand a-bility	Number of explanations	More than 45 minutes	10-15 minutes	10-15 minutes
Organisation	Number of elements for model organisation	2	4	4
Cross referencing	Number of cross-reference links	1	3	3
Executability	Tools to execute the model	Yes	Yes	No

PL/SQL security model. We found the PL/SQL model of low *understandability*. We were not able to understand the PL/SQL model without a proper explanation provided by the model developers. All together it took us more than 45 minutes to grab some security concerns defined in the PL/SQL model. On the one hand the reason might be that we as the evaluators, were not the experts in the PL/SQL language. But, on the other hand, taking into account that the security models should be used to communicate with the users of the software systems (who are not familiar with PL/SQL neither), the time spent to understand security concerns could be even longer.

As presented in Section 4.3, the PL/SQL model is *organised* into the library that accumulates different security-oriented procedures. Thus, this model contains a structure, which could guide finding the relevant security concerns.

Furthermore the PL/SQL model is presented as a plain-text source code, thus it does not contain any hyperlinks that would *cross-reference* related security concerns (but also see Section 4.5.2). On the other hand the library structure could be used to follow from one security procedure to another (in our case between three procedures, defined regarding to the user *role*). However these links could be used only manually; no tool support for them is provided.

Finally, regarding the PL/SQL model *executability*, it is possible to compile this model using the Oracle database management system resulting in a running application.

SecureUML security model. The Secure UML model is well *understood* by those readers familiar with the UML modelling notation. This also opens the way to communicate this model to a larger audience, including various project stakeholders, potential direct users of the system, the systems analysts, and the developers. Our personal experience is that this model is quite intuitive and did not require a big effort (around 10-15 minutes) to understand it.

As described in Section 4.3, the SecureUML model consists of several diagrams. It is also supported by a modelling tool (in our case – MagicDraw), which simplifies managing the model itself and support the model organisation. The tool provides the containment view and zoom means (see Fig. 9), which developer could use to find the relevant model elements, navigate between and within the model diagrams. As illustrated in Fig. 6 the navigation map diagram helps to navigate from the content diagram to diagrams presenting different security concerns.

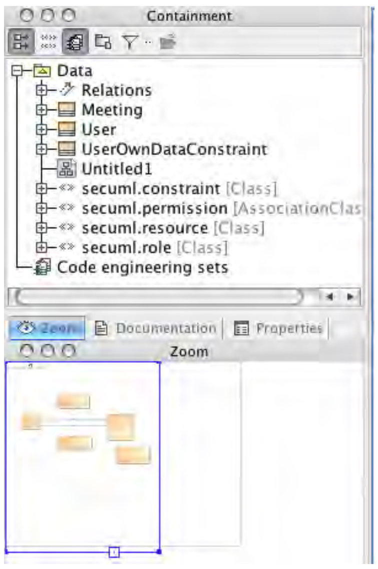


Fig. 9. Means to support SecureUML model organisation provided by the tool

Model *cross-references* includes links between the navigation map and separate diagrams, between the containment views and separate diagrams and model elements. It is also possible to define cross-references between the separate model diagrams (however this possibility was not used in our case).

The SecureUML model is *executable*: there exists a number of the transformation rules defined using the Velocity[3] language (interpretable by the MagicDraw tool).These rules define how to transform the model to PL/SQL code, which could be executed through Oracle database management system.

UMLsec security model. Regarding the social actor interpretation, we result in the same assessment of the UMLsec model as for the SecureUML model. For instance, we found that both models can be understood in 10-15 minutes. The UMLsec contains 4 elements for its organisations (since it is

[3] http://velocity.apache.org/engine/devel/user-guide.html

created using MagicDraw, the same modelling tool as the SecureUML security model). Similarly it includes three means to cross reference inter-related parts.

However we were not able to execute the UMLsec model – there are no means to generate the PL/SQL code from this model (at least using the MagicDraw tool). Thus there exist a potential field for improvement regarding the technical interpretation aspect.

4.5. Validating the Evaluation Results

After performing the evaluation of the security models, next step is to validate the received results. In this section we will characterise the potential threats to validity. We will also describe what feedback we received from the models authors regarding our evaluation scores.

4.5.1. Threats to Validity

In our case study *only* two evaluators assessed the security models according to their knowledge and experience. This certainly raises the level of subjectivity and influences the *internal validity* of the case study. To mitigate this threat the evaluation results were communicated to the model developers.

In our case the SEQUAL framework was instantiated with a *certain* set of qualitative properties (and their measures). This certainly affects the *conclusion validity*, because if any other qualitative properties were applied, it might result in different outcome. But this threat is rather limited because these qualitative properties are theoretically sound and the selection is based on the previous experience (i.e., [4], [11], [15], [21], [24]).

In this case study we analysed *only* three different security models and these models were quite limited in their size. This might influence the *external validity* by a fact, that different results might be received if some other security models (created either using PL/SQL, SecureUML, UMLsec or any other language) would be analysed. However our research subject is providing a solution to an industry problem; thus, we believe that our analysis is generalisable in similar situations.

Finally, we try to avoid a use of single type of measuring that might affect the *construct validity*. The evaluation of the security models is followed with the communication of the received results to the models developers (see Section 4.5.2). This certainly reduces a risk of the mono-interpretation.

4.5.2. Communicating Results to Developers

We reviewed our results together with the developers of the security models. Firstly, the developers noted that the overall quality of both models could be improved if these evaluation results were taken into account. For example, the *traceability*, *annotation*, and *understandability* of the PL/SQL model could be easily improved using code comments. However, the developers acknowledged that this is not the case in the common practice; or the code comments, even if they are present, are not sufficient.

Secondly, developers provided few remarks regarding some qualitative properties. For instance, *semantic completeness* could be improved by presenting concrete instances in the models (similarly as done in [2] and [19]). This means hard coding in the PL/SQL model and object presentation in the SecureUML model; however, doing so we would neglect the principle of generosity in modelling.

In order to improve *syntactic validity* of the SecureUML model we could write the authorisation constraints in OCL instead of SQL. However the current approach to transform the SecureUML model does not have rules for the OCL interpretation. Further, it is not possible to perform transformation from the UMLsec security model to the executable code. Certainly the targeted transformation templates (as they are provided for the models created in SecureUML) could improve the *executability* of UMLsec.

On the one hand, a tool used to make the PL/SQL model, does not support hyper-linking. Although there exist several PL/SQL editing tools (e.g., *Oracle SQLDeveloper* or *Quest Software Toad for Oracle*, actually used by our industrial partner) that supports cross-references between various model elements, these were not used in this case study. On the other hand, developers also indicated that PL/SQL grammar principles, the ones, which allow expressing procedures (e.g., *PROCEDURE meeting_permissions* in Fig. 5) and referring to them from the main code, could also be seen as textual *cross-referencing*. We took this in mind when scoring for the *Number of cross-reference links*.

4.6. Reporting on the Quality of the Security Models

Table 4 shows the summary of the overall comparison of the security models. We found that three qualitative properties (i.e., *traceability*, *syntactic completeness*, and *executability*) score equally for the PL/SQL and SecureUML models. One qualitative property – *syntactic validity* – is found to be better in the PL/SQL model. The seven remaining qualitative properties (i.e., *semantic completeness*, *semantic correctness*, *annotation*, *modifiability*, *understandability*, *organisation*, and *cross-referencing*) are evaluated to be higher in the SecureUML model.

Regarding models in PL/SQL and UMLsec, we see that PL/SQL was scoring better for *executability* qualitative property. Three qualitative properties – *traceability*, *syntactic validity* and *syntactic completeness* – are assessed equally. The remaining seven qualitative properties (*semantic completeness, semantic correctness, annotation, modifiability, understandability, organisation*, and *cross-referencing*) are evaluated better for the security model created in UMLsec.

Table 4. Summary of quality assessment for the security models

Model A created in	Model B created in	Model A is better in	Two models score equal in	Model B is better in
PL/SQL	**SecureUML**	Syntactic validity	Traceability, syntactic completeness, executability	Semantic completeness, semantic correctness, annotation, modifiability, understandability, organisation, and cross-referencing
		1 qual. property	*3 qual. properties*	*7 qual. properties*
PL/SQL	**UMLsec**	Executability	Traceability, syntactic validity, syntactic completeness	Semantic completeness, semantic correctness, annotation, modifiability, understandability, organisation, cross-referencing
		1 qual. property	*3 qual. properties*	*7 qual. properties*
SecureUML	**UMLsec**	Semantic completeness, semantic correctness, annotation, executability	Traceability, modifiability, syntactic completeness, understandability, organisation, cross-referencing	Syntactic validity
		4 qual. properties	*6 qual. properties*	*1 qual. property*

Six qualitative properties, namely *traceability, modifiability, completeness, understandability, organisation*, and *cross referencing* – are evaluated equally both for the SecureUML and for the UMLsec security models. One qualitative property – syntactic validity – is found better for the UMLsec model. The remaining four qualitative properties (*semantic completeness, semantic correctness, annotation*, and *executability*) are evaluated better for the SecureUML security model.

5. Discussion

In this section we finalise our work. Firstly, we discuss the related work regarding the link between the RBAC, security languages and the model-driven security. Next, we conclude our paper and highlight few future research directions.

5.1. RBAC and Security Languages

In [1] the BRAC$_0$ pattern is applied for comparison of security modelling approaches. The survey shows that, on the one hand, SecureUML does not explicitly model security criteria (such as confidentiality, integrity, and availability) but it focuses on modelling the solutions to security problems guided by the RBAC nature. With SecureUML, a modeller can define assets, however, the language does not allow expressing attacks or harms to the assets. On the other hand, UMLsec is guided by security criteria, however it does not have means to model them explicitly. The UMLsec application is driven by analysis of system vulnerabilities: (i) once security vulnerabilities have been identified, the system design is progressively refined to eliminate the potential threats; (ii) the refinement of the design might be continued until the system satisfies the security criteria. Although UMLsec was analysed based on the $BRAC_0$ pattern, authors does not specifically indicate how well this approach is suitable for the RBAC modelling.

In [12] Jayaram and Mathur investigate how the practice of software engineering blends with the requirements of secure software. The work describes a two-dimensional relationship between the software lifecycle stages and modelling approaches used to engineer security requirements. A part of the study is dedicated to the RBAC modelling using SecureUML and UMLsec. Authors indicate that UMLsec is rather general approach than specific, thus it cannot be used to model access control policies solely. On the other hand SecureUML is suggested as the means to specify access control policies. However SecureUML cannot describe protected resources (system design), thus, it has to be used in conjunction with a base modelling language.

Elsewhere in [22] [23] the SecureUML and UMLsec are compared in order to determine the transformation points between models of these languages. It was noticed the limitation of SecureUML to indicate security criteria, but this language is well suited to engineer security controls after the security decisions are done. It was also observed that the UMLsec application follows the standard security modelling methods [26] and it could provide means for the RBAC modelling: it helps defining the dynamic characteristics of the secure system. The analysis suggests that both SecureUML and UMLsec can complement each other and result in more complete specifications of secure information systems (where both static and dynamic system characteristics are understood).

Although the identified works are useful regarding their timely comparison of the modelling languages against the RBAC model, these studies remain theoretical. It is suggested that such an approach could be used at the initial stage of the languages selection, but for the deeper understanding one needs more fine-grained analysis of the development means. Thus our current proposal – an approach to assess the quality of the security models – suggests the means for the hands-on testing of the modelling and development languages for security. Using our proposal the developers are encouraged to apply the modelling and development languages in order to understand the quality of the resulting security models.

5.2. Model-driven Security

We found none empirical studies that would compare quality of security models prepared using approaches from *different development stages*. The literature reports on a number of case studies [5], [33], [34] analysing different characteristics of the *model-driven development*. Mostly these studies focus on the benefits and on the infrastructure needed for the model-driven development. Similarly to [3], [20], [34] we observe that security model facilitates automatic code generation, i.e., the SecureUML security model is *executable* through its generation to PL/SQL code. We also argue that the security models should be prepared with the high-quality modelling language [5] that ensures the model *semantic completeness*, and tools [20] that guarantee model *syntactic validity* and *syntactic completeness*. Only then one could expect that model-driven security could yield a higher productivity with respect to a traditional development [34].

We identified only one case study performed by Clavel *et al* [3], reporting on the SecureUML application in practice. Here authors observe that although the security models are integrated with the data models, the security design remains independent, reusable and evolvable. In our work we also observe that *semantic correctness* of SecureUML and UMLsec models is high, because the representation is oriented to the security aspects. We also observe that SecureUML and UMLsec models are *modifiable*, which means the first step towards model evolvability. Like in [3] we identify that the SecureUML and UMLsec models are understandable at least to readers who are familiar with UML. This might ease communication of requirements and design solutions to project stakeholders [20].

5.3. Conclusion and Future Work

In this paper we have developed a systematic approach to compare quality of security models. Our approach is based on the instantiation of the SEQUAL framework [15] [16]. To illustrate the performance of our proposal we have executed a cases study, where we have compared quality of three security models. One model is prepared at the *implementation stage* using PL/SQL [9]; other two models are developed at the *system design stage* using SecureUML [2] [19] and UMLsec [13]. We resulted in (*i*) a higher quality for the SecureUML security model regarding UMLsec and PL/SQL; and (*ii*) higher quality for the UMLsec security model regarding PL/SQL. Thus, it suggests that practitioners should consider security analysis at the earlier stages (at least design or maybe even requirements engineering) of the software system developing. However we also note that *executability* of the UMLsec model is worse than *executability* of the PL/SQL model. Thus, if one wishes to create executable models he would prefer PL/SQL (or SecureUML) instead of UMLsec.

Our comparison also identifies important directions [33] for improvement of the security analysis at the early stages. For example, a mature *security modelling method* needs to be introduced in order to guide discovery of the early security requirements and to support security quality assurance through overall project planning. This would allow improving the *traceability* qualitative property, also facilitating recording of the rationales for security decisions.

Another concern includes development and improvement of the modelling *tools* (e.g., MagicDraw and Velocity interpreter) that would support the translation of the design models (e.g., SecureUML) to the implementation code (e.g., PL/SQL). For instance, we need to define guidelines and transformation rules for the OCL-based authorisation constraints. This would also improve the *syntactic validity* of the SecureUML model. On the other hand *executability* of the UMLsec security model is not supported at all – this might result in that practitioners would select the PL/SQL language instead.

For the successful adoption by practitioners, model driven security analysis should be compatible with the working *processes*. We plan to perform another case study where we would investigate quality of processes to develop security models at the system design stage (e.g., using SecureUML, UMLsec or other modelling language) against quality of processes to develop security models at the system implementation stages (e.g., using PL/SQL).

Finally, we need to support a *goal-driven process* [33], where we would define goals to introduce security model-driven development systematically. In this paper we specifically focused on the security policy for the data model. Our future goal is to develop transformation rules that would facilitate implementation of the security concerns at the system application and presentation levels.

Acknowledgment. This research was conducted while the first and third authors were at the Software Technology and Applications Competence Centre (STACC) and the second author was at Logica Estonia. The research is partly funded by the EU Regional Development Funds via Enterprise Estonia. We also thank the anonymous referee for the helpful comments and suggestions.

References

1. Bandara, A., Shinpei, H., Jurjens, J., Kaiya, H., Kubo, A., Laney, R., Mouratidis, H., Nhlabatsi, A., Nuseibeh, B., Tahara, Y., Tun, T., Washizaki, H., Yoshioka, N., Yu, Y.: Security Patterns: Comparing Modelling Approaches. Technical Report No 1009/06, Department of Computing Faculty of mathematics, Computing Technology, The Open University (2009)
2. Basin, D., Doser, J., Lodderstedt, T.: Model Driven Security: from UML Models to Access Control Infrastructure. ACM Transactions on Software Engineering and Methodology (TOSEM), 15 (1), 39--91. (2006)
3. Clavel, M., Silva, V., Braga, C., Egea, M.: Model-driven Security in Practice: an Industrial Experience, In Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications, Springer-Verlag, pp. 326--337. (2008)
4. Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Srimani, P., Ta, A., Theofanos, M.: Identifying and Measuring Quality in a Software Requirements Specification. In Proceedings of the 1st International Software Metrics Symposium, pp. 141--152. (1993)
5. de Miguel, M., Jourdan, J., Salicki, S.: Practical Experiences in the Application of MDA. In Proceedings of the 5th International Conference on The Unified Modeling Language, Springer-Verlag, 128--139, (2002)
6. Elahi, G., Yu, E.: A Goal Oriented Approach for Modeling and Analyzing Security Trade-Offs, In: Parent et al. (eds.), Proceedings of the 26th International Conference on Conceptual Modelling (2007)

7. Feather, M.S., Fickas, S., Finkelstein, A., van Lamsweerde A.: Requirements and Specification Exemplars. Automated Software Engineering, 4: 419--438. (1997)
8. Ferraiolo D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST Standard for Role-based Access Bontrol. ACM Transactions on Information and System Security (TISSEC), 4(3), 224--274. (2001)
9. Feuerstein, S., Pribly, B.: Oracle PL/SQL Programming. O'Reilly Media Inc, 4th edition edition (2005)
10. Giorgini, P., Massacci, F., Mylopoulos, J., Zannone, N.: Modeling Security Requirements Through Ownership, Permision and Delegation. In Proceedings of the 13th IEEE International Conference on Requirements Engineering, IEEE Computer Society (2005)
11. Hakkarainen S., Matulevičius R., Strašunskas D., Su X. and Sindre G.: A Step Towards Context Insensitive Quality Control for Ontology Building Methodologies. In Proceedings of the CAiSE 2004 Open INTEROP-EMOI Workshop, 205--216. (2004)
12. Jayaram, K.R., Mathur, A.P.: Software Engineering for Secure Software – State of the Art: a Survey. Technical report CERIAS TR 2005-67, Department of Computer Sciences & CERIAS, Purdue University (2005)
13. Jurjens, J.: Secure Systems Development with UML. Springer-Verlag Berlin Heidelberg, (2005)
14. Knodel, J., Anastasopolous, M., Forster, T., Muthig, D.: An Efficient Migration to Model-driven Development (MDD). Electronic Notes in Theoretical Com puter Science 137 17--27. (2005)
15. Krogstie, J.: A Semiotic Approach to Quality in Requirements Specifications. In Proceedings of IFIP 8.1 working Conf. on Organisational Semiotics, 231--249. (2001)
16. Krogstie, J.: Using a Semiotic Framework to Evaluate UML for the Development for Models of High Quality. In: Siau, K., Halpin, T. (eds.) Unified Modelling Language: Sys- tem Analysis, Design and Development Issues, IDEA Group Publishing, pp. 89--106. (1998)
17. Lin, L., Nuseibeh, B., Ince, D., Jackson, M.: Using Abuse Frames to Bound the Scope of Security Problems. In Proceedings of the 12th IEEE International Conference on Requirements Engineering, IEEE Computer Society 354--355. (2004)
18. Lindland, O. I., Sindre, G., Sølvberg, A.: Understanding Quality in Conceptual Modelling. IEEE Software, 11(2), pp. 42--49. (1994)
19. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-based Modeling Language for Model-driven Security. In Proceedings of the 5th International Conference on The Unified Modeling Language, LNCS, vol. 2460 Springer-Verlag, 426--441. (2002)
20. MacDonald, A., Russell, D., Atchison, B.: Model-driven Development within a Legacy System: An Industry Experience Report. In Proceedings of the 2005 Australian Software Engineering Conference (ASWEC'05). IEEE Computer Science. (2005)
21. Matulevičius, R.: Process Support for Requirements Engineering: A Requirements Engineering Tool Evaluation Approach. PhD theses. Norwegian University of Science and Technology. (2005)
22. Matulevičius, R., Dumas, M.: A Comparison of SecureUML and UMLsec for Role-based Access Control, Proceedings of the 9th Conference on Databases and Information Systems, 171--185. (2010)

23. Matulevičius, R., Dumas, M.: "Towards Model Transformation between SecureUML and UMLsec for Role-based Access Control," Databases and Information Systems VI, IOS Press, 339--352. (2011)
24. Matulevičius, R., Heymans, P.: Comparison of Goal Languages: an Experiment. In Proceedings of the Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2007), Trondheim, Norway, Springer-Verlag, 18--32. (2007)
25. Matulevičius, R., Lepmets, M., Lakk, H., Sisask, A.: Comparing Quality of Security Models: a Case Study. In Local Proceedings of the 14th East-European Conference on Advances in Database and Information Systems. University of Novi sad, Serbia, 95 - 109. (2010)
26. Mayer N.: Model-based Management of Information System Security Risk. PhD Thesis, University of Namur (2009)
27. McDermott, J., Fox, C.: Using Abuse Case Models for Security Requirements Analysis. In Proceedings of the 15th Annual Computer Security Applications Conference (1999)
28. Moody, D.L.: Theoretical and Practical Issues in Evaluating the Quality of Conceptual Models: Current State and Future Directions. Data and Knowledge Engineering 55 (3) 243--276. (2005)
29. Mouratidis, H.: Analysing Security Requirements of Information Systems using Tropos. In Proceedings 1st Annual Conference on Advances in Computing and Technology 55--64. (2006)
30. Piattini, M., Genero, M., Poels, G., Nelson, J.: Towards a Framework for Conceptual Modelling Quality. In: Genero, M., Piattini, M., Calero, C. (eds.) Metrics for Software Conceptual Models, Imperial College Press, London 1--18. (2005)
31. Sindre, G.: Mal-activity Diagrams for Capturing Attacks on Business Processes. In Proceedings of the Working Conference on Requirements Engineering: Foundation for Software Quality, Springer-Verlag Berlin Heidelberg 355--366. (2007)
32. Sindre, G., Opdahl, A.L.: Eliciting Security Requirements with Misuse Cases. Requirements Engineering Journal 10 (1) 34--44. (2005)
33. Staron, M.: Adopting Model Driven Software Development in Industry – A Case Study at Two Companies. In the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006). Springer-Verlag 57--72. (2006)
34. The Middleware Company: Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach: Productivity Analysis, MDA Productivity case study. (2003)
35. van Lamsweerde, A.: Elaborating Security Requirements by Construction of Intentional Anti-models. In Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society 148--157. (2004)

Appendix

In order to get the impression on how the SecureUML security model (e.g., see Fig. 7) is transformed into the PL/SQL code, we included a sample of the transformation outcome with respect to the *Update* security action. Similarly the PL/SQL code is generated for other three security actions – *Select*, *Insert* and *Delete*.

```
-- Imported common-sql.vtl
CREATE OR REPLACE TRIGGER Meeting_sec_update_trg
  INSTEAD OF UPDATE ON Meeting_v
  REFERENCING NEW AS NEW OLD AS OLD
  FOR EACH ROW
DECLARE
  self      Meeting%ROWTYPE;
  ex_denied EXCEPTION;
BEGIN
  SELECT *
    INTO self
    FROM Meeting res
   WHERE res.ID = :OLD.ID;
  IF util.null_eq(:NEW.start, :OLD.start) != 'Y' -- start updated
   THEN
    IF 1 != 1 OR sec.is_role('User') = 'Y' AND
       self.owner = sec.get_username() AND
       self.end > SYSDATE -- Permission from UserUpdateOwnMeeting
     THEN
      self.start := :NEW.start;
    ELSE
      RAISE ex_denied;
    END IF;
  END IF;
  IF util.null_eq(:NEW.end, :OLD.end) != 'Y' -- end updated
   THEN
    IF 1 != 1 OR sec.is_role('User') = 'Y' AND
       self.owner = sec.get_username() AND
       self.end > SYSDATE -- Permission from UserUpdateOwnMeeting
     THEN
      self.end := :NEW.end;
    ELSE
      RAISE ex_denied;
    END IF;
  END IF;
  IF util.null_eq(:NEW.owner, :OLD.owner) != 'Y' -- owner updated
   THEN
    IF 1 != 1 OR
       sec.is_role('User') = 'Y' AND
       self.owner = sec.get_username() AND
       self.end > SYSDATE -- Permission from UserUpdateOwnMeeting
     THEN
      self.owner := :NEW.owner;
    ELSE
      RAISE ex_denied;
    END IF;
  END IF;
  IF util.null_eq(:NEW.location, :OLD.location) != 'Y' -- location updated
   THEN
    IF 1 != 1 OR
       sec.is_role('User') = 'Y' AND
       self.owner = sec.get_username() AND
       self.end > SYSDATE -- Permission from UserUpdateOwnMeeting
     THEN
      self.location := :NEW.location;
    ELSE
      RAISE ex_denied;
    END IF;
  END IF;

  UPDATE Meeting res
     SET ROW = self
   WHERE res.ID = :OLD.ID;
EXCEPTION
  WHEN ex_denied THEN
    raise_application_error(-20000, 'Access denied!');
END;
/
```

Dr. Raimundas Matulevičius received his PhD diploma from the Norwegian University of Science and Technology, Norway in the area of computer and information science. Currently Matulevičius holds an associated professor position at the Institute of Computer Science, University of Tartu, in Estonia. Matulevičius' research interests cover information systems and requirements engineering, system and software development processes, model-driven development, system and software security, and security risk management. Currently, the publication record includes more than 50 articles published in the peer-reviewed international journals, conferences and workshops. Matulevičius was invited for multiple times to co-review papers for the international journals (e.g., REJ, TOSEM, SoSyM, COSE). Few years in a row he is invited to be a program committee member at the international workshops and conferences (e.g., CAiSE, REFSQ, PoEM and other).

Henri Lakk is a master's degree student at University of Tartu, where he is also giving labs and lectures. His study and research interest includes model driven security of information system. Lakk is working also in Webmedia Estonia as a PL/SQL programmer.

Dr. Marion Lepmets is a recognised researcher on software and IT service quality, process improvement and assessment. She is currently a Post-Doctoral fellow in Public Research Centre Henri Tudor conducting research on IT service quality measurement and process improvement impact on IT service quality. She is a technical program committee member at SPICE, EuroSPI and Baltic IT&DB conferences, and Luxembourgish representative to ISO/IEC JTC1 SC7 (software and systems standards subcommittee).

Received: December 31, 2010; Accepted: April 29, 2011.

DOI: 10.2298/CSIS110107006A

GammaPolarSlicer

Sérgio Areias, Daniela da Cruz, Pedro Rangel Henriques,
and Jorge Sousa Pinto

Departamento de Informática e CCTC
Universidade do Minho
Braga, Portugal

Abstract. In software development, it is often desirable to reuse existing software components. This has been recognized since 1968, when Douglas McIlroy of Bell Laboratories proposed basing the software industry on reuse. Despite the failures in practice, many efforts have been made to make this idea successful.
In this context, we address the problem of reusing annotated components as a rigorous way of assuring the quality of the application under construction. We introduce the concept of *caller-based slicing* as a way to certify that the integration of an annotated component with a contract into a legacy system will preserve the behavior of the former.
To complement the efforts done and the benefits of the slicing techniques, there is also a need to find an efficient way to visualize the annotated components and their slices. To take full profit of visualization, it is crucial to combine the visualization of the control/data flow with the textual representation of source code. To attain this objective, we extend the notion of System Dependence Graph and slicing criterion.

Keywords: safety reuse, caller-based slicing, annotated system dependency graph.

1. Introduction

Reuse is a very simple and natural concept, however in practice it is not so easy. According to the literature, selection of reusable components has proven to be a difficult task [9]. Sometimes this is due to the lack of maturity on supporting tools that should easily find a component in a repository or library [11]. Also, non experienced developers tend to reveal difficulties when describing the desired component in technical terms. Most of the times, this happens because they are not sure of what they want to find [11, 12]. Another barrier is concerned with reasoning about component similarities in order to select the one that best fits in the problem solution; usually this is an hard mental process [9].

Integration of reusable components has also proven to be a difficult task, since the process of understanding and adapting components is difficult, even for experienced developers [9]. Another challenge to component reuse is to certify that the integration of such component in a legacy system is correct. This is, to verify that the way the component is invoked will not lead to an incorrect behavior.

A strong demand for formal methods that help programmers to develop correct programs has been present in software engineering for some time now. The Design by Contract (DbC) approach to software development [10] facilitates modular verification and certified code reuse. The contract for a component (a procedure) can be regarded as a form of enriched software documentation that fully specifies the behavior of that component. So, a well-defined annotation can give us most of the information needed to integrate a reusable component in a new system, as it contains crucial information about some constraints safely obtaining the correct behavior from the component.

In this context, we say that the annotations can be used to verify the validity of every component's invocation; in that way, we can guarantee that a correct system will still be correct after the integration of that component. This is the motivation for our research: to find a way to help on the safety reuse of components.

This article introduces GamaPolarSlicer, a tool that we are currently developing to identify when an invocation is violating the component annotation, and display, whenever possible, a diagnostic or guidelines to correct it. For such a purpose, the tool implements the **caller-based slicing** algorithm, that takes into account the calls of an annotated component to certify that it is being correctly used.

The remainder of this paper is structured into 5 sections. Section 2 is devoted to basic concepts. In this section the theoretical foundation for GamaPolarSlicer is settle down; the notions of caller-based slicing and annotated system dependence graph are defined. Section 3 gives a general overview of GamaPolarSlicer, introducing its architecture; each block on the diagram will be explained. Sub-section 4 complements the architecture discussing the decisions taken to implement the tool and presenting the interface underdevelopment. Section 5, also a central one, illustrates the main idea through a concrete example. As to our knowledge we do not known any tool similar to GamaPolarSlicer, in Section 6 we discuss related work concerned with the use of slicing technique for annotated programs. Then the paper is closed in Section 7.

2. Basic Concepts

We consider that each procedure consists of a body of code, annotated with a precondition and a postcondition that form the procedure specification, or *contract*. The body may additionally be annotated with loop invariants. Occurrences of variables in the precondition and postcondition of a procedure refer to their values in the pre-state and post-state of execution of the procedure respectively.

2.1. Caller-based slicing

In this section, we briefly introduce our slicing algorithm.

Definition 1 (Annotated Slicing Criterion) *An* annotated slicing criterion *of a program $\mathcal{P}$ consists of a triple $C_a = (a, S_i, V_s)$, where $a \in \{\alpha, \delta\}$ is an annotation* *of $\mathcal{P}_a$ (the annotated callee), S_i correspond to the statement of $\mathcal{P}$ calling $\mathcal{P}_a$ and V_s is a subset of the variables in $\mathcal{P}$ (the caller), that are the actual parameters used in the call and constrained by α or δ.*

Definition 2 (Caller-based slicing) *A* caller-based slice *of a program $\mathcal{P}$ on an annotated slicing criterion $C_a = (\alpha, call_f, V_s)$ is any subprogram $\mathcal{P}'$ that is obtained from $\mathcal{P}$ by deleting zero or more statements in a two-pass algorithm:*

1. *a first step to execute a backward slicing with the traditional slicing criterion $C = (call_f, V_s)$ retrieved from C_a — $call_f$ corresponds to the call statement under consideration, and V_s corresponds to the set of variables present in the invocation $call_f$ and intervening in the precondition formula (α) of f*
2. *a second step to check if the statements preceding the $call_f$ statement will lead to the precondition satisfaction of the callee;*

For the second step in the two-pass algorithm, in order to check which statements are respecting or violating the precondition we are using abstract interpretation, in particular symbolic execution.

According to the original idea of *James King* in [7], symbolic execution can be described as “instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values. The execution proceeds as in a normal execution except that values may be symbolic formulas over the input symbols.”

Using symbolic execution we will be able to propagate the precondition of the function being called through the statements preceding the call statement. In particular, to integrate symbolic execution with our system, we are thinking in use JavaPathFinder [1]. JavaPathFinder is a tool than can perform program execution with symbolic values. Moreover, JavaPathFinder can mix concrete and symbolic execution, or switch between them. JavaPathFinder has been used for finding counterexamples to safety properties and for test input generation.

The main goal of our caller-based slicing algorithm is to ease the use of annotated components by discovering statements that are critical for the satisfaction of the precondition or postcondition (i.e, that do not verify it, or whose value can lead to the non-satisfaction) before or after calling an annotated procedure (a tracing call analysis of annotated procedures). In the work reported here, we just deal with preconditions and statements before the call.

2.2. Annotated System Dependence Graph (SDGa)

In this section we present the definition of Annotated System Dependence Graph, SDG$_a$ for short, that is the internal representation that supports our slicing-based code analysis approach. We start with some preliminary definitions.

Definition 3 (Procedure Dependence Graph) *Given a procedure $\mathcal{P}$, a Procedure Dependence Graph, PDG, is a graph whose vertices are the individual statements and predicates (used in the control statements) that constitute the* *body of $\mathcal{P}$, and the edges represent control and data dependencies among the vertices.*

In the construction of the PDG, a special node, considered as a predicate, is added to the vertex set: it is called the *entry* node and is decorated with the procedure name.

A control dependence edge goes from a predicate node to a statement node if that predicate affects the execution of the statement. A data dependence edge goes from an assignment statement node to another node if the variable assigned at the source node is used (is referred to) in the target node.

Additionally to the natural vertices defined above, some extra assignment nodes are included in the PDG linked by control edges to the entry node: we include an assignment node for each formal input parameter, another one for each formal output parameter, and another one for each returned value — these nodes are connected to all the other by data edges as stated above. Moreover, we proceed in a similar way for each call node; in that case we add assignment nodes, linked by control edges to the call node, for each actual input/output parameter (representing the value passing process associated with a procedure call) and also a node to receive the returned values.

Definition 4 (System Dependence Graph) *A System Dependence Graph, SDG, is a collection of Procedure Dependence Graphs, PDGs, (one for the main program, and one for each component procedure) connected together by two kind of edges: control-flow edges that represent the dependence between the caller and the callee (an edge goes from the call statement into the entry node of the called procedure); and data-flow edges that represent parameter passing and return values, connecting actual$_{in,out}$ parameter assignment nodes with formal$_{in,out}$ parameter assignment nodes.*

Definition 5 (Annotated System Dependence Graph) *An Annotated System Dependence Graph, SDG$_a$, is a SDG in which some nodes of its constituent PDGs are annotated nodes.*

Definition 6 (Annotated Node) *Given a PDG for an annotated procedure $\mathcal{P}_a$, an Annotated Node is a pair $< S_i, a >$ where S_i is a statement or predicate (control statement or entry node) in $\mathcal{P}_a$, and a is its annotation: a pre-condition α, a post-condition ω, or an invariant δ.*

The differences between a traditional SDG and a SDG$_a$ are:

- Each procedure dependence graph (PDG) is decorated with a precondition as well as with a postcondition in the entry node;
- The *while* nodes are also decorated with the loop invariant (or true, in case of invariant absence);
- The *call* nodes include the pre- and postcondition of the procedure to be called (or true, in case of absence); these annotations are retrieved from the respective PDG and instantiated as explained below;

We can take advantage from the *call linkage dictionary* present in the SDG_a (inherited from the underlying SDG) — the mapping between the variables present in the call statement (the actual parameters) and the formal parameters of the procedure — to associate the variables used in the calling statement with the formal parameters involved in the annotations. Figure 1 shows an example of a SDG_a.

3. GamaPolarSlicer Architecture

As referred previously, our goal is to ease the process of incorporating an annotated component into an existent system. This integration should be smooth, in the sense of that it should not turn a correct system into an incorrect one.

To assure this, there is the need to verify a set of conditions with respect to the annotated component and its usage. It is necessary to:

– to verify the component correctness with the respect to its contract (using a traditional *Verification Condition Generator*, already incorporated in GamaSlicer [5], available at `http://gamaepl.di.uminho.pt/gamaslicer`);
– to verify if the actual calling context preserves the precondition;
– to verify if the component is properly used in the actual context after the call;
– Given a reusable component and a set of calling points, specify the component body according to the concrete calling needs.

The whole process is a bit complex and was divided in a set of smaller problems (*divide and conquer*). The tool under discussion in this document will only focus on the second item, working with preconditions and backward slicing. Notice that the third and fourth conditions will be addressed by future projects.

The chosen architecture, designed to achieve the second condition, was based on the classical structure of a language processor. Figure 2 shows the defined GamaPolarSlicer architecture.

Source Code can be a Java project or only Java files to analyze by the tool.

Lexical Analysis, Syntactic Analysis, Syntactic Analysis: the Lexical layer converts the input into symbols that will be later used in the identifiers table. The Syntactic layer uses the result of the Lexical layer above and analyzes it to identify the syntactic structure of it. The Semantic layer adds the semantic information to the result from the Syntactic layer. It is in this layer that the identifier table is built. These three layers, usually are always present in language processors.

Invocations Repository is the data structure where all function calls processed during the code analysis are stored. The contract verification will be applied to each one of these calls and the slicing criterion of each one will consider the parameters struct.

Annotated Components Repository is the data structure where all components with a formal specification (precondition and postcondition at least)

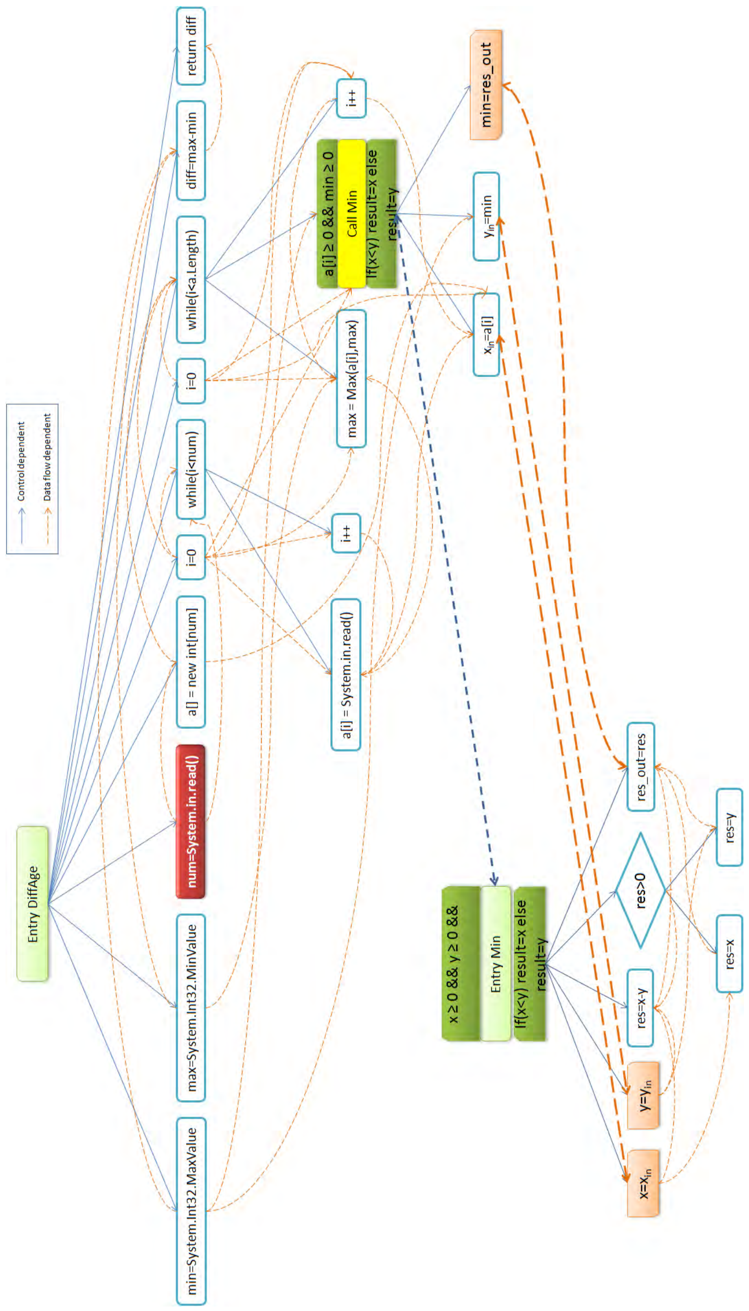


Fig. 1. SDG$_a$ for a program

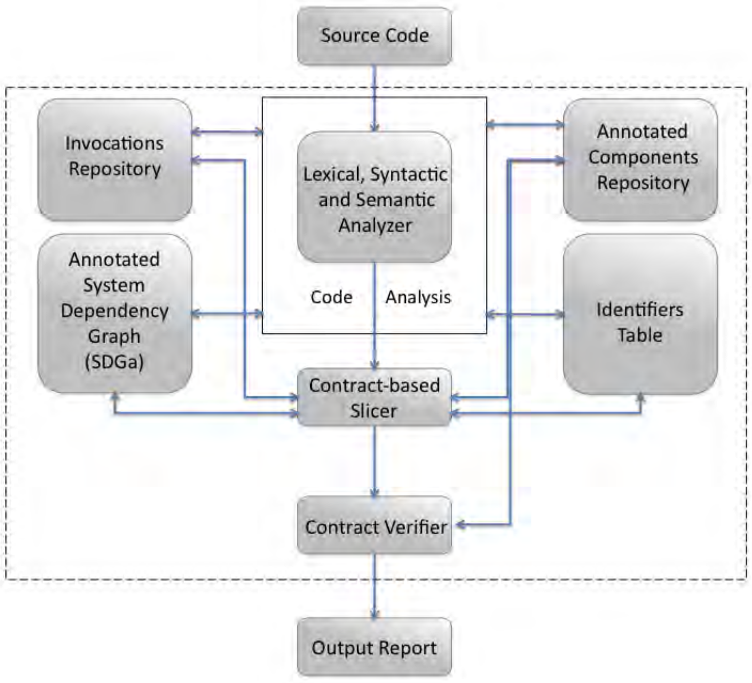


Fig. 2. GammaPolarSlicer Architecture

are stored. All these components will be later used in the slicing process in order to filter all the calls (from the invocation repository) defined without any type of annotation. This repository has an important role when verifying if the call respects the component contract.

Identifiers Table flags, always, an important role on the implementation of the processor. All symbols and associated semantic processed during the code analysis phase are stored here. It will be one of the backbones of all structures and of all stages of the tool process.

Annotated System Dependence Graph is the internal representation chosen to support our slicing-based code analysis approach. Constructed during the code analysis, this type of graph allows to associate formal annotations , like preconditions, postconditions or even invariants, to the its nodes (see Section 2.2).

Caller-based Slicing is the layer where the backward slicing is applied to each annotated component call. It uses both invocations repository and annotated components repository to extract the parameters to execute the slicing for each invoked annotated component. The resulting slice is a SDG_a this a subgraph of the original SDG_a, with all the statements relevant to the particular call.

Contract Verification using the slice that resulted from the layer above, and using the component contract, this layer analyzes every node on the slice and verifies in all of them if there are guarantees that every annotation in the contract is respected.

Output Report describes all contract violations found during the whole process. All violations found are marked with the degree of relevance in order to aid the user in the revision process. In the future, the tool will provide some suggestions to solve these issues, and a graphic display of the violations over the SDG_a.

4. GamaPolarSlicer Implementation

To address all the ideas, approaches and techniques presented in this paper, it was necessary to choose the most suitable technologies and environments to support the development.

To address the *design-by-contract* approach we decide to use the Java Modeling Language (JML) [1]. JML is a formal behavior interface specification language, based on *design-by-contract* paradigm, that allows code annotations in Java programs [8]. JML is quite useful as it allows to describe how the code should behave when running it. Also it allows the specification of the syntactic interface [8]. Preconditions, postconditions and invariants are examples of formal specifications that JML provides.

[1] http://www.cs.ucf.edu/ leavens/JML/

As the goal of the tool is not to create a development environment but to support one, our first thought was to implement it as an Eclipse [2] plugin. The major reasons that led to this decision were:

– the large community and support. Eclipse is one of the most popular frameworks to develop Java applications and thus a perfect tool to test our goal;
– the fact that it includes a great environment to develop new plugins. The Plugin Development Environment (PDE) [3] that allows a faster and intuitive way to develop Eclipse plugins;
– the built-in support for JML, freeing us from checking the validity of such annotations.

After the first days of the development process we realized that Java has a limitation regarding the number of bytes per class (only allows 65535 bytes per class). This limitation prevented us of continue the work with Java because the parser we were generating for Java/JML grammar exceeded this limit of bytes. This led us to abandon the idea of the Eclipse plugin and implement GamaPolarSlicer using Windows Forms and C# (under .NET framework). Figure 3 shows the current interface of GamaPolarSlicer.

4.1. Tool Workflow

As depicted in the architecture (see Figure 2), our tool is divided in a set of phases where each one solves a particular task. In this section we will explain how these phases interact with each other and how data flows between them.

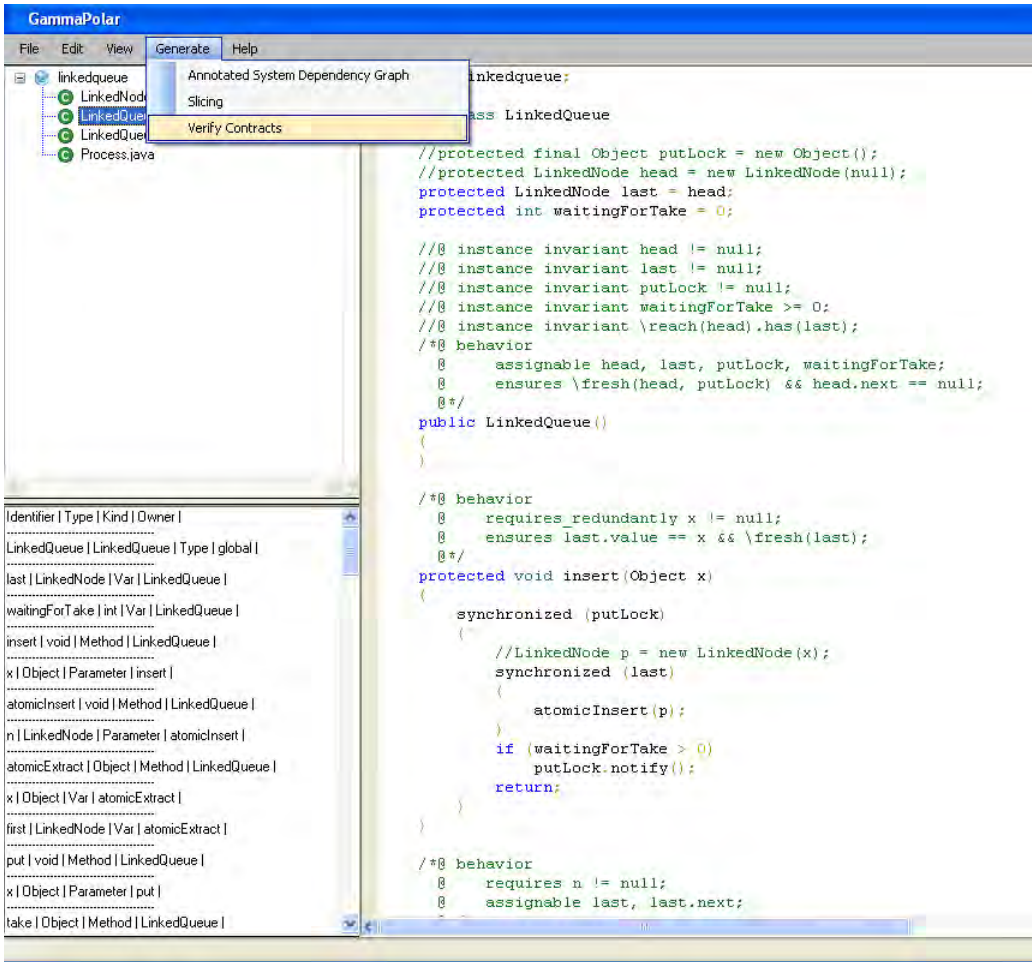
The tool begins analyzing the source code (Java code/JML annotations) in order to extract all symbols and to construct all data structures. In order to ease the slicing process it is mandatory to have an appropriate data structure to support this type of techniques. For this job we have chosen the SDG_a(SDG_a) (see Section 2.2). Using all the gathered information during the code analysis we are able to construct this graph.

The graph and the Identifier Table construction are made once for each input file processed. At the end of these steps, the system will have a set of Identifier Tables and a set of SDG_a. The union between all the SDG_a will result in the SDG_a for the entire source code. The same happens to the set of Identifier Table.

After building all the data structures, the backward slicing is then applied to a component invocation and the resulting slices together with the component contract are used to verify if its call respects the contract. These steps are applied to the set of calls resulting of the intersection between the Invocation Repository and the Annotated Components Repository.

During this process (depicted in the Figure 4), if a violation is found, a textual report is issued. Also a graphic report can be selected. This graphic report uses the constructed SDG_a.

[2] http://www.eclipse.org/
[3] http://www.eclipse.org/pde/



```
Identifier | Type | Kind | Owner |
------------------------------
LinkedQueue | LinkedQueue | Type | global |
------------------------------
last | LinkedNode | Var | LinkedQueue |
------------------------------
waitingForTake | int | Var | LinkedQueue |
------------------------------
insert | void | Method | LinkedQueue |
------------------------------
x | Object | Parameter | insert |
------------------------------
atomicInsert | void | Method | LinkedQueue |
------------------------------
n | LinkedNode | Parameter | atomicInsert |
------------------------------
atomicExtract | Object | Method | LinkedQueue |
------------------------------
x | Object | Var | atomicExtract |
------------------------------
first | LinkedNode | Var | atomicExtract |
------------------------------
put | void | Method | LinkedQueue |
------------------------------
x | Object | Parameter | put |
------------------------------
take | Object | Method | LinkedQueue |
```

```
inkedqueue;

ass LinkedQueue

//protected final Object putLock = new Object();
//protected LinkedNode head = new LinkedNode(null);
protected LinkedNode last = head;
protected int waitingForTake = 0;

//@ instance invariant head != null;
//@ instance invariant last != null;
//@ instance invariant putLock != null;
//@ instance invariant waitingForTake >= 0;
//@ instance invariant \reach(head).has(last);
/*@ behavior
  @     assignable head, last, putLock, waitingForTake;
  @     ensures \fresh(head, putLock) && head.next == null;
  @*/
public LinkedQueue()
{
}

/*@ behavior
  @    requires_redundantly x != null;
  @    ensures last.value == x && \fresh(last);
  @*/
protected void insert(Object x)
{
    synchronized (putLock)
    {
        //LinkedNode p = new LinkedNode(x);
        synchronized (last)
        {
            atomicInsert(p);
        }
        if (waitingForTake > 0)
            putLock.notify();
        return;
    }
}

/*@ behavior
  @    requires n != null;
  @    assignable last, last.next;
```

Fig. 3. Interface of GamaPolarSlicer prototype

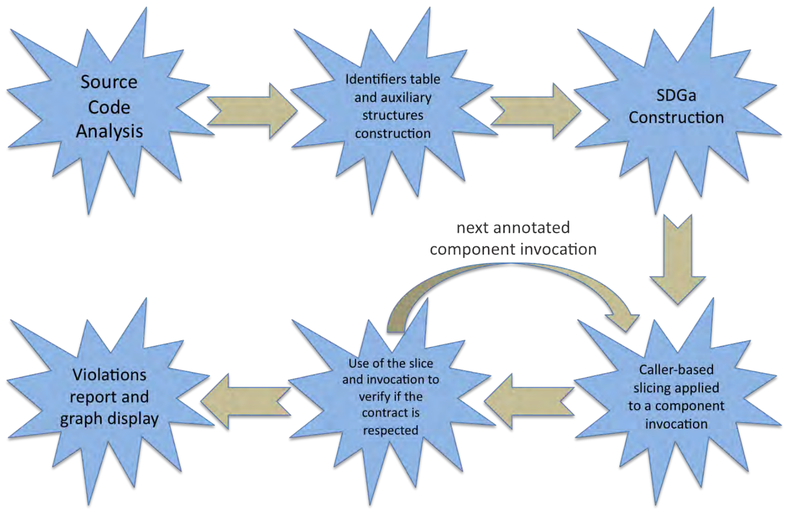

Source Code Analysis
Identifiers table and auxiliary structures construction
SDGa Construction
next annotated component invocation
Caller-based slicing applied to a component invocation
Use of the slice and invocation to verify if the contract is respected
Violations report and graph display

Fig. 4. Tool Workflow

4.2. Contract Verification Strategies

As already shown, the contract verification is applied upon the slices that result from the caller-based slicing process. This implies the verification of all statements on the slices to check possible violations. Depending on the statement type, there are a few critical verifications that need to be made. For readable purposes, we will use the following notation in the remainder of this paper:

- **Call** refers to the function invocation for which we want to apply the contract verification;
- **Caller** is the component where the call occurs;
- **Callee** is the component invoked.

Please consider the example 1 with two annotated components, where one of the components invokes the other.

On the notes in red, we can see that one of the parameters of the call we want to verify is also a parameter on the caller. As the verification is only made on caller (as standalone component), there is no way to verify the value of the parameter at the beginning. This lead us to the first critical verification, precondition versus precondition.

Precondition vs Precondition When the call and the caller share a parameter we decided to certify it value using the caller precondition. Doing this, we have three possible cases:

1. the caller has an annotation for the parameter and the callee does not;

Example 1 Precondition violation

```
/ $*$ @ behavior
@ requires $a > 0;$
@ ensures $pot = a^b;$
@ $*$ /
public int sqr(int a, int b) {
    $int\ pot = 1, i;$
    for(i=0;i¡b;i++) {
        $pot = mult(a, pot);$
    }
    return pot;
}
/ $*$ @ behavior
@ requires $c > 10$ && $d > 0;$
@ ensures $pot = c * d;$
@ $*$ /
public int mult(int c, int d) {
    $int\ res = c * d;$
    return res;
    }
```

2. the caller does not have an annotation for the parameter and the callee does;
3. both, the caller and the callee, have an annotation for the parameter.

In the first case, it is obvious that does not change anything. If the callee does not have an annotation for the parameter then it means the parameter can assume any value.

The second case brings ambiguity to the problem. If the caller does not have an annotation for the parameter, then there is no way to guarantee that its value will respect the clause on the call contract. Even if after the verification of all statements, the value respects the clause, that value will always be dependent of the value received as parameter on the caller.

The third case, and the most complex one, gives us chance to predict a value for the parameter on the call moment. With the annotation we can calculate or predict the set of values the parameter can take during the execution of the method. To do this we have created an object with a set of flags that tell us what type of value we have and the range of values that can take.

Please consider that we have the following annotation:

```
requires x>0 && x<200
```

After processing this annotation, the object will have the flags for values higher than, lower than and between activated. The between flag is activated when the annotation contains a closed interval.

These flags also help us to make comparisons between annotations. We can compare preconditions with preconditions and even preconditions with postconditions. The last one is very important to the second critical verification.

Precondition vs Postcondition Most of all pieces of source code have function calls. When the call of these functions affects the value of a parameter on the call that we are trying to verify, then forces the verification of their postcondition (if defined). This is what we will discuss in this section.

When we found a statement with a function call in the slice result, we verify if the invoked component exists on the loaded source code. If it is an external component, like one included from an imported library, then we have no way to guarantee that the program will work correctly after this point.

During the review process of one of our papers we received a question that raised questions for another issue. The question was, ”(. . .) depends on the (human) reader’s knowledge that an input function might not have return a positive integer (or even any number); but how does the slicer knows this?” (the given example was using integers). When we identify a call to an external function, we add an entry on the output report with a warning, alerting to the fact that a few verifications must be make in order to guarantee that all calls, to an annotated component, that receive value as parameter, will have the contract respected. We recognize this type of functions using all the data structures constructed during the analysis process. If a call is found in a slicing result, but has no entry on the identifier table, then is considered a call to an external function.

Everything discussed until now in this section happen when found a call to an external function. But how about, when the function is on the identifier table and on the repositories? When this happen we have three possible cases:

1. the call we are verifying the contract has no annotation for the parameter with the resulting value of the function call;
2. the found call has no postcondition and the call we are verifying has an annotation for the parameter with the resulting value of the function call;
3. the found call has postcondition and the call we are verifying has an annotation for the parameter with the resulting value of the function call;

In the first case, the result of the found call makes no difference as the parameter has no restrictions of value.

The second case will generate a warning message as we are not able to predict the values of the parameter making impossible to guarantee that the contract will be respected.

The last case force the calculation of the possible values, to be used on the next iterations, using the postcondition. All the information is stored in the objects already seen. These objects are later used to compare the postcondition and precondition annotations regarding a particular parameter in order to find contract violations.

Values vs Precondition This last critical verification occurs every time during of the verification of the statements on the slicing result. Each time the parameter suffers a change, the values it can take must be recalculated. This may look easier than it really is.

If we have an assignment it is pretty easy to calculate the new value but if we have the same assignment inside an `if` block, for example, the complexity increases significantly. We must assure that both values (if the condition is true and if it is not) are used to compare with the call precondition.

Having all this in consideration, we decided to use a flexible list in order to store the list of values the parameter can accept. Every time we found a new path in the code to reach the call we are verifying, we create a new entry on the list with the calculated value. The way we have defined the object, seen in section 4.2, also allow us to compare values with annotations.

In case of violations, these comparisons always lead to error messages. At this point we are able to find contract violations without any doubts so there is no reason to generate warning messages.

4.3. Tool Views

The assessement made to other tools, developed under our group, have shown that a variety of views are in some way needed to work with the tool, and give a better understanding of its results. Following this, the tool provides four different types of views: the Code View, the Identifier Table View, the $\mathtt{SDG}_a$ View and the Slicing View.

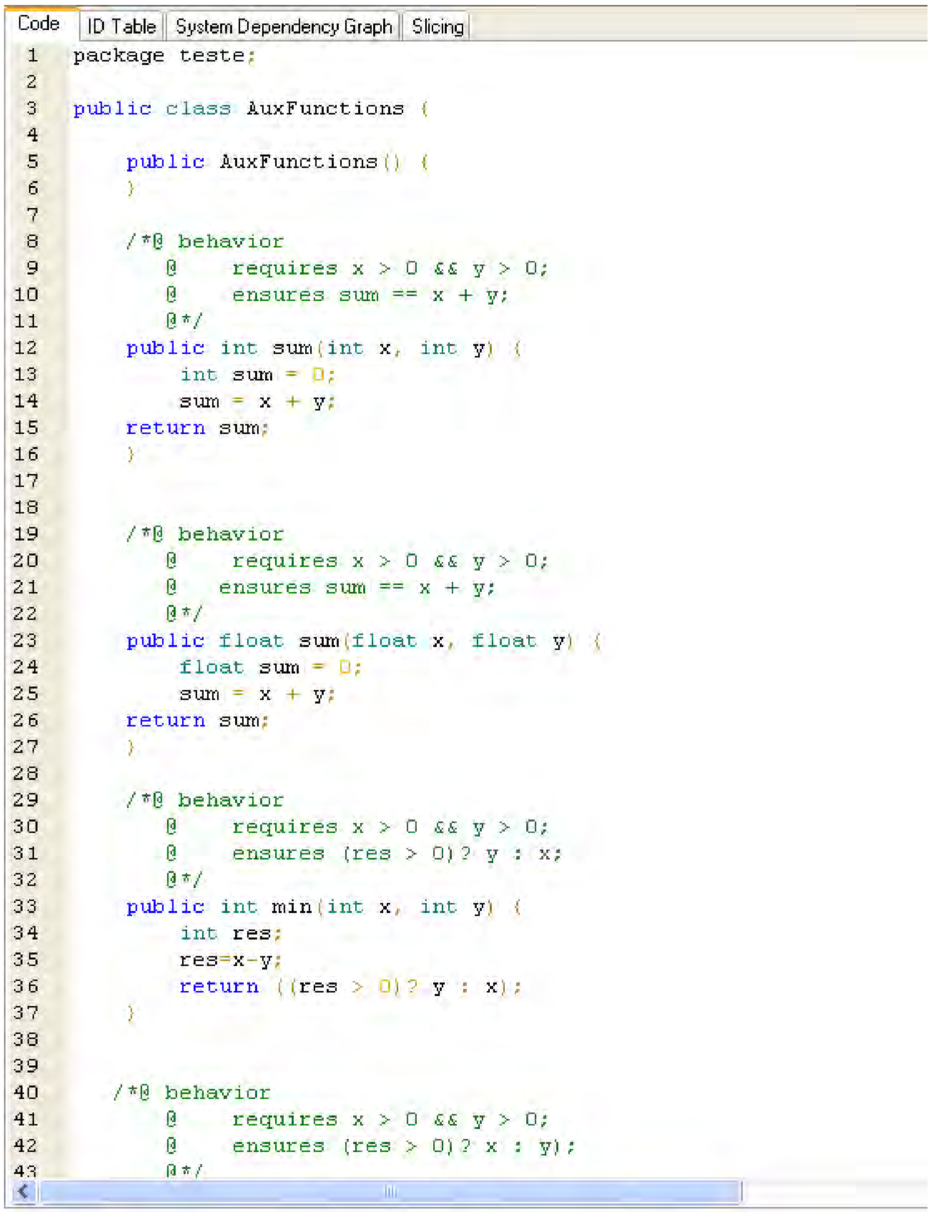
The user has access to the Code view (Figure 5) as the default view where has access to the source code that will be used as input in the verification. The Java code is highlighted using scintillaNet[4] library to improve its readability. This is a library that can be imported by Visual Studio, that provides us a special text box where we can define all the color definitions we want to highlight our code.

The Identifier Table view shows the information collected for all symbols in the selected class. The information can be filtered in order to visualize only the details of the symbols in a particular method selected by the user. Figure 6 shows the identifier table of an entire class.

The $\mathtt{SDG}_a$ view shows all the control flow dependencies and data flow dependencies present on the source code. In order to help to visualize which contracts and statements are being violated, we display the SDG_a with such entities colored in red. The Figure 7 shows the representation of a graph by the tool.

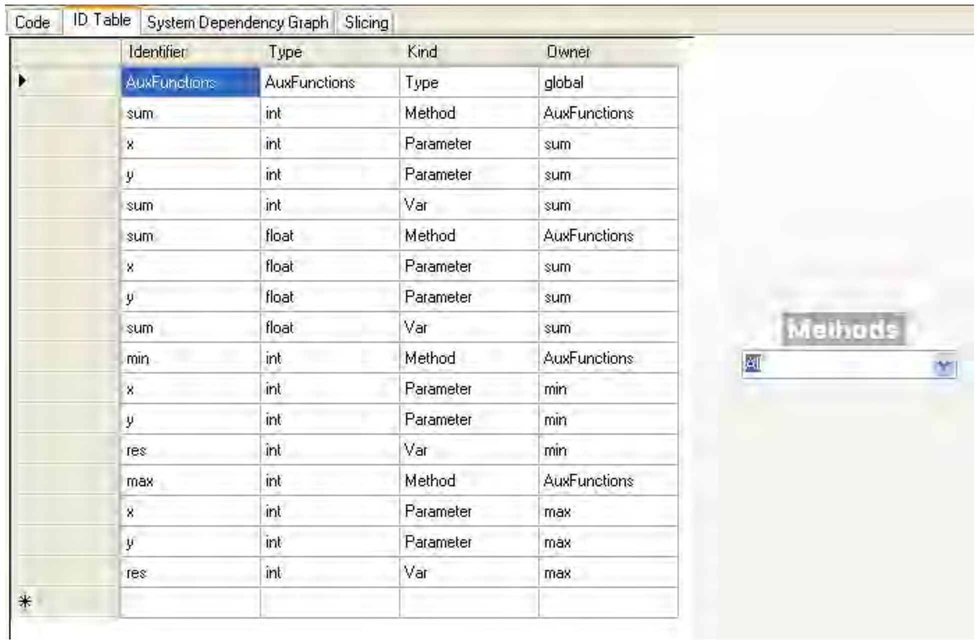
The Slicing view (Figure 8) was included to show the result of an intermediate calculation. As the slicing isolates the statements relevant to a particular call (the statement that we are interested to our work), then will probably ease in the understanding and correctness of any error found during the verification.

[4] http://scintillanet.codeplex.com/



```
package teste;

public class AuxFunctions {

    public AuxFunctions() {
    }

    /*@ behavior
       @     requires x > 0 && y > 0;
       @     ensures sum == x + y;
       @*/
    public int sum(int x, int y) {
        int sum = 0;
        sum = x + y;
    return sum;
    }


    /*@ behavior
       @     requires x > 0 && y > 0;
       @    ensures sum == x + y;
       @*/
    public float sum(float x, float y) {
        float sum = 0;
        sum = x + y;
    return sum;
    }

    /*@ behavior
       @     requires x > 0 && y > 0;
       @     ensures (res > 0)? y : x;
       @*/
    public int min(int x, int y) {
        int res;
        res=x-y;
        return ((res > 0)? y : x);
    }


   /*@ behavior
       @     requires x > 0 && y > 0;
       @     ensures (res > 0)? x : y);
       @*/
```

Fig. 5. GamaPolarSlicer Code View



Identifier	Type	Kind	Owner
AuxFunctions	AuxFunctions	Type	global
sum	int	Method	AuxFunctions
x	int	Parameter	sum
y	int	Parameter	sum
sum	int	Var	sum
sum	float	Method	AuxFunctions
x	float	Parameter	sum
y	float	Parameter	sum
sum	float	Var	sum
min	int	Method	AuxFunctions
x	int	Parameter	min
y	int	Parameter	min
res	int	Var	min
max	int	Method	AuxFunctions
x	int	Parameter	max
y	int	Parameter	max
res	int	Var	max

Fig. 6. GamaPolarSlicer Identifier Table View

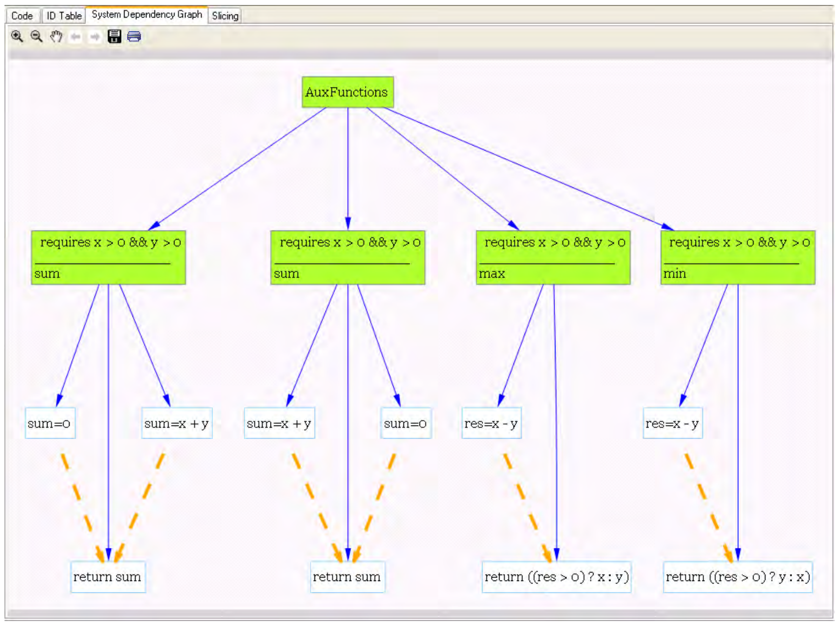
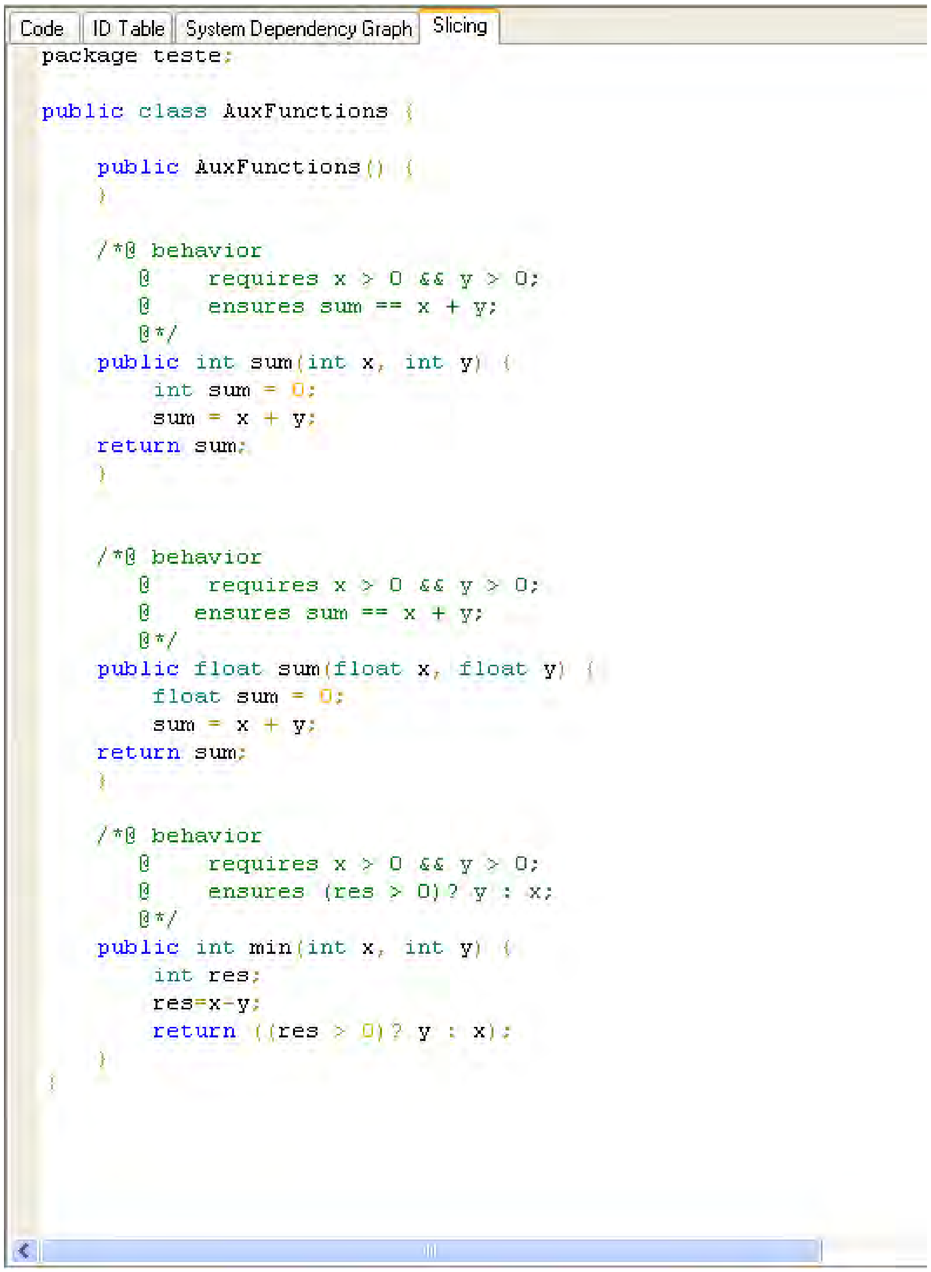


Fig. 7. GamaPolarSlicer SDG$_a$ View



```
package teste;

public class AuxFunctions {

    public AuxFunctions() {
    }

    /*@ behavior
       @     requires x > 0 && y > 0;
       @     ensures sum == x + y;
       @*/
    public int sum(int x, int y) {
        int sum = 0;
        sum = x + y;
    return sum;
    }


    /*@ behavior
       @     requires x > 0 && y > 0;
       @    ensures sum == x + y;
       @*/
    public float sum(float x, float y) {
        float sum = 0;
        sum = x + y;
    return sum;
    }

    /*@ behavior
       @     requires x > 0 && y > 0;
       @     ensures (res > 0)? y : x;
       @*/
    public int min(int x, int y) {
        int res;
        res=x-y;
        return ((res > 0)? y : x);
    }
}
```

Fig. 8. GamaPolarSlicer Slicing View

5. An illustrative example

To illustrate what we intended to achieve, please consider the Example 2 listed below that computes the maximum difference among student ages in a class. This component reuses other two: the annotated component `Min`, defined in Example 3, that returns the lowest of two positive integers (Figure 5 shows the view of the code provided by GamaPolarSlicer); and Max, defined in Example 4, that returns the greatest positive integer.

Example 2 DiffAge

```
public int DiffAge() {
      int min = System.Int32.MaxValue;
      int max = System.Int32.MinValue;
      int diff;

      System.out.print(”Number of elements: ”);
      int num = System.in.read();
      int[] a = new int[num];
      for(int i=0; i¡num; i++) {
            a[i] = System.in.read();
      }

      for(int i=0; i¡a.Length; i++) {
            max = Max(a[i],max);
            min = Min(a[i],min);
      }

      diff = max - min;
      System.out.println(”The gap between max and min age is ” + diff);
      return diff;
}
```

Example 3 Min

$/*@$ requires $x \geq 0$ && $y \geq 0$
$@$ ensures $(x > y)?$ \result $== x$: \result $== y$
$@*/$

```
public int Min(int x, int y) {
int res;
$res = x - y$;
return (($res < 0$)? x : y);
}
```

Example 4 Max

$/*@$ requires $x \geq 0$ && $y \geq 0$
$@$ ensures $(x > y)?$ \result $==y$: \result $==x$
$@*/$

```
public int Max(int x, int y) {
int res;
$res = x - y$;
return ((res > 0)? x : y);
}
```

Let us consider that we want to analyze the `Min` invocation present in the `DiffAge` component.

Our slicing criterion will be: $C_a = (x \geq 0\&\&y \geq 0, Min, \{a[i], min\})$

In the second step, a backward slicing process is performed taking into account the variables present in V_s. Then, using the obtained slices, the detection of contract violations starts. For that, the precondition is back propagate (using symbolic execution) along the slice to verify if it is preserved after each statement. Observing the slice for the variable `a[i]`, listed in the example 5 below, it can not be guaranteed that all integer elements are greater than zero; so a potential precondition violation is detected.

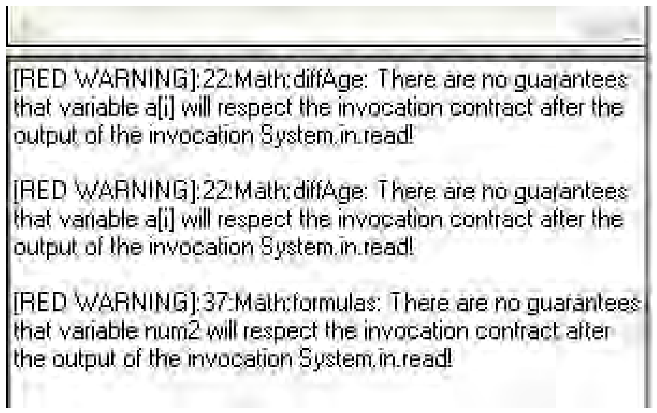
Example 5 Backward Slicing for a[i]

```
    int[] a = new int[num];
    for(int i=0; i<num; i++) {
        a[i] = System.in.read();
    }
    for(int i=0; i<a.Length; i++) {
        max = Max(a[i],max);
        min = Min(a[i],min);
    }
```

The third step consists in the notification of all the contract violations detected. In the example above, the user will receive a *warning* (Figure 9 shows the output report from GamaPolarSlicer) alerting to the possible invocation of Min with negative numbers (what does not respect the precondition).

6. Related Work

In this section we review the published work on the area of slicing annotated programs, as those contributions actually motivate the present proposal.



```
[RED WARNING]:22:Math:diffAge: There are no guarantees
that variable a[i] will respect the invocation contract after the
output of the invocation System.in.read!

[RED WARNING]:22:Math:diffAge: There are no guarantees
that variable a[i] will respect the invocation contract after the
output of the invocation System.in.read!

[RED WARNING]:37:Math:formulas: There are no guarantees
that variable num2 will respect the invocation contract after
the output of the invocation System.in.read!
```

Fig. 9. GamaPolarSlicer Output Report

In [4], *Comuzzi et al* present a variant of program slicing called *p-slice* or *predicate slice*, using Dijkstra's weakest preconditions (wp) to determine which statements will affect a specific predicate. Slicing rules for assignment, conditional, and repetition statements were developed. They presented also an algorithm to compute the minimum slice.

In [3], *Chung et al* present a slicing technique that takes the specification into account. They argue that the information present in the specification helps to produce more precise slices by removing statements that are not relevant to the specification for the slice. Their technique is based on the weakest precondition (the same present in *p-slice*) and strongest post-condition — they present algorithms for both slicing strategies, backward and forward.

Comuzzi et al [4], and *Chung et al* [3], provide algorithms for code analysis enabling to identify suspicious commands (commands that do not contribute to the postcondition validity).

In [6], *Harman et al* propose a generalization of conditioned slicing called pre/post conditioned slicing. The basic idea is to use the pre-condition and the negation of the post-condition in the conditioned slicing, combining both forward and backward conditioning. This type of program slicing is based on the following rule: "Statements are removed if they cannot lead to the satisfaction of the negation of the post condition, when executed in an initial state which satisfies the pre-condition". In case of a program which correctly implements the pre- and post-condition, all statements from the program will be removed. Otherwise, those statements that do not respect the conditions are left, corresponding to statements that potentially break the conditions (are either incorrect or which are innocent but cannot be detected to be so by slicing). The result of this work can be applied as a correctness verification for the annotated procedure.

7. Conclusion

As can be seen in section 5, the motivation for our research is to apply slicing, a well known technique in the area of source code analysis, to create a tool that aids programmers to build safety programs reusing annotated procedures.

The tool under construction, GamaPolarSlicer, was described in Section 3. Its architecture relies upon the traditional compiler structure; on one hand, this enables the automatic generation of the tool core blocks, from the language attribute grammar; on the other hand, it follows an approach in which our research team has a large knowhow (apart from many DSL compilers, we developed a lot of Program Comprehension tools: Alma, Alma2, WebAppViewer, BORS, and SVS). The new and complementary blocks of GamaPolarSlicer implement slice and graph-traversal algorithms that have a sound basis, as described in Section 2; this allows us to be confident in there straight-forward implementation.

At the moment, the tool is capable to apply the Caller-based Slicing to a program and compute precise slices. Also the computed slices are displayed by the tool to ease the comprehension of the program by the developer, allowing him to focus on the relevant aspects of the program. This tool is also very useful on the program comprehension on its general.

One of our goals was to check if it was possible to do a contract verification with low computing effort and reasonable precision. This was successfully accomplished but we still need more empirical studies so that we can strengthen our conclusions regarding its efficiency and reliability.

The tool still presents some limitations in the contract verification process. Expand it in order to process annotations regarding any Java data type does not appear to be an easy job and can make the tool to become less accurate.

GamaPolarSlicer will be included in Gama project (for more details see `http://gamaepl.di.uminho.pt/gama/index.html`). This project aims at mixing specification-based slicing algorithms with program verification algorithms to analyze annotated programs developed under Contract-base Design approach. GamaSlicer is the first tool built under this project for intra-procedural analysis that is available at `http://gamaepl.di.uminho.pt/gamaslicer/`.

To test the behaviour of our algorithms and the tool output and performance, we selected a collection of diversified programs (medium size and medium complexity). As the objective of that phase was to verify the correctness of the algorithms and its coverage, we were exclusively concerned with the variety of the test cases concerning call paths and the kind of contract annotations. After that phase, that finished successfully, we will be concerned with scalability. For that purpose, a new collection of tests of large size will be used to measure the performance degeneration. We are aware that the biggest difficulty we will face is to find in the industrial or academic worlds huge programs with annotations, but we need to obtain them (even generating the test cases) to fully test GamaPolarSlicer.

Although reuse was not the topic of the paper (just some considerations were drawn in the Introduction), reuse is the main motivation for GamaPolarSlicer development. We are preparing an experiment to assess the validity of our proposal and the usefulness of the tool [2].

References

1. Anand, S., Păsăreanu, C.S., Visser, W.: Jpf-se: a symbolic execution extension to java pathfinder. In: TACAS'07: Proceedings of the 13th international conference on Tools and algorithms for the construction and analysis of systems. pp. 134–138. Springer-Verlag, Berlin, Heidelberg (2007)
2. Areias, S.: Contracts and Slicing for Safety Reuse. Master's thesis (Dec 2010)
3. Chung, I.S., Lee, W.K., Yoon, G.S., Kwon, Y.R.: Program slicing based on specification. In: SAC '01: Proceedings of the 2001 ACM symposium on Applied computing. pp. 605–609. ACM, New York, NY, USA (2001)
4. Comuzzi, J.J., Hart, J.M.: Program slicing using weakest preconditions. In: FME '96: Proceedings of the Third International Symposium of Formal Methods Europe on Industrial Benefit and Advances in Formal Methods. pp. 557–575. Springer-Verlag, London, UK (1996)
5. da Cruz, D., Henriques, P.R., Pinto, J.S.: Gamaslicer: an online laboratory for program verification and analysis. In: Proceedings of the 10th Workshop on Language Descriptions Tools and Applications (LDTA'10) (2010)
6. Harman, M., Hierons, R., Fox, C., Danicic, S., Howroyd, J.: Pre/post conditioned slicing. icsm 00, 138 (2001)
7. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
8. Leavens, G.T., Cheon, Y.: Design by contract with jml (2004)
9. Maiden, N.A.M., Sutcliffe, A.G.: People-oriented software reuse: the very thought. In: Advances in Software Reuse - Second International Workshop on Software Reusability. pp. 176–185. IEEE Computer Society Press (1993)
10. Meyer, B.: Applying "design by contract". Computer 25(10), 40–51 (1992)
11. Sherif, K., Vinze, A.: Barriers to adoption of software reuse a qualitative study. Inf. Manage. 41(2), 159–175 (2003)
12. Shiva, S.G., Shala, L.A.: Software reuse: Research and practice. In: ITNG. pp. 603–609. IEEE Computer Society (2007), `http://dblp.uni-trier.de/db/conf/itng/itng2007.html\#ShivaS07`

Sérgio Areias got his degree in "Computer Engineering", at University of Minho (UM) in 2008. In 2010, he finished the M.Sc. with the dissertation "Contracts and Slicing for Safety Reuse", also at University of Minho. Since 2011, he is working at University of Minho under the research project named CROSS (An Infrastructure for Certification and Re-engineering of Open Source Software), and the research team of "gEPL, the Language Processing group".

Daniela da Cruz received a degree in "Mathematics and Computer Science", at University of Minho (UM), and now she is a Ph.D. student of "Computer Science" also at University of Minho, under the MAPi doctoral program. She joined the research and teaching team of "gEPL, the Language Processing group" in

2005. She is teaching assistant in different courses in the area of Compilers and Formal Development of Language Processors; and Programming Languages and Paradigms (Procedural, Logic, and OO). She was also involved in several research projects (CROSS, DSLpc, PCVIA).

Pedro Rangel Henriques got a degree in ”Electrotechnical/Electronics Engineering”, at FEUP (Porto University), and finished a Ph.D. thesis in ”Formal Languages and Attribute Grammars” at University of Minho. In 1981 he joined the Computer Science Department of University of Minho, where he is a teacher/researcher. Since 1995 he is the coordinator of the ”Language Processing group” at CCTC (Computer Science and Technologies Center). He teaches many different courses under the broader area of programming: Programming Languages and Paradigms; Compilers, Grammar Engineering and Software Analysis and Transformation; etc. Pedro Rangel Henriques has supervised Ph.D. (11), and M.Sc. (13) thesis, and more than 50 graduating trainingships/projects, in the areas of: language processing (textual and visual), and structured document processing; code analysis, program visulaization/animation and program comprehension; knowledge discovery from databases, data-mining, and data-cleaning. He is co-author of the ”XML & XSL: da teoria a prática” book, publish by FCA in 2002; and has published 3 chapters in books, and 20 journal papers.

Jorge Sousa Pinto got his ”Docteur de l'École Polytechnique” degree in 2001. Since then he has been a lecturer at the University of Minho, and since 2005 he has served as the deputy director of the Computer Science and Technology Centre at that university. He is the author of about 30 research papers, and a co-author of the textbook ”Rigorous Software Development – An Introduction to Program Verification”. His research interests comprise both deductive and model checking-based approaches to program verification. He is a senior member of the Association for Computing Machinery.

Received: January 7, 2011; Accepted: April 13, 2011.

DOI: 10.2298/CSIS101220005W

Animation of Tile-Based Games Automatically Derived from Simulation Specifications

Jan Wolter, Bastian Cramer, and Uwe Kastens

University of Paderborn
Department of Computer Science
Fürstenallee 11, 33102 Paderborn, Germany
jwolter@mail.uni-paderborn.de, {bcramer, uwe}@uni-paderborn.de

Abstract. Visual Languages (VLs) are beneficial particularly for domain-specific applications, since they can support ease of understanding by visual metaphors. If such a language has an execution semantics, comprehension of program execution may be supported by direct visualization. This closes the gap between program depiction and execution.
To rapidly develop a VL with execution semantics a generator framework is needed which incorporates the complex knowledge of simulating and animating a VL on a high specification level.
In this paper we show how a fully playable tile-based game is specified with our generator framework DEViL. We illustrate this on the famous Pac-man[1] game.
We claim that our simulation and animation approach is suitable for the rapid development process. We show that the simulation of a VL is easily reached even in complex scenarios and that the automatically generated animation is mostly adequate, even for other kinds of VLs like diagrammatic, iconic or graph based ones.

Keywords: visual languages, DEViL, simulation, animation, tile-based games, pac-man

1. Introduction

A prominent representative of a visual language is the Unified Modeling Language (UML) [10] which is often used in software engineering process. Even smaller languages precoined for a specific domain are popular, because they can use visual metaphors of the target domain. In general an instance of such a visual language is used to produce source code of a different domain, e.g. Java Code from an UML class diagram.

To gain acceptance in rapid prototyping, generator frameworks are used which can generate graphical structure editors for such visual languages from high-level specifications. These generators incorporate expert knowledge to produce a complete development environment for a VL with all features known from typical text editors like cut and paste, printing, drag and drop and so on.

[1] Pac-man® is a registered trademark of Namco.

Unfortunately there is still a gap between program depiction and the generated code of that program. The programmer has to keep in mind what the program, he just created, does when it is executed. This gap is known as the gulf of execution [9]. Simulation and animation of the visual language instance can help to narrow this gap. The execution semantics of a visual language (if it has one) can be integrated into the visual language. Hence the VL instance is no longer static. It can be simulated and smoothly animated. The user can ”see” his language being executed before he generates code.

This helps to avoid mistakes at a very early stage and it supports program comprehension which is a challenging task especially in languages where many things happen in parallel.

The *Development Environment for Visual Languages*, DEViL, is a generator framework for visual languages which produces graphical editors from declarative high-level specifications. We extended it with simulation and animation support for VLs whereas a smooth and challenging animation can be derived automatically from a simple simulation specification. In this paper we want to show that our simulation specification language is powerful to simulate even complex behavior. We claim that the language helps in rapid prototyping, because simulation becomes an easy task due to powerful encapsulated concepts like event driven simulation and the extension of the visual semantic model to constitute a tailored simulation model. We will show that the automatically derived animation is suitable in most situations.

We will demonstrate this on the famous Pac-man game. It has a playful character, but it is also a challenging language for simulation, because of the complex navigation concepts of the ”ghost” pawns in the game.

The paper is structured as follows: First we introduce the DEViL system and its underlying specification concepts with particular attention to simulation and animation. In Section 3 we give a brief description of the Pac-man game. In the next section we present our Pac-man Editor with special attention to the strategies of the ghost characters. Section 5 addresses related work and section 6 completes the exposition with a conclusion and a look at other implemented languages.

2. The DEViL System

The DEViL framework generates syntax-directed graphical structure editors for visual languages. The generated environments support all features of commonly used editors. Especially 2.5D views on the underlying semantic model are supported. A more in depth look at the generator framework and its generated products with respect to usability can be found in [13].

DEViL has already been successfully used for projects with nameable companies like Bosch [3], VW or SagemOrga [15]. The specification of this Pacman Game Editor was one of many bachelor resp. master-theses that used the DEViL framework.

The specification process to generate ”static” environments - environments without simulation and animation support - is divided into three parts. As can be seen later in this paper, simulation and animation support can be extended easily by the reuse of components of some of these three specification steps. Hence, an user of the DEViL system who can build visual development environments, can extend a language with simulation and animation support with reasonable effort.

To generate a structure editor for DEViL one first specifies the semantic model of the visual language. This is done with DSSL (DEViL Structure Specification Language). The semantic model abstracts from the visual representation. It stores just the information necessary to describe the semantics of the visual program. DSSL is inspired by an object oriented design with classes, inheritance, attributes and references. Fig. 1 shows a part of the specification of the semantic model for our Pac-man Editor. DEViL can generate an editor with a tree based structure manipulation view from this part of the specification.

```
CLASS Tile {
  columnRef: REF Column;
  item: SUB Item?;
}
ABSTRACT CLASS Item {
  name: VAL VLString;
}
CLASS Pacman INHERITS Item {
  direction: VAL VLInt INIT "2";
  angle: VAL VLInt INIT "0";
  clockwise: VAL VLBoolean;
}
```

Fig. 1. Part of the semantic model for the Pac-man Editor.

To obtain an advanced visual representation, the semantic model (created with DSSL) is decorated with so called ”visual patterns”. Visual patterns define how constructs of the structure tree should look like. E.g. one can specify that some part of the structure tree should be laid out as the abstract concept ”list” and aggregated nodes play the role of ”list elements”. Control attributes may modify this layout, for instance the list could be constituted vertically instead of horizontally. DEViL provides a huge library of precoined visual patterns with various possibilities to adapt their layout and appearance. A subset of this library is for example ”sets, lists, trees, formulae or matrices”. Technically, symbols of the semantic structure definition inherit from these visual patterns. The attribute evaluator, generated by LIGA [4], of the underlying compiler generator framework, Eli [5], computes the final graphical representation.

The last (optional) step of the specification process is the definition of a code generator. Here, all of the tools of the Eli system to analyze the visual language instance can be used. A more detailed description of the VL specification process can be found in [14].

In order to separate concerns of specification, simulation and animation have to be distinguished: simulation is the raw execution semantics of the visual language and animation is the smooth depiction of discrete execution of VL programs. Some visual languages have a precisely defined execution semantics, e.g. the firing of tokens in a Petri-net may be smoothly depicted by animation. For other visual languages simulation and animation may require to extend the semantic model to represent the simulation states or its graphical representation.

The presented Pac-man Editor (Fig. 4) considered as a visual language has a number of pawns that can be placed on a tile-based board which constitute the playing field. The pawns are typed structure objects of this VL. The Pac-man Editor has only four different pawns: ”wall”, ”ghost”, ”powerpill” and ”pac-man”. Additionally, some structure objects are needed to represent the rows and columns of the board. Hence, our Pac-man VL is a playground editor where the user can create custom levels.

To specify a simulation for the Pac-man Editor we have to define the state space and the state transitions. Both can be specified in our simulation specification language DSIM.

```
MODEL {
  CLASS Tile {
    OBJECT pill OF PowerPill: "THIS.item.CHILDREN[0]";
    position: VAL VLPoint?;
    diffVal: VAL VLInt INIT "0";
    visited: VAL VLBoolean INIT "0";
  }
  CLASS Pacman {
    OBJECT tile OF Tile: "THIS.PARENT.PARENT";
  }
}
```

(a) Simulation model.

```
EVENTS {
  goGhost(Tile from, Tile to){
    Item i = REMOVE(from.item, FIRST);
    INSERT(to.item, i, FIRST);
  }
  eatPacman(Tile from, Tile to){
    IF(#[0]Root.sound == VLBoolean(1)){
      vlPlaySound("pacmanDeath.wav");
    }
    REMOVE(to.item, FIRST);
    Item i = REMOVE(from.item, FIRST);
    INSERT(to.item, i, FIRST);
    FIRE gameLost(#[0]Root)@TIME_NOW + 1;
  }
}
```

(b) Events.

Fig. 2. Simulation model in DSIM and some events which can be scheduled in the simulation loop.

Fig. 2 (a) shows the specification of the simulation model in DSIM. As can be seen, we again reuse DSSL concepts and we can extend the semantic model of the visual language to reach a new model that is suited for simulation. In this case we extended the semantic model class `Tile` (see Fig. 1). We can introduce new attributes that are needed for simulation purposes only or extend our simulation model with so called path expressions to traverse the simulation model tree at run time. Both model the state space for the simulation.

We could also narrow the semantic model of the visual language in our simulation model. This can be done if parts of the semantic model of the visual language are only needed for representation purposes and not for simulation.

```
FOREACH ghost IN [Ghost] {
  IF(ghost.strategy == VLInt(1)) {
    Tile to = NEIGHBOUR_TILE(mapping, NEUMANN, ghost.tile, Pacman);
    IF(NOTNULL(to) AND (ghost.eatable == VLBoolean(0))){
      FIRE eatPacman(ghost.tile, to) @TIME_DIRECT;
    }
    ELSE {
      IF(NOTNULL(#[0]Pacman)) {
        Tile to = NEIGHBOUR_EMPTY_TILE_RANDOM(mapping, NEUMANN, ghost.tile);
        IF(NOTNULL(to)) {
          FIRE goGhost(ghost.tile, to) @TIME_DIRECT;
        }
      }
    }
  }
}
```

Fig. 3. Part of the simulation loop.

Fig. 3 shows an excerpt of the behavior specification part of DSIM. Here the simulation model can be inspected and events can be scheduled that modify an instance of the simulation model. Hence we follow the event based approach to simulation. Events are scheduled for an arbitrary time. Any event can trigger arbitrary so called simulation modification actions. These actions modify the simulation model and they constitute the interface to the animation framework. The excerpt shows the behavior specification of the ghost pawns. They try to eat Pac-man if it is located on a neighbour tile. If not, the ghost moves according to its strategy to the next tile.

In DSIM the following simulation modification actions exist which also form the interface to the animation part:

- `REMOVE` a structure object.
- `INSERT` a new structure object instance or insert a structure object, that was removed before. The latter would yield a `MOVE` action.
- `COPY` a structure object.
- `CHANGE_VAL` to change a primitive attribute or a reference.

In Fig. 2 (b) some events with corresponding simulation modification actions can be seen. The specific characteristics of the simulation modification actions is that an animation can be automatically derived from such a specification.

The default animations triggered are: *slow shrinking* to invisibility of an object that is removed, *slow growing* of an object that is newly inserted. *Linear moving* (with optional easing) of a structurally moved object. Copied objects move from their copy source to their destination while changing their transparency value from invisible to visible. Since editors generated by DEViL are syntax-directed structure editors, the creation or removal of structure objects can have side effects to other structure objects with respect to their size or position. These objects are automatically adapted smoothly, in that they are *morphed*. Even colors of structure objects are adapted smoothly.

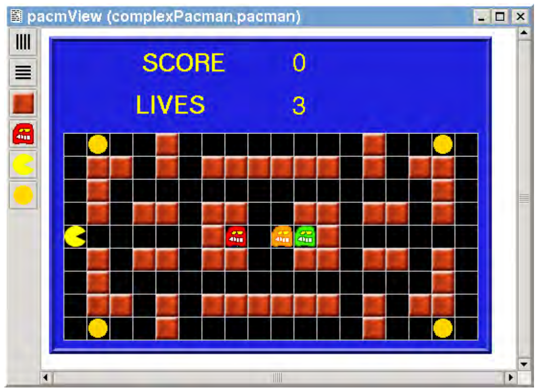
The default animation behaviour is sufficient for most automatically derived animations as can be seen later. But, in some cases the default animation that is automatically triggered is not what the animator of a visual language desires. Here the animator can override the default behavior with so called animated visual patterns (AVPs). The AVPs can be decorated like the visual patterns to a structure object and tell the structure object in what way it is animated if a certain simulation modification action occurs. For instance if a token in a Petri-net is removed it should not shrink to invisibility which is the standard animation. The desired animation is to move the token to the fired transition, hence the used AVP to override the default behavior for remove is *AVP*`OnRemove`*Move*. We have AVPs for changing size and transparency values of structure objects, for moving, scaling, rotating and so on. All of them can be combined and adapted to the needs of the animation.

A more detailed description of DEViL's simulation and animation facilities can be found in [2].

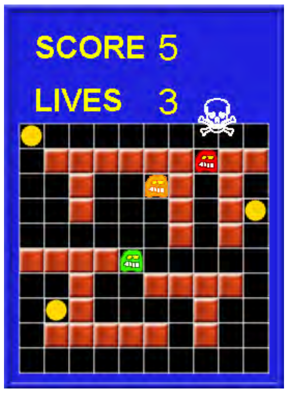
3. Pac-man

Pac-man is the most popular arcade computer game in the eighties of the last century and it was originally developed by Toru Iwatani for the Namco company in 1980. Because of the large degree of esteem different versions of the game have been reprogrammed many times for several systems like home computers, game-consoles and recently even for the iPhone [8]. The game is very interesting in terms of navigating a character around a structured playground, accumulating points, avoiding and (in some cases) attacking non-player game characters.

The classic version of Pac-man is an one-player game where the human player routes the Pac-man around a maze with the goal to avoid the four *ghost* characters and to eat as much pills as possible. Initially the pills are placed in each walk-in field of the playground and will be eaten via the achievement of the field by Pac-man. The overall four ghosts roam through the maze trying to catch Pac-man. This is successful when a ghost achieves a tile in which Pac-man is located. In this case Pac-man looses one of his initial three lives and the game restarts when Pac-man has just one life. Each of the four ghosts pursues a different strategy to eat the Pac-man.



(a) Starting position of a Pac-man game.



(b) A dynamic animation object.

Fig. 4. Screenshots of our Pac-man game.

Besides the normal pills in each tile there are four *powerpills* which are located near each corner of a maze. When Pac-man eats a powerpill he gets a special score and is able to eat ghosts on his part. In this case all ghosts change their color to blue for few moments, reverse their direction, and usually move more slowly. If Pac-man eats a ghost, he gets a special score and the ghost resurrects in the middle of the maze after a few moments. In addition to the previously seen options there is one more possibility to increment the score: sometimes a symbol of a fruit appears at a random position of the maze, which also gives the chance to get extra points.

The game ends when all pills have been eaten or Pac-man has lost all of his lives. In the former case the player reaches a new level which is more difficult than the previous one. This can be achieved for example by faster moving ghosts.

Fig. 4 (a) shows a playground of a Pac-man game, which has been build with our Pac-man Editor. Besides the Pac-man the figure shows three different ghosts, wall items and powerpills.

4. Pac-man Editor

Our Pac-man Editor is structured as a *multi document interface* (MDI) and offers the ability to create user-defined playgrounds for Pac-man games. The user has the option to insert different items to the playground, e.g. Pac-man, ghosts, powerpills or wall items. It is also possible to expand the playground by adding rows and columns. A playground which is constructed in such a way allows to play Pac-man as mentioned above.

The specification of the semantic model was the first task to implement this editor. The most important part of the semantic model is the matrix structure. An object of the matrix class is associated with an arbitrary number of columns and rows. Each row owns several tiles, which includes in turn an item or not. The item is an abstract class and the concrete subclasses are either Wall-item, Pac-man, Ghost or Powerpill.

To realize a correct semantic playground it is essential to avoid more than one Pac-man or a game without Pac-man. Hence, the DEViL System provides the ability to specify consistency constraints on various levels. E.g. cardinalities in the semantic model or specialized callback functions which can navigate the structure tree. All these checks are automatically performed before simulation. Hence, only a correct Pac-man game instance can be simulated.

Besides the consistency constraints the language designer can implement initialization functions for each class of the semantic structure. Such a function is a callback function and will be automatically called by the system, if a new object has been created. We used this, for example, to realize a default playground dimension of 10×10 tiles.

4.1. Strategies

With our editor it is possible to allot one of overall three different strategies to each ghost. We draw our inspiration with respect to the strategies of Repenning [12]:

Random The ghost roams randomly through the maze. At each step it evaluates the walk-in fields in the *von Neumann neighbourhood* [17] and chooses one randomly.

Incremental Approach The ghost tries to move closer to the Pac-man. At each step it evaluates the empty neighbour tiles and selects the closest one in euclidean sense.

Hill-climbing Due to the fact that the incremental approach does not permit the overcoming of walls, the strategy of hill-climbing affords this. To achieve this goal, diffusion values are used for each tile. These are used to spread the ”scent” of the Pac-man in the maze. The value represents the closeness of a ghost to Pac-man. The largest value gets the tile in which Pac-man is allocated. Starting from this tile, the value is distributed to all walk-in fields of the playground. Every tile which is not accessible, e.g. a tile with a wall item, gets a negative diffusion value. At each step of the game the ghost selects the tile which has the largest diffusion value. Due to the fact that the diffusion values must be recalculated at each step, this brings the ghost closer to Pac-man. Fig. 5 illustrates the allocation of diffusion values and the way a ghost must go to get Pac-man.

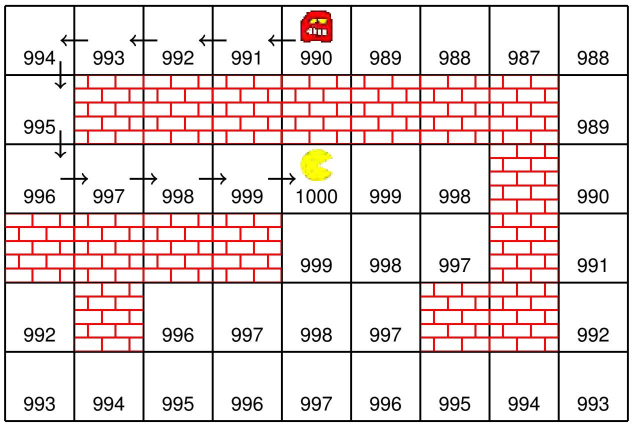


Fig. 5. Distribution of diffusion values to apply hill-climbing.

A closer look to the implementation of the hill-climbing strategy is available in the next section. Amongst other things we describe the implementation of ghost strategies in DSIM.

4.2. Simulation

The user interaction via keyboard is essential for the Pac-man game. The DEViL System provides the ability to define arbitrary keyboard events which can be processed in the simulation.

We used the simulation model to add particular attributes which are necessarily needed for the simulation. An extract is given in Fig. 2 (a). We extend the Pac-man class with a tile attribute, which allows the access of the tile in which Pac-man is located, from the context of a Pac-man object. Besides others, we had extended the tile class with an attribute which stores the diffusion value of a tile. This is needed to realize the hill-climbing strategy. Keep in mind, that these attributes only exist in the simulation model, not in the semantic model of the Pac-man VL.

In the event block we specified events, which can be scheduled at an arbitrary simulation time in the loop block. Hence, our simulator follows an event driven approach. We had implemented overall 16 different events. Fig. 6 shows two events. The `coordinatePacman` event gets the Pac-man instance and a direction to move to. It checks, whether a powerpill or a ghost is in the way. If so Pac-man tries to eat the ghost resp. the powerpill. If there is nothing to eat Pac-man just walks to the next tile, the `goPacman` event is called. This event again calls two events to increment the score and to compute the rotation, which is needed for the animation. Finally the Pac-man pawn is removed from the actual tile and inserted to the tile in the desired direction. This yields a `MOVE` action for the Pac-man pawn.

```
coordinatePacman(Pacman pacman, VLInt direction){
  Tile go = NEIGHBOUR_TILE(default, NEUMANN, pacman.tile, PowerPill);
  IF(NOTNULL(go)){
    FIRE eatPowerpill(pacman.tile, go, pacman, direction) @TIME_DIRECT;
  } ELSE {
    go = NEIGHBOUR_TILE(default, NEUMANN, pacman.tile, Ghost);
    IF(NOTNULL(go)){
      FIRE eatGhost(pacman.tile, go, pacman, direction) @TIME_DIRECT;
    }ELSE{
      go = NEIGHBOUR_TILE(default, pacman.tile, direction);
      IF(NOTNULL(go) AND (SIZE(go.item) == VLInt(0))){
        FIRE goPacman(pacman.tile, go, pacman, direction) @TIME_DIRECT;
      }
    }
  }
}

goPacman(Tile from, Tile to, Pacman p, VLInt d){
  FIRE incrementScore(#[0]Score, 1) @ TIME_DIRECT;
  FIRE computeRotation(p,d) @ TIME_DIRECT;
  Item i = REMOVE(from.item, FIRST);
  INSERT(to.item, i, FIRST);
}
```

Fig. 6. Coordination of the Pac-man pawn.

In each simulation step we have to compute the diffusion value for the hill-climbing strategy. This is done by a call of a C function. The function computes the value via a simple breadth-first search. Afterwards the ghost has to pick the target tile which has the largest diffusion value. To get a specific neighbour, we extended the simulation language such that we can access structure objects (of a specific type) in the neighbourhood of a given tile. All editors generated with DEViL that make use of tiling have the same underlying model. Due to this fact we could identify a subset of tile-access functions which are often needed and generalize these functions. This lead to a decrease of hand written C-code.

Fig. 7 shows some neighbour access functions. The first function counts the ghosts in *Moore neighbourhood* [16] of the Pac-man. A computation of the neighbour tile in south direction of a ghost shows the second function. The last function returns a random tile in *von Neumann* neighbourhood of Pac-man.

```
NEIGHBOUR_COUNT(mapping, MOORE, pacman.tile, Ghost);
NEIGHBOUR_TILE(mapping, ghost.tile, S);
NEIGHBOUR_TILE_RANDOM(mapping, NEUMANN, pacman.tile);
```

Fig. 7. Exemplary neighbour access functions.

4.3. Animation

The default animation which is automatically derived from the simulation specification is almost adequate. A ghost and the Pac-man move fast from the start tile to the target tile in each simulation step. This is the case, because the animation framework interprets the modification actions `REMOVE` and `INSERT` as a *moving* animation. Furthermore the Pac-man shrinks to invisibility when he is caught by a ghost.

But Pac-man looks in the desired viewing direction until he has accessed the target tile. It would be nicer if Pac-man rotates to the desired viewing direction in the start tile before he moves to the target tile. Now the idea is to override the default behaviour for Pac-man. All animations are typed over their simulation modification action. Hence, we need to override the default animation pattern `MOVE`, because the Pac-man is moved (`REMOVE`d and `INSERT`ed) on the playground. We do it with the specification in Fig. 8 (a). We use the animated visual patterns `OnMoveRotate` and `OnMoveMove`. `OnMoveRotate` rotates a structure object if it is moved. Hence, we have to override the angle and rotate attributes. The angle and rotate attributes are stored in the ”pacman” class (see Fig. 1) and will be computed via the *computeRotation* event in each simulation step. In addition we override the duration attribute to specify the duration of the rotate operation. In this configuration the rotation and the move are scheduled at the same simulation time, but we want the animation of the rotation to appear before the animation of the move. Hence the `OnMoveMove` animation must be animated after the `OnMoveRotate` animation. Hence, we have to assign the value 2 to the time attribute. Furthermore we override the duration attribute to indicate the duration time for a move operation. As can be seen, besides the simulation time, we have an animation time which defines an order of the animations and which can easily be adapted to gain a desired animation.

The animation framework offers the possibility to animate objects which are not part of the semantic model (so called *dynamic objects*). If Pac-man is caught, we have used this feature to display a skull (see Fig. 4 (b)). For such a purpose it is only necessary to use the provided visual patterns as described in Fig. 8 (b). The visual pattern `CreateDynamicObject` reacts to a modification action and offers the possibility to add a drawing. As seen in Fig. 8 (b) we override the modification action attribute to react to a remove action. Furthermore, we override the drawing attribute to add the skull drawing. In order that the skull moves bottom-up from the position of the Pac-man, we had used the pattern `MoveDynamicObject`. We also had used the pattern `OnRemoveShrink` to show the skull temporary.

The specification of an animation in DEViL is straight forward: first specify a simulation, then derive the animation automatically. Hence the animation is a formal mapping of its simulation part. At last, animations can be adapted by overriding the default animations through the application of a huge declarative animation pattern library.

```
SYMBOL pacmView_Pacman INHERITS VPContainerElement, VPForm,
       AVPOnMoveRotate, AVPOnMoveMove, AVPOnRemoveShrink
COMPUTE
  SYNT.drawing = ADDROF(PacmanDrawing);
  SYNT.onMoveRotateAngle = THIS.pers_angle;
  SYNT.onMoveRotateClockwise = THIS.pers_clockwise;
  SYNT.onMoveRotateDuration = 600;
  SYNT.onMoveMoveRaiseDisplayOrder = 1;
  SYNT.onMoveMoveAnimationTime = 2;
  SYNT.onMoveMoveDuration = 900;
  SYNT.onRemoveShrinkAnimationTime = 10;
END;
```

(a) Mapping of AVPs with control attributes to override default animation.

```
SYMBOL pacmView_Pacman INHERITS AVPCreateDynamicObject,
     AVPMoveDynamicObject
COMPUTE
  SYNT.createDynamicObjectModificationAction = REMOVE;
  SYNT.createDynamicObjectDrawing = ADDROF(SkullDrawing);
  SYNT.createDynamicObjectPosition = POSITION(
          SELECT(THIS.oPosition, sub(VLPoint(0,10))));

  SYNT.moveDynamicObjectDuration = 8000;
  SYNT.moveDynamicObjectStartPosition = POSITION(
          SELECT(THIS.oPosition, sub(VLPoint(0,10))));
  SYNT.moveDynamicObjectEndPosition = POSITION(
          SELECT(THIS.oPosition, sub(VLPoint(0,60))));
END;
```

(b) Creating a dynamic animation object and moving it.

Fig. 8. Animation of Pac-man.

5. Related Work

The Agentsheets system [11] can generate tile based simulations and games. The specification process is fully graphical and rule based. Agentsheets uses the programming by demonstration paradigm. In the rules one can access neighbour tiles through the help of icons with specific arrows. This is the visual variant of our neighbour access functions. Agentsheets is restricted to tile based simulation whereas our system can also handle diagrammatic or iconic visualizations.

In the area of generator frameworks for visual language environments the GenGed [1] system makes use of graph transformation and visual rewrite rules to specify simulation. To store the simulation state, rules must be extended. This is similar to our simulation model which can extend the semantic model of a VL. GenGed uses a formal mapping between simulation and animation. This is comparable to our simulation modification actions which trigger default animations. They are the interface between simulation and animation framework. Smooth and complex animations can not be specified with GenGed.

The DiaGen [6] system also uses graph transformation to specify a visual language. Some editors already support simulation and animation. Interesting is that every animation step is a state of the underlying graph transformation system whereas we interpolate between two adjacent simulation model states.

6. Conclusion

The specification of the Pac-man editor is a straight forward task. Table 1 shows that we needed 220 LOC for the whole simulation part including all ghost strategies and 400 LOC for hand written C-code. The other specs part needs only 236 LOC. The second column of the table shows a decrease of total LOC from 883 to 646 LOC. This is because of the neighbour access functions we had generalized. This reduced the LOC of C-code nearly by 250 LOC and we need only 7 additional LOC in the simulation specification to realize the mapping between concrete and generalized matrix structure. The 156 LOC of C-code is just a simple tile initialization function for the hill-climbing strategy. The automatically derived animation is sufficient to play the game. The 27 LOC are just syntactic sugar.

Table 1. Distribution of the specification complexity.

	LOC	**LOC with access fct.**	**generated LOC**
simulation	220	227	87.504
animation	27	27	
C-code	400	156	
other specs.	236	236	
	883	646	87.504

The DSIM language with its narrow interface to the animation and its constructs tailored for the simulation of visual languages has already been approved in other VLs with execution semantics like Petri-nets, a Datapath simulation, electronic circuits and even the game Ludo. Table 2 shows the amount of LOC for simulation and animation part of already implemented editors. The examples in line two and three are based on the Petri-net simulation shown in line one. They show the simulation of the well-known dining philosophers and a simulation of a signal light of a four-way-crossing. Both are structurally coupled to the Petri-net with DEViL's internal declarative coupling mechanism. A simulation of the Petri-net automatically triggers synchronization functions in the philosophers resp. signal-light view. The simulator detects these triggerings and calls the animation framework. Hence, additional specification amount is not needed.

As can be seen in Table 2 the automatically triggered animation is mostly sufficient. We need to adapt the animation only in simulations where animations

Table 2. Simulation and Animation LOC of other VLs.

	Simulation	**Coupling**	**Animation**	**Anim. syntactic sugar**
Petri-nets	29		4	0
Dining-Philosophers	29	95	4	0
Signal-Lights	29	57	4	0
Logo	211		3	3
Game of Life	39		0	0
Ludo	338		0	0
Statecharts	78		2	0
Bubblesort	13		0	0
Quicksort	93		0	0
Heapsort	225		0	0
CPU Datapath	263		160	0
Washing bay	35		0	0
Electronic circuits	99		109	0
DTM	96		9	9
Monitor	162		1	0
Sokoban	157		10	0
LR(1)-Parser	199		0	0
Traffic simulation	3372		26	6

depend on the context of their structure objects. E.g. this is the case in our CPU datapath simulation where an animation of an instruction is different whether it is located in an instruction decoder, in an accumulator or somewhere else.

The already implemented VLs have a very diverse appearance: we have diagrammatic, iconic and graph based depictions.

We just finished a visual language for traffic simulation. It is also a tile-bases VL and can simulate behaviour of cars following the well-known traffic simulation model of Nagel and Schreckenberg [7]. The language supports user defined rules, traffic analyses, the definition of routes and driver profiles. All in all we specified more than 3000 LOC for the simulation part. Hence the language is one of the most complex examples and it proves DEViL's ability to support large simulations and team work, because it was specified in a students project with 10 participants and a workload of one year.

We implemented about 19 different languages with simulation and animation support up to now. The overall simulation specification amount is 5500 LOC. For the animation we only needed about 350 LOC. Hence the standard animation is mostly sufficient and only needs few adaption.

Interesting extensions to our system would be semantic zooming, camera views or even isometric views. Here also a pattern based approach is imaginable.

Also outstanding is a visual language for DSIM and an usability study.

References

1. Bardohl, R.: GenGed: A generic graphical editor for visual languages based on algebraic graph grammars. In: 1998 IEEE Symp. on Visual Lang. pp. 48–55 (Sep 1998)
2. Cramer, B., Kastens, U.: Animation automatically generated from simulation specifications. In: VLHCC '09: Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 157–164. IEEE Computer Society, Washington, DC, USA (2009)
3. Cramer, B., Klassen, D., Kastens, U.: Entwicklung und Evaluierung einer domänenspezifischen Sprache für SPS-Schrittketten. In: Fahland, D., Sadilek, D.A., Scheidgen, M., Weileder, S. (eds.) DSML. CEUR Workshop Proceedings, vol. 324, pp. 59–73. CEUR-WS.org (2008), `http://dblp.uni-trier.de/db/conf/dsml/dsml2008.html#CramerKK08`
4. Kastens, U.: An attribute grammar system in a compiler construction environment. In: Proceedings of the International Summer School on Attribute Grammars, Application and Systems. Lecture Notes in Computer Science, vol. 545, pp. 380–400. Springer Verlag (1991)
5. Kastens, U., Pfahler, P., Jung, M.: The Eli system. In: Koskimies, K. (ed.) Proceedings of 7th International Conference on Compiler Construction CC'98. pp. 294–297. No. 1383 in Lecture Notes in Computer Science, Springer Verlag (Mar 1998)
6. Minas, M.: Concepts and realization of a diagram editor generator based on hypergraph transformation. Science of Computer Programming 44(2), 157–180 (Aug 2002), `http://www.elsevier.com/gej-ng/10/39/21/86/49/29/abstract.html`
7. Nagel, K., Schreckenberg, M.: A cellular automaton model for freeway traffic. Journal de Physique I 2, 2221–2229 (Dec 1992)
8. Namco Games: Pacman for iPhone. http://www.appsafari.com/games/2741/pacman-for-iphone/ (2008), [Online; accessed 16-December-2010]
9. Norman, D.A., Draper, S.W.: User Centered System Design; New Perspectives on Human-Computer Interaction. L. Erlbaum Associates Inc., Hillsdale, NJ, USA (1986)
10. Object Management Group: Unified Modeling Language (UML), version 2.2 (2009), http://www.omg.org/technology/documents/formal/uml.htm
11. Repenning, A.: AgentSheets®: an Interactive Simulation Environment with End-User Programmable Agents. In: Interaction 2000, Tokyo, Japan (2000)
12. Repenning, A.: Collaborative Diffusion: Programming Antiobjects. In: OOPSLA 2006, ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications, (Portland, Oregon, 2006). IEEE Press (2006)
13. Schmidt, C., Cramer, B., Kastens, U.: Usability evaluation of a system for implementation of visual languages. In: Symposium on Visual Languages and Human-Centric Computing. pp. 231–238. IEEE Computer Society Press, Coeur d'Alne, Idaho, USA (Sep 2007)

14. Schmidt, C., Kastens, U., Cramer, B.: Using DEViL for implementation of domain-specific visual languages. In: Proceedings of the 1st Workshop on Domain-Specific Program Development. Nantes, France (Jul 2006), `http://ag-kastens.upb.de/paper/dspd2006-devil.pdf`
15. Schmidt, C., Pfahler, P., Kastens, U., Fischer, C., Gmbh, O.K.: Simtelligence designer/j: A visual language to specify sim toolkit applications. In: Proceedings of the Second Workshop on Domain Specific Visual Languages (OOPSLA 2002 (2002), `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.2.9269`
16. Wikipedia, The Free Encyclopedia: Moore neighborhood. http://en.wikipedia.org/wiki/Moore_neighborhood (2010), [Online; accessed 16-December-2010]
17. Wikipedia, The Free Encyclopedia: Von Neumann neighborhood. http://en.wikipedia.org/wiki/Von_Neumann_neighborhood (2010), [Online; accessed 16-December-2010]

Jan Wolter received his bachelor degree in Computer Science from the University of Paderborn, Germany in 2009. He spent one year student assistant in the area of visual languages. Currently he is working on his master thesis concerning a concept to implement visual three-dimensional languages.

Bastian Cramer received his Ph.D. in Computer Science from the University of Paderborn in 2010. He joined the research group ”Programming Languages and Compilers” of Prof. Kastens at the same university. His research focus is the generation of software from specifications and especially the generation of environments for visual domain specific languages. He has several years of experience in language design in corporation with the automotive industry. In his Ph.D. thesis he evaluated the possibilities of simulation and animation of visual languages.

Uwe Kastens graduated as a ”Diplom-Informatiker” in 1972 at the University of Karlsruhe, Germany. In 1976 he received his doctorate in Computer Science from that University. Since 1982 he has been a Professor of Practical Computer Science at the University of Paderborn, Germany. His major research areas are methods and tools for language implementation, domain-specific languages, and program analysis. Since 1988 he has been a member of the IFIP Working Group 2.4 (System Implementation Languages, Software Implementation Technology) and was its chairman from 1991 to 1996.

Received: December 20, 2010; Accepted: February 16, 2011.

DOI: 10.2298/CSIS101116008R

Solving Difficult LR Parsing Conflicts by Postponing Them

C. Rodriguez-Leon[1] and L. Garcia-Forte[1]

Departamento de EIO y Computación,
Universidad de La Laguna
casiano@ull.es, lgforte@ull.es,
http://nereida.deioc.ull.es

Abstract. Though yacc-like LR parser generators provide ways to solve shift-reduce conflicts using token precedences, no mechanisms are provided for the resolution of difficult shift-reduce or reduce-reduce conflicts. To solve this kind of conflicts the language designer has to modify the grammar. All the solutions for dealing with these difficult conflicts branch at each alternative, leading to the exploration of the whole search tree. These strategies differ in the way the tree is explored: GLR, Backtracking LR, Backtracking LR with priorities, etc. This paper explores an entirely different path: to extend the yacc conflict resolution sublanguage with new constructs allowing the programmers to explicit the way the conflict must be solved. These extensions supply ways to resolve any kind of conflicts, including those that can not be solved using static precedences. The method makes also feasible the parsing of grammars whose ambiguity must be solved in terms of the semantic context. Besides, it brings to LR-parsing a common LL-parsing feature: the advantage of providing full control over the specific trees the user wants to build.

Keywords: parsing, lexical analysis, syntactic analysis.

1. Introduction

Yacc-like LR parser generators [3] provide ways to solve shift-reduce conflicts based on token precedence. No mechanisms are provided for the resolution of difficult reduce-reduce or shift-reduce conflicts. To solve such kind of conflicts the language designer has to modify the grammar. Quoting Merrill [5]:

> *Yacc lacks support for resolving ambiguities in the language for which it is attempting to generate a parser. It does a simple-minded approach to resolving shift/reduce and reduce/reduce conflicts, but this is not of sufficient power to solve the really thorny problems encountered in a genuinely ambiguous language*

Some context-dependency ambiguities can be solved through the use of lexical tie-ins: a flag which is set by the semantic actions, whose purpose is to alter the way tokens are parsed [1, p. 106]. But it is not always possible or easy to resort to this kind of tricks to fix some context dependent ambiguity.

A more general solution is to extend LR parsers with the capacity to branch at any multivalued entry of the LR action table. For example, Bison [1], via the `%glr-parser directive` and Elkhound [4] provide implementations of the Generalized LR (GLR) algorithm [11]. In the GLR algorithm, when a conflicting transition is encountered, the parsing stack is forked into as many parallel parsing stacks as conflicting actions. The next input token is read and used to determine the next transitions for each of the top states. If some top state does not transit for the input token it means that path is invalid and that branch can be discarded. Though GLR has been successfully applied to the parsing of ambiguous languages, the handling of languages that are both context-dependent and ambiguous is more difficult [10, p. 3]. The Bison manual [1] points out the following caveats when using GLR:

> *. . . there are at least two potential problems to beware. First, always analyze the conflicts reported by Bison to make sure that GLR splitting is only done where it is intended. A GLR parser splitting inadvertently may cause problems less obvious than an LALR parser statically choosing the wrong alternative in a conflict. Second, consider interactions with the lexer with great care. Since a split parser consumes tokens without performing any actions during the split, the lexer cannot obtain information via parser actions. Some cases of lexer interactions can be eliminated by using GLR to shift the complications from the lexer to the parser. You must check the remaining cases for correctness.*

The strategy presented here extends yacc conflict resolution mechanisms with new ones, supplying ways to resolve conflicts that can not be solved using static precedences. The algorithm for the generation of the LR tables remains unchanged, but the programmer can modify the parsing tables during run time.

The technique involves labelling the points in conflict in the grammar specification and providing additional code to resolve the conflict when it arises. Crucially, this does not requires rewriting or transforming the grammar, trying to resolve the conflict in advance, backtracking or branching into concurrent speculative parsers. Instead, the resolution is postponed until the conflict actually arises during parsing, whereupon user code inspects the state of the underlying parse engine to decide the appropriate solution. There are two main benefits: Since the full power of the native universal hosting language is at disposal, any grammar ambiguity can be tackled. We can also expect - since the conflict handler is written by the programmer - a more efficient solution which reduces the required amount of backtracking or branching.

This technique can be combined to complement both GLR and backtracking LR algorithms [10] to give the programmer a finer control of the branching process. It puts the user - as it occurs in top down parsing - in control of the parsing strategy when the grammar is ambiguous, making it easier to deal with efficiency and context dependency issues. One disadvantage is that it requires some knowledge of LR parsing. It is conceived to be used when none of the available techniques - static precedences, grammar modification, backtracking LR or Generalized LR - produces satisfactory solutions. We have implemented these techniques in `eyapp` [7], a yacc-like LALR parser generator for Perl [13, 6].

This paper is divided in six sections. The next section introduces the Postponed Conflict Resolution (PPCR) strategy. The following three sections illustrate the way the technique is used. The first presents an ambiguous grammar where the disambiguating rule is made in terms of the previous context. The next shows the technique on a difficult grammar that has been previously used in the literature [1] to illustrate the advantages of the GLR engine: the declaration of enumerated and subrange types in Pascal [12]. The last example deals with a grammar that can not be parsed by any LL(k) nor LR(k), whatever the value of `k`, nor for packrat parsing algorithms [2]. The last section summarizes the advantages and disadvantages of our proposal.

2. The *Postponed Conflict Resolution* Strategy

The *Postponed Conflict Resolution* (PPCR) is a strategy to apply whenever there is a shift-reduce or reduce-reduce conflict which is unsolvable using static precedences. It delays the decision, whether to shift or reduce and by which production to reduce, to parsing time. Let us assume the `eyapp` compiler announces the presence of a reduce-reduce conflict. The steps followed to solve a reduce-reduce conflict using the PPCR strategy can be divided in two activities: conflict identification and mapping (steps 1a to 1d) and writing the solver (step 2a).

1. **Conflict Identification and Mapping**
   (a) Identify the conflict: What LR(0)-items/productions and tokens are involved?.
   Tools must support that stage, as for example via the `.output` file generated by `eyapp`. Suppose we identify that the participants are the two LR(0)-items $A \rightarrow \alpha_{\uparrow}$ and $B \rightarrow \beta_{\uparrow}$ when the lookahead token is `@`.
   (b) Give a name to the productions: the software must allow the use of symbolic labels to refer by name to the productions involved in the conflict. Let us assume that production $A \rightarrow \alpha$ has label `:rA` and production $B \rightarrow \beta$ has label `:rB`. A difference with `yacc` is that in `eyapp` productions can have *names* and *labels*. In `eyapp` names and labels can be explicitly given using the directive `%name`, using the following syntax:

   `%name :rA` $A \rightarrow \alpha$
   `%name :rB` $B \rightarrow \beta$

   (c) Give a symbolic name to the conflict. In this case we choose `isAorB` as name of the conflict.

(d) Inside the *body* section of the grammar, mark the points of conflict using the new reserved word `%PREC` followed by the conflict name:

`%name :rA` $A \to \alpha$ `%PREC IsAorB`
`%name :rB` $B \to \beta$ `%PREC IsAorB`

2. **Writing the Conflict Handler**

(a) Introduce a `%conflict` directive inside the *head* section of the translation scheme to specify the way the conflict will be solved. The directive is followed by some code - known as the *conflict handler* - whose mission is to modify the parsing tables. This code will be executed each time the associated conflict state is reached. This is the usual layout of the conflict handler:

```
%conflict  IsAorB {
  if (is_A) { $self->YYSetReduce('@', ':rA' ); }
       else { $self->YYSetReduce('@', ':rB' ); }
}
```

The call to `is_A` represents the context-dependent dynamic knowledge that allows us to take the right decision. It is usually a call to a nested parser for A but it can also be any other contextual information we have to determine which one is the right production.
Inside a conflict handler the Perl default variable `$_` refers to the full input text and `$self` refers to the parser object.
Variables in Perl - like `$self` - have prefixes like `$` (scalars), `@` (lists), `%` (hashes or dictionaries), `&` (subroutines), etc. specifying the type of the variable. These prefixes are called *sigils*. The sigil `$` indicates a *scalar* variable, i.e. a variable that stores a single value: a number, a string or a reference. In this case `$self` is a reference to the parser object. The arrow syntax `$object->method()` is used to call a method: it is the equivalent of the dot operator `object.method()` used in most OOP languages. Thus the call

```
$self->YYSetReduce('@', ':rA' )
```

is a call to the `YYSetReduce` method of the object `$self`.
The method `YYSetReduce` provided by `Parse::Eyapp` receives a token, like `'@'`, and a production label, like `:rA`. The call

```
$self->YYSetReduce('@', ':rA' )
```

sets the parsing action for the state associated with the conflict `IsAorB` to reduce by the production `:rA` when the current lookahead is `@`. The token argument `'@'` is optional. If omitted, the set of conflictive tokens will be used.

The procedure is similar for shift-reduce conflicts. Let us assume we have identified a shift-reduce conflict between LR-(0) items $A \to \alpha_{\uparrow}$ and $B \to \beta_{\uparrow}\gamma$ for some token `'@'`. Only steps 1d and 2a change slightly:

1d'. Again, we must give a symbolic name to $A \to \alpha$ and mark with the new `%PREC` directive the places where the conflict occurs:

```
%name :rA  $A \to \alpha$  %PREC IsAorB
           $B \to \beta$  %PREC IsAorB  $\gamma$
```

2a'. Now the conflict handler calls the `YYSetShift` method to set the `shift` action:

```
%conflict  IsAorB {
  if (is_A) { $self->YYSetReduce('@', ':rA' ); }
  else { $self->YYSetShift('@'); }
}
```

The token argument `'@'` of `YYSetShift` is optional. If omitted, the set of conflictive tokens is used instead.

In order to clarify the use of PPCR we will address three different kind of conflicts:

– A simple case of dynamically changing the associativity to show the use of the `%conflict` directive (section 3)
– The classical subrange/enum Pascal conflict [1, p. 21] presented in section 4 depicts the use of preparsing, the `%explore` directive and some details of the `eyapp` compiler
– The parsing of a non LR(k) unambiguous grammar (section 5)

3. A Simple Example

The following example[1] accepts lists of two kind of commands: *arithmetic expressions* like `4-2-1` or one of two *associativity commands*: `left` or `right`. When a `right` command is issued, the semantic of the `'-'` operator is changed to be right associative. When a `left` command is issued the semantic for `'-'` returns to its classic left associative interpretation. Here follows an example of input. Between shell-like comments appears the expected output:

[1] For the full examples used in this paper, see [8]

```
$ cat input_for_dynamicgrammar.txt
2-1-1 # left:  0 = (2-1)-1
RIGHT
2-1-1 # right: 2 = 2-(1-1)
LEFT
3-1-1 # left:  1 = (3-1)-1
RIGHT
3-1-1 # right: 3 = 3-(1-1)
```

We use a variable `$reduce` (initially set to `1`) to decide the way in which the ambiguity `NUM-NUM-NUM` is solved. If `false` we will set the `NUM-(NUM-NUM)` interpretation. The variable `$reduce` is modified each time the input program emits a `LEFT` or `RIGHT` command.

Following the steps outlined above, and after looking at the `.output` file, we see that the items involved in the announced shift-reduce conflict are

$$expr \to expr_{\uparrow} - expr$$
$$expr \to expr - expr_{\uparrow}$$

and the lookahead token is `'-'`. We next mark the points in conflict in the grammar using the `%PREC` directive (see Figure 1)

```
%%
p:
    /* empty */   {}
  | p c           {}
;

c:
    $expr { print "$expr\n"}
  | RIGHT { $reduce = 0}
  | LEFT  { $reduce = 1}
;
```

```
expr:
     '(' $expr ')'  { $expr }
   | %name :M
     expr.left        %PREC LoR
       '-' expr.right %PREC LoR
        { $left - $right }

   | NUM
;
```

Fig. 1. An Example of Context Dependent Ambiguity Resolution

The *dollar* and *dot* notation used in some right hand sides (rhs) like in `expr.left '-' expr.right` and `$expr` is used to associate variable names with the attributes of the symbols.

The conflict handler `LoR` defined in the header section is:

```
%conflict LoR {
  if ($reduce) {$self->YYSetReduce(':M')}
  else         {$self->YYSetShift()}
}
```

If `$reduce` is `true` we set the parsing action to *reduce* by the production labelled `:M`, otherwise we choose the *shift action*.

Observe how PPCR allow us to dynamically change at will the meaning of the same statement.

4. Nested Parsing

This section illustrates the technique through a problem that arises in the declaration of enumerated and subrange types in the programming language Pascal. The problem is taken from the Bison manual, (see section '*Using GLR on Unambiguous Grammars*' [1, p. 21]) where it is used as a paradigmatic example of when to switch to the GLR engine [1]. Here are some cases:

```
type subrange = lo .. hi;
type enum = (a, b, c);
```

The original language standard allows only numeric literals and constant identifiers for the subrange bounds (`lo` and `hi`), but Extended Pascal (ISO/IEC 10206) [12] and many other Pascal implementations allow arbitrary expressions there. This gives rise to declarations like the following:

`type subrange = (a) .. b;`	`type enum = (a);`

The corresponding declarations look identical until the '`..`' token. With normal LALR(1) one-token lookahead it is not possible to decide between the two forms when the identifier '`a`' is parsed. It is, however, desirable for a parser to decide this, since in the latter case '`a`' must become a new identifier to represent the enumeration value, while in the former case '`a`' must be evaluated with its current meaning, which may be a constant or even a function call. The Bison manual considers and discards several potential solutions to the problem to conclude that the best approach is to declare the parser to use the GLR algorithm. To aggravate the conflict we have added the C *comma* operator inside `expr`, making room for the generation of declarations more difficult to parse as:

```
type subrange = (a, b, c) .. (d, e);
type enum = (a, b, c);
```

Here is our modification of the vastly simplified subgrammar of Pascal type declarations found in [1].

```
%token ID  = /([A-Za-z]\w*)/
%token NUM = /(\d+)/

%left   ','
%left   '-' '+'
%left   '*' '/'

%%

type_decl : 'TYPE' ID '=' type ';'
;

type :
      '(' id_list ')'
    | expr '..' expr
;
```

```

id_list :
      ID
    | id_list ',' ID
;

expr :
      '(' expr ')'
    |  expr '+' expr
    |  expr '-' expr
    |  expr '*' expr
    |  expr '/' expr
    |  expr ',' expr
    |  ID
    |  NUM
;
```

4.1. Identifying the problem

When used as a normal LALR(1) grammar, eyapp correctly complains about two reduce/reduce conflicts:

```
$ eyapp -v pascalenumeratedvsrange.eyp
2 reduce/reduce conflicts
```

The generated `.output` file tell us that both conflicts occur in state 11. It also give us the contents of state 11:

```
State 11:
    id_list -> ID . (Rule 4)
    expr -> ID .    (Rule 12)
    ')' [reduce using rule 12 (expr)]
    ')' reduce using rule 4 (id_list)
    ',' [reduce using rule 12 (expr)]
    ',' reduce using rule 4 (id_list)

    '*' reduce using rule 12 (expr)
    '+' reduce using rule 12 (expr)
    '-' reduce using rule 12 (expr)
    '/' reduce using rule 12 (expr)
```

From the inspection of state 11 we can conclude that the two reduce-reduce conflicts occur between productions `id_list -> ID` and `expr -> ID` in the presence of tokens '`)`' and '`,`'. To solve the conflict we label the two involved productions and set the `%PREC` directives:

```
id_list :
      %name ID:ENUM
      ID                    %PREC rangeORenum
    | id_list ',' ID
;
expr : '(' expr ')'
    | %name ID:RANGE
      ID                    %PREC rangeORenum
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | expr ',' expr
    | NUM
;
%%
```

4.2. Pre-parsing the Incoming Input

To find which production applies we will pre-parse the input at the point where a range or a enumerated type is expected. To achieve it, we introduce an auxiliary syntactic variable `Lookahead` marking the point where the nested parsing starts:

```
type_decl : 'type' ID '=' type ';' ;

type :
      %name ENUM
      Lookahead '(' id_list ')'
    | %name RANGE
      Lookahead expr '..' expr
;
```

The semantic action associated with `Lookahead` is to check if the incoming input matches a range:

```
Lookahead: /* empty */
  $is_range = $self->YYPreParse('range');
;
```

The call to the method `$_[0]->YYPreParse('range')` uses a `range` parser to parse the input from the current position. It returns true if the `range` parser finds a conformant substring starting at that point.

The `range` parser recognizes the language defined by the subgrammar:

```
range: expr '..' expr ';'
```

where the definition of `expr` is as in the previous grammar.

The conflict handler in the head section decides which production must be used in terms of the value of `$is_range`:

```
  %{
    my $is_range = 0;
  %}
  %conflict rangeORenum {
   if ($is_range)
    { $self->YYSetReduce('ID:RANGE'); }
   else
    { $self->YYSetReduce('ID:ENUM'); }
  }
```

The `eyapp` compiler uses these token definitions

```
%token ID  = /([A-Za-z]\w*)/
%token NUM = /(\d+)/
```

to automatically generate the lexical analyzer. The rest of the head section of the grammar set the classic static priorities for the arithmetic operators:

```
%left   ','
%left   '-' '+'
%left   '*' '/'
```

4.3. Compiling the Grammar

To produce the parser, we start compiling the auxiliary grammar:

```
$ eyapp -P range.eyp
```

The `-P` option used when compiling `range` tells `eyapp` to produce parsing tables that will accept if a prefix of the input belongs to the language generated by the `range` grammar[2]. We then proceed to compile the full grammar[3]:

```
$ eyapp -TC enumvsrange.eyp
```

The `eyapp` compiler provides a default `main` which will be used if no `main` is provided. The default `main` accepts several command line arguments:

```
$ ./enumvsrange.pm -t -i -m 1 -c 'type e = (x, y, z);'
```

[2] By default, the generated parser only accepts if the full input conforms to the grammar
[3] Option `-T` tells the compiler to insert semantic actions in order to produce the syntax tree. Option `-C` is used to generate an executable

Option `-t` tells the `main` to print the result returned by the parser: a description of the syntax tree will be printed. Options `-i` and `-m 1` control the way the syntax tree is shown. Option `-c` is followed by the input for the parser. It indicates that the input is given in the command line. The execution of the former command produces the following output:

```
typeDecl_is_type_ID_type(
  TERMINAL[e],
  ENUM(
    idList_is_idList_ID(
      idList_is_idList_ID(ID(TERMINAL[x]), TERMINAL[y]),
      TERMINAL[z])))
```

4.4. The `%explore` Directive

In the previous grammar we explicitly introduced a new syntactic variable `Lookahead` to set the point for nested parsing. The `eyapp` programmer can use the

```
          %explorer conflictName { CODE }
```

directive inside the head section to declare the code in charge of the nested parsing:

```
%explorer rangeORenum {
            $is_range = $_[0]->YYPreParse('range');
}
```

Then the point where the exploration starts is marked inside the grammar body using the `%conflictname?` syntax:

```
type :
      %name ENUM
      %rangeORenum? '(' id_list ')'
    | %name RANGE
      %rangeORenum? expr '..' expr
;
```

The `eyapp` compiler will mimic the technique outlined in the previous section, creating a new syntactic variable, let us call it `Lh`, whose only empty production has as associated semantic action the code defined in the `%explorer` directive:

```
Lh: /* empty */ { $is_range = $_[0]->YYPreParse('range') }
```

The points where the `%rangeORenum?` directive appears are substituted by that variable:

```
type :
      %name ENUM
      Lh '(' id_list ')'
    | %name RANGE
      Lh expr '..' expr
;
```

forcing the execution of the exploration code at that points.

5. Conflicts Requiring Unlimited Look-ahead

The following unambiguous grammar can not be parsed by any LL(k) nor LR(k), whatever the value of `k`, nor by packrat parsing algorithms [2].

```
%%
T: S          ;
S: x S x | x  ;
%%
```

Though it is straightforward to find equivalent LL(1) and LR(1) grammars (the language is even regular: `/x(xx)*/`), even GLR [11] and Backtrack LR parsers [5] for this grammar will suffer of a potentially exponential complexity in the input size. The unlimited number of look-aheads required to decide if the current `x` is in the middle of the sentence, leads to an increase in the number of branches to explore. To make the problem more difficult and more representative, let us assume `x` is not a token but defines the language of the arithmetic expressions.

The challenge is to make the parser work *without changing* the grammar. As in the previous example we start identifying the conflict - which we name `isInTheMiddle` -, labelling as `:MIDx` the reduction item and marking the exploration point:

```
%token NUM = /(\d+)/
%token OP  = /([-+*\/])/

%%
T: %isInTheMiddle? S ;

S:
     x  %PREC isInTheMiddle S x
  |  %name :MIDx
     x  %PREC isInTheMiddle
;

x:   NUM | x OP NUM
;
%%
```

The exploration code uses the auxiliary parser `ExpList` to compute the number of `x`s in the list. Variable `$nxr` is then used to store the mid position:

```
%explorer isInTheMiddle {
   ($nxr) = $self->YYPreParse('ExpList');
   $nxr = int ($nxr/2);
}
```

When `YYPreParse` is called in a list context like above - observe the parenthesis around `$nxr` - it returns the semantic value computed by `ExpList`. The `ExpList.eyp` grammar computes the number of `x`s:

```
%%
S: $S x  { $S + 1 } |  x  { 1 }  ;
%%
```

Where the definition of `x` is as in the previous grammar.

The conflict solver code is quite simple: it keeps the position of the current `x` inside the state/persistent variable `$nxs`. The reduction is called when the middle point is reached:

```
%conflict isInTheMiddle {
  state $nxs = 0;

  $nxs++;
  if ($nxs == $nxr+1) {
    $self->YYSetReduce(':MIDx' );
    $nxr = $nxs = 0;
  }
  else { $self->YYSetShift() }
}
```

6. Conclusions

The strategy presented in this paper extends the classic yacc precedence mechanisms with new dynamic conflict resolution mechanisms. These new mechanisms provide ways to resolve conflicts that can not be solved using static precedences. They also provides finer control over the conflict resolution process than other alternatives. There are no limitations to PPCR parsing, since the conflict handler is implemented in a universal language and it then can resort to any kind of nested parsing algorithm. The conflict resolution mechanisms presented here can be introduced in any LR parsing tools, since they are independent of the implementation language and the language used for the expression of the semantic actions. One disadvantage of PPCR is that it requires more effort than branching methods like GLR or backtracking LR. With some effort, the PPCR methodology can be extended to be merged with GLR and backtracking LR, allowing for a mixed exploration that uses both branching (GLR) and correct pruning (PPCR). This research line seems worth to explore.

LR conflict removal is a laborious task. The number of conflicts in a programming language can reach tens and even hundreds: The original grammars of Algol-60 and Scheme result in 61 and 78 conflicts respectively with an average density of one conflict for each two productions. By adding Postponed Conflict Resolution to the classical precedence and associativity settings we can fix the conflicts in such grammars without modifying the grammars. Removing conflicts while preserving the grammar is preferable to rewriting the grammar in several situations: When using a conflict removal tool like the one described in [9], since the language designer will be still familiar with the resulting grammar, when the original grammar better reflects the author ideas about the syntax and semantic of the language, when the original grammar is easier to read and to understand (size matters) and when such unambiguous grammar is hard or impossible to find. As future work, we intend to address the building of tools assisting the process of conflict identification and conflict removal without modifying the original grammar.

Acknowledgments. This work has been supported by the EC (FEDER) and the Spanish Ministry of Science and Innovation inside the 'Plan Nacional de I+D+i' with the contract number TIN2008-06491-C04-02. It has also been supported by the Canary Government project number PI2007/015.

References

1. Donnelly, C., Stallman, R.M.: Bison: the yacc-compatible parser generator. Technical report, Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139 (2010)
2. Ford, B.: Functional pearl: Packrat parsing: Simple, powerful, lazy, linear time. Massachusetts Institute of Technology. Cambridge, MA (2002)
3. Johnson, S.C.: Yacc: Yet another compiler compiler. AT&T Bell Laboratories Technical Report July 31, 1978 2, 353–387 (1979)
4. Mcpeak, S.: Elkhound: A fast, practical GLR parser generator (2004), [Online]. Available: `http://scottmcpeak.com/elkhound/`
5. Merrill, G.H.: Parsing Non-LR(k) Grammars with Yacc. Software, Practice and Experience 23(8), 829–850 (1993)
6. Randal, A., Sugalski, D., Totsch, L.: Perl 6 and Parrot Essentials. O'Reilly Media (2004)
7. Rodríguez-León, C.: Parse::Eyapp Manuals (2007), [Online]. Available at CPAN: `http://search.cpan.org/dist/Parse-Eyapp/`
8. Rodríguez-León, C., García-Forte, L.: Grammar Repository (2010), [Online]. Available at google-code: `http://code.google.com/p/grammar-repository/`
9. Teixeira Passos, L., Bigonha, M.A., Bigonha, R.: A methodology for removing LALR(k) conflicts. In: Journal of Universal Computer Science. pp. 735–752 (2007)
10. Thurston, A.D., Cordy, J.R.: A backtracking LR algorithm for parsing ambiguous context-dependent languages. In: 2006 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 2006). pp. 39–53. Toronto (2006)

11. Tomita, M.: The generalized LR parser/compiler - version 8.4. In: Proceedings of International Conference on Computational Linguistics (COLING'90). pp. 59–63. Helsinki, Finland (1990)
12. van der Veen, V.: Extended pascal iso 10206:1990, [Online]. Available: `http://www.standardpascal.org/iso10206.txt`
13. Wall, L., Christiansen, T., Schwartz, R.: *Programming Perl. O'Reilly & Associates (1996)*

C. Rodriguez-Leon is a full professor of Computer Science at Universidad de La Laguna, Spain. He received his Diploma in Mathematics and his Doctorate (Ph. D.) in 1978 and 1987, respectively, both from the same University. He is in the Editorial Board of journals like Parallel Computing, International Journal of Computational Science and Engineering (IJCSE), etc. and has been in the Program Committee of several parallel computing conferences including EuroPVM/MPI, HeteroPar, EuroPar, HLPP, CIT, ICA3PP, CACIC, INFORUM, WGISD, etc. His research interest includes Parallel Algorithms, Evolutionary Computation, Combinatorial Optimization, Distributed Computing, Language Processing and High Level Programming.

L. Garcia-Forte is a Ph. D. student of the Department of Statistics, Operation Research and Computation at Universidad de La Laguna, Spain. He received his Diploma in Computer Engineering in 1997. His main research interests focus on Programming Languages and Parallel Systems.

Received: November 16, 2010; Accepted: April 19, 2011.

DOI: 10.2298/CSIS110110007L

Detecting Concurrency Anomalies in Transactional Memory Programs

João Lourenço, Diogo Sousa, Bruno Teixeira and Ricardo Dias*

CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica, Portugal
{joao.lourenco, rjfd}@di.fct.unl.pt
{dm.sousa, bct18897}@fct.unl.pt

Abstract. Concurrent programs may suffer from concurrency anomalies that may lead to erroneous and unpredictable program behaviors. To ensure program correctness, these anomalies must be diagnosed and corrected. This paper addresses the detection of both low- and high-level anomalies in the Transactional Memory setting. We propose a static analysis procedure and a framework to address Transactional Memory anomalies. We start by dealing with the classic case of *low-level dataraces*, identifying concurrent accesses to shared memory cells that are not protected within the scope of a memory transaction. Then, we address the case of *high-level dataraces*, bringing the programmer's attention to pairs of memory transactions that were misspecified and should have been combined into a single transaction. Our framework was applied to a set of programs, collected form different sources, containing well known low- and high-level anomalies. The framework demonstrated to be accurate, confirming the effectiveness of using static analysis techniques to precisely identify concurrency anomalies in Transactional Memory programs.

Keywords: testing, verification, concurrency, software transactional memory, static analysis.

1. Introduction

Concurrent programming is inherently hard. The fact that more than one ordering of events may take place at runtime leads to an exponential growth in the number of both valid and invalid program states. Programs with concurrency errors may reach invalid states and expose unpredicted and anomalous behaviors. Thus, more than just convenient, tool based approaches tackling the automatic verification and validation of programs are essential building-blocks in the

* This work was partially supported by Sun Microsystems under the "Sun Worldwide Marketing Loaner Agreement #11497", by the Centro de Informática e Tecnologias da Informaçao (CITI), and by the Fundaçao para a Ciencia e Tecnologia (FCT/MCTES) in the research projects Byzantium (PTDC/EIA/74325/2006) and RepComp (PTDC/EIA-EIA/108963/2008), and research grant SFRH/BD/41765/2007.

process of developing correct concurrent programs. This paper addresses the identification of both low- and high-level dataraces in the Transactional Memory [12, 18, 20, 21] setting.

Data races are among the most notorious concurrency errors. A program suffers from a *low-level datarace*, or simply *datarace*, when two threads concurrently access a shared variable with no concurrency control enforced, and at least one of those accesses is an update. Low-level dataraces may be avoided by synchronizing the conflicting threads, e.g., using locks, thus enforcing that critical sections, program code blocks that are mutually exclusive, will not be executed concurrently.

A program free from low-level dataraces may still exhibit concurrency anomalies resulting from the a scope misspecification, where two or more correctly synchronized critical sections should be merged into a single one to ensure the program's correctness. We shall call these errors *high-level dataraces* or *high-level anomalies*. Likewise, low-level dataraces are also referred in this paper as *low-level anomalies*.

Transactional Memory (TM) [11,19] is a promising approach that offers multiple advantages for concurrent programming. In contrast to locks, which enforces mutual exclusion, TM is neutral concerning the execution model, resorting in a transactional monitor to establish the transactional properties at run-time. The transactional monitor may opt to enforce mutual exclusion, as with locks, or to allow transactions to execute concurrently, optimistically assuming they will not conflict, and later aborting and restarting those that do conflict.

TM is inherently immune to some of the concurrency anomalies that are common in lock-based programs, such as deadlocks. Data races are among the anomalies that can still be observed. A transaction is only shielded against another transaction, in the same way that a lock-protected critical section is only protected from another critical section which holds a common lock. Therefore, in the TM setting, non-transactional and transactional code may also compete when accessing shared variables, leading to low-level dataraces. Likewise high-level dataraces in lock-based programs, the misspecification of the scope of two or more memory transactions may lead to high-level dataraces in the TM setting.

There are several approaches for detecting low-level dataraces in lock-based programs, both static [6, 9, 14], dynamic [8, 13, 16], and hybrid [17]. Likewise, there are also some approaches for detecting high-level dataraces in lock-based programs [3, 5, 10, 22, 24]. As locks relate to transactions, these works on low- and high-level dataraces also relate to TM, but none of them targets specifically this setting, which is in the core of our approach.

In Section 2, we discuss a process that enables the usage of a low-level datarace detector meant for locks in a TM-based program. In Section 3 we propose a definition for high-level dataraces in the TM setting and address their detection using static analysis, by conservatively extracting all possible concurrent execution traces of a program and searching for anomalies using a pattern-

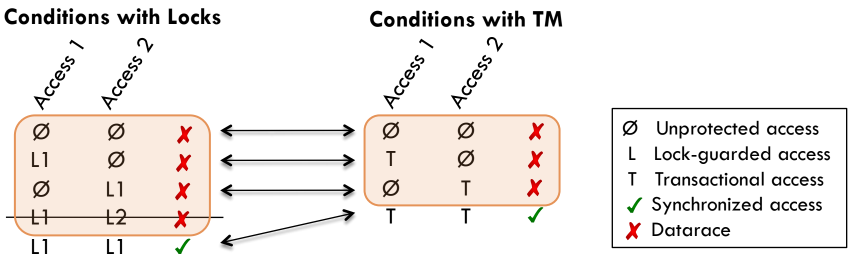


Fig. 1. Conditions for a datarace in Transactional and Locks.

based heuristic approach. We then discuss the related work in Section 4, followed by the conclusions and future work in the last section.

2. Low-Level Dataraces in Transactional Memory

Locks enforce mutual exclusion between critical sections. If two critical sections are protected by at least one common lock, then no two threads may execute them at the same time. Transactional memory, on the other hand, does not enforce mutual exclusion. Instead, two transactional code blocks may execute concurrently, provided by the TM run-time with the guarantees of *Isolation* and *Atomicity*. TM usually provides the *serializability* of transactions, ensuring that if two memory transactions take place concurrently and both succeed, then its final outcome is the same as if those two transactions were executed one after the other. Violations of serializability usually lead to dataraces.

Consider the distinct situations that may lead to a low-level datarace between two lock-based synchronized threads:

1. None of the accesses are performed while holding a lock;
2. One of the accesses is performed holding no locks; or
3. Both accesses are performed while holding disjoint sets of locks.

When using locks, the user chooses which critical sections shall be mutually exclusive by acquiring and holding the appropriate lock. In the TM setting, all transactions are guaranteed to be atomic and isolated from all other concurrent transactions, which excludes the third case above. Hence, as illustrated in Figure 1, the only two situations that may lead to low-level dataraces in TM are:

1. None of the accesses is performed in the scope of a transaction; or
2. Only one of the accesses is performed in the scope of a transaction.

Listing 1 illustrates cases of dataraces in both lock-based and transactional memory programs (left and right columns respectively). In the lock-based case, blocks A, B, and C, are assumed to execute concurrently. Likewise for the TM case, where blocks W, X, and Y, are also assumed to execute concurrently.

```
// A
synchronized (a) {
    a.x = 0;
}
...
// B
print(a.x);
...
// C
synchronized (this) {
    a.x++;
}
```

```
// W
atomic {
    a.x = 0;
}
...
// X
print(a.x);
...
// Y
atomic {
    a.x++;
}
```

Listing 1. Example of a low-level dataraces with locks (left) and with transactional memory (rigth)

The lock-based version has two different kinds of dataraces. Blocks A and B have a no-lock conflict, as block B is accessing $a.x$ without holding a lock. The same applies to blocks C and B. Blocks A and C have a wrong-lock conflict, as both are holding different locks and thus their concurrent accesses to $a.x$ are not protected. All the dataraces in the TM-based version are of a single type. Blocks W and X have a data race resulting from the execution of block X outside the scope of a transaction, and the same applies to blocks Y and X.

Our approach to identify low-level dataraces in TM programs resorts to their similarities and relations to the low-level dataraces in lock-based programs, interpreting the TM `atomic` blocks as if they were synchronized on a single global lock and then apply techniques and tools used in the detection of dataraces in lock-based programs. Hence, each of the scenarios of low-level datarace in TM maps into an analogous scenarios in a single lock setting, as denoted by the arrows in Figure 1, making this approach both sound and complete [21].

2.1. Detection Approach

Our approach to identify low-level dataraces in transactional memory programs is depicted in Figure 2.

The TM Java program is processed with AJEX [7], an extension to Polyglot [15], that recognizes the keyword `atomic` as a new Java construct to denote a transactional memory code block and generates the corresponding Abstract Syntax Tree (AST). The AST generates by AJEX is then traversed using the Polyglot framework and the transactional blocks are replaced with blocks synchronized on a single unused global lock. The definition of the new global lock is added to the main class, if one exists, otherwise to another arbitrary class. The identifier of the global lock has a fresh name inside the possessing class and is a public static object. Figure 3 illustrates this transformation process.

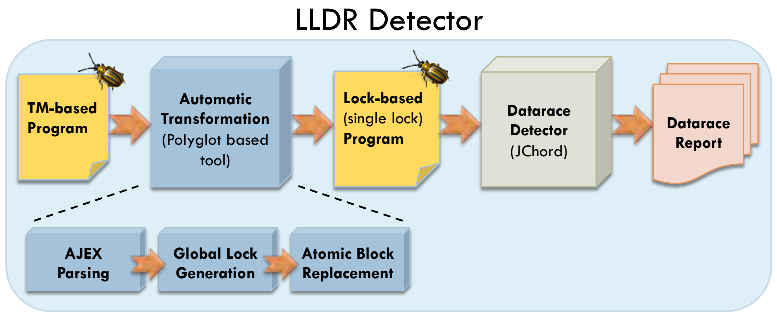

LLDR Detector
TM-based Program
Automatic Transformation (Polyglot based tool)
Lock-based (single lock) Program
Datarace Detector (JChord)
Datarace Report
AJEX Parsing
Global Lock Generation
Atomic Block Replacement

Fig. 2. The low-level datarace detection procedure.

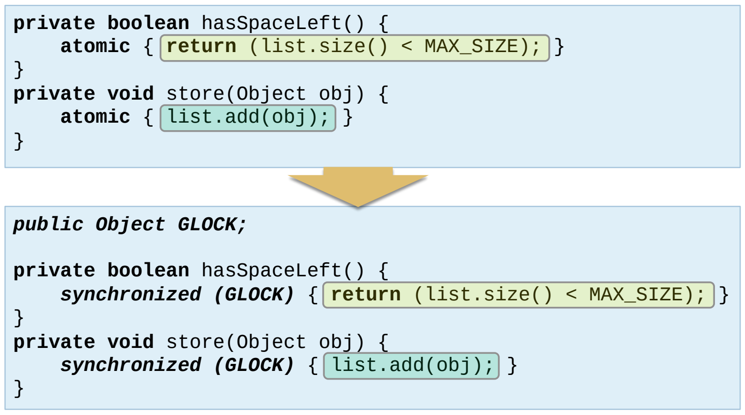
This automatic transformation process generates a Java compliant version of the original TM-based program. This program is then fed to a lock-based datarace detector. We used JChord [14] as a datarace detector, but other similar tools could be used instead.

2.2. Experiments

In order to validate our approach for transforming TM programs to synchronized single-lock programs, a set of validation tests have been carried out [21]. Some of these tests are well-known erroneous programs intended to benchmark validation tools like our own. Others were developed specifically to test our tool, containing simple stub programs with dataraces. We also tested Lee-TM [2], a renowned transactional memory benchmark. It was necessary to have two versions of each test, one using locks and another using TM. This implies that the existing tests meant for locks had to be manually rewritten using TM. We succeeded in keeping the original semantics (and errors) in the TM versions of the test programs, except for a small number of well identified cases.

Tests were carried out by initially running JChord on the lock-based version of each test. The results were registered for future reference. Then, we applied our approach to the TM versions of those tests, by transforming them into single global-lock programs and feeding them to JChord. The results were again registered. For each test, the results for both executions of JChord — in the original and transformed TM versions — were then compared. All the results obtained fit into one of the following scenarios: for tests where the lock-based and TM-based versions were strictly equivalent, the analysis results were equivalent as well; when the TM and lock-based versions of a test would have slightly different semantics, since some lock-based bugs could not be replicated using the TM model, results were slightly different, but all those differences could be clearly mapped to the semantic variations between the two versions.

As an example, consider the Lee-TM benchmark. By running JChord in the original lock-based version, we identified 52 dataraces. A careful analysis of



```
private boolean hasSpaceLeft() {
    atomic { return (list.size() < MAX_SIZE); }
}
private void store(Object obj) {
    atomic { list.add(obj); }
}
```

```
public Object GLOCK;

private boolean hasSpaceLeft() {
    synchronized (GLOCK) { return (list.size() < MAX_SIZE); }
}
private void store(Object obj) {
    synchronized (GLOCK) { list.add(obj); }
}
```

Fig. 3. Transformation of transactions into code blocks synchronized in a global lock.

these dataraces proved them all to be either false positives or, although confirmed, harmless. Running JChord in the single-lock version resulting from our automatic transformation process, we identified 48 dataraces, all with a direct correspondence to the dataraces observed in the original lock-based version. The remaining 4 dataraces correspond to *wrong-lock* situations, which do not exist in TM.

3. High-level Dataraces

A program that is free from low-level dataraces may still suffer from concurrency errors. Unlike low-level dataraces, high-level dataraces do not result from unsynchronized accesses to shared variables, but rather from a combination of multiple synchronized accesses, which may lead to incorrect behaviors if executed in a specific order.

As an example, consider the program in Listing 2, showing a bounded data structure whose size cannot go beyond `MAX_SIZE`. All accesses to the `list` fields are safely enclosed inside transactions, therefore no low-level datarace exists. But there is nonetheless a high-level datarace.

In the function `attemptToStore()`, a thread always checks for available room before storing an item in the queue. However, between the executions of `hasSpaceLeft()` and `store()`, the list may be filled by another concurrent thread executing the same code, leaving no space left; thus, the first thread would now be adding an element to an already full list. Both method calls to `hasSpaceLeft()` and `store()` should have been executed inside the same transaction. This was not the case, thus leading to a high-level datarace.

In the following sections we will discuss the conditions that may trigger high-level anomalies, propose a possible categorization of those anomalies, and present our approach for their identification.

```
private boolean hasSpaceLeft() {
    atomic { return (this.list.size() < MAX_SIZE); }
}

private synchronized void store(Object obj) {
    atomic { this.list.add(obj); }
}

public void attemptToStore(Object obj) {
    if (this.hasSpaceLeft()) {
        // list may become full!
        this.store(obj);
    }
}
```

Listing 2. Example of a High-level Anomaly

3.1. Thread Atomicity

High-level anomalies are related to sets of transactions involving different threads, which leave the program in an inconsistent state when executed in a specific order. This happens because two or more of the transactions executed by one thread are somehow related, making assumptions about each other (e.g., assuming success), but there is a scheduling in which another thread issues a concurrent transaction which breaks that assumption. The simplest way to solve this problem is to merge those related transactions into a single one. Furthermore, through empirical observation, it seems that many of such anomalies involve only three transactions. Two consecutive transactions from one thread and a third transaction from another thread, that when scheduled to run between the other two, causes an anomaly.

Without further information from the developer intention on the program semantics, at compile time it is not possible to infer all the relations among transactions. It is possible, however, to identify transactions that may or will affect other transactions, and use this information to identify potential high-level anomalies.

Consider a coordinate pair object shared between multiple threads. Assume that a thread T_1 issues a transaction $t_{1.1}$ to read value x, and then issues transaction $t_{1.2}$ to read y. In between them, thread T_2 could issue transaction $t_{2.1}$ which sets both values to 0, and so thread T_1 would have read values corresponding to the old x and new y (zero), when it is likely that both read operations were meant to read one single instant, i.e., either both before or after $t_{2.1}$. In this scenario, the final outcome is not equivalent to a situation in which both read operations were ran without interleaving. The property of a set of threads whose interleavings are guaranteed to be equivalent to their sequential execution is called *thread atomicity* [24], and will be further discussed in Section 4. It is common to pursue thread atomicity as being a correctness criterion.

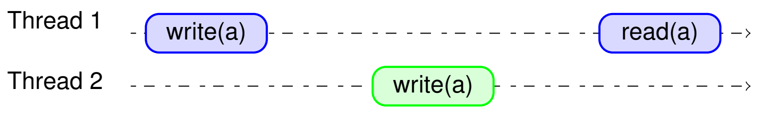


Fig. 4. An unserializable pattern which does not appear to be anomalous.

3.2. Anomaly Patterns in Transactional Memory Programs

Since full thread atomicity may be too restrictive, thus triggering too many false positive scenarios, we opted for a more relaxed semantic that allows a restricted number of atomicity violations. As an example of an atomicity violation which in principle is not an error, consider the example in Figure 4, where each rectangle corresponds to a transactional code block. The second operation in T_1 will be retrieving the results written by T_2. In order for this set of threads to be serializable, and thus thread atomic, all possible interleavings would have to be equivalent to the scenario in which the read immediately follows the write of the same thread.

However, given the specific context of TM and the set of operations presented in Figure 4, it seems unintuitive that this particular set would contain an error. The read operation is retrieving a, and it seems unlikely that an operation will be performed based on the value written before by the same thread, as it would possibly be already outdated. The only error scenario involving this particular setup would be the case in which after the read, the first thread would do a set of operations that depend on both, the value just read and the value previously written and assuming them to be equal.

We propose a framework for detecting a configurable set of patterns, and we opted to include only those most likely will result in concurrency anomalies. Out of all the patterns that incur in atomicity violations, we have isolated three highly suspicious patterns which describe possible high-level anomalies. These patterns are summarized in Figure 5 (with $x \neq y$).

Read–write–Read or ***RwR* —** Non-atomic global read. A thread reads a global state in two or more separate transactions, and the global state was changed by another thread meanwhile. If the first thread makes assumptions based on that state, it will most probably be a high-level anomaly.

Write–read–Write or ***WrW* —** Non-atomic global write. This is the opposite scenario from above. A thread is changing the global shared state in multiple separate transactions. Other thread reading the global state will observe this state as inconsistent.

Read–write–Write or ***RwW* —** Non-atomic *compare-and-swap*. In this pattern a thread checks a variable value, and based on that value it changes the state of the variable. If the variable was changed meanwhile, the update will probably may not make sense anymore.

Anomalies between two consecutive transactions can be defined a triple of transactions (T_1, T_2, T_3), such that the execution of T_2 by one thread interferes

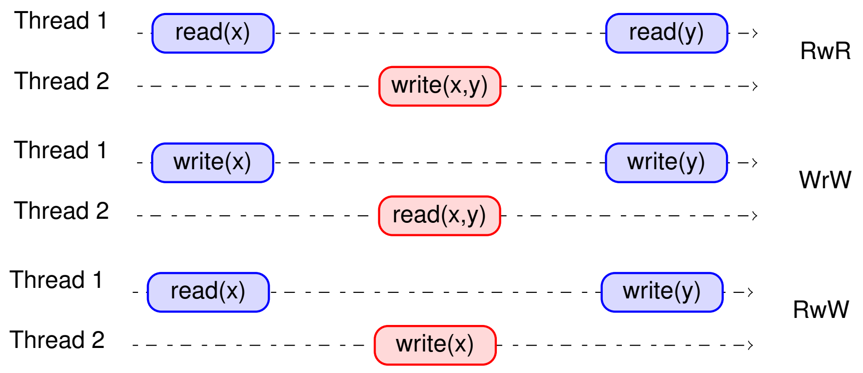


Fig. 5. Common anomalous access patterns.

with the normal execution of T_1—T_3 by another thread. It is common for a program to have multiple anomalies $A = (T_1, T_2, T_3)$ and $A' = (T_1, T_2', T_3)$. Even though these anomalies are of different nature, they reflect the lack of atomicity between the same pair of transactions. We define a *main anomaly* as a triple (T_1, S, T_2) were S is the set of transactions that interfere with the expected execution of T_1—T_3.

In the following, we will present our approach for statically matching suspicious patterns against the program source code, and will report on the experiments that assess the applicability and effectiveness of these patterns.

3.3. Symbolic Execution of Transactional Memory Programs

To detect high-level anomalies in TM programs, we perform a symbolic execution of the program and generate a set of possible execution traces of the transactional code. From these traces, we generate the set of possible interleavings of transactional code blocks and check if there are matches with any of the patterns identified in Section 3.2. Our approach for the detection of high-level anomalies in TM programs was also implemented resorting to the Polyglot [15] framework and AJEX [7].

The thread traces are obtained by performing a symbolic execution of the given program. When the program to be analyzed is loaded, all class declarations that contain main thread methods are retrieved. This includes classes that have a execution entry point such as a `main()` method, and classes that inherit from `java.lang.Thread` or `java.lang.Runnable` containing a `run()` method declaration. Hence, we obtain a list of all thread bootstrap methods. Statements in these thread methods are then analyzed. Whenever a transactional code block is found, it is added to the current trace, together with the full list of read and write operations of that transaction.

```
<?xml version="1.0" encoding="UTF-8"?>
<classes>
  <class id="a.package.AClass">
    <method id="doThings(Object,␣Object)">
      <changes>this</changes> <!-- Target object may change -->
      <changes>1</changes> <!-- First argument may change -->
    </method>
  </class>
</classes>
```

Listing 3. Example of the XML file that specified access to unavailable methods.

To find out the read and write operations of a method call, we in-line the called code, i.e., we replace the method call with the body of the target method, so that the transactions performed by that method are still seen as being performed by the current thread. Care must be taken not to perform infinite in-linings when in the presence of recursive methods.

In real world programs much code is already compiled, and the original source code is unavailable (this includes Java standard libraries). It may also happen that a program calls native methods, that may not be analyzed by our tool. When these cases arises our tools issues a warning, with the full qualified signature of the method it cannot analyze. With this information the user can build a database of the accesses performed by methods whose source code is not available. This is performed with an XML file as illustrated in Listing 3.

Additional challenges derive from disjunctions in the program control flow. When there are multiple possible execution flows, such as with *if-else* or *switch* statements, the current trace must still represent all possible executions. Instead of having numerous alternative traces for the same thread, a special disjunction node is added to the trace, symbolizing a disjunction point, where the execution can follow one of the multiple alternative paths. Thus, the trace actually takes the form of a tree representing all the transactional blocks in all the possible execution paths for a thread.

Finally, we also have to deal with loop structures in the input program. This is solved by considering the representative scenarios of the execution of loops. The trace tree only considers the cases in which the loop is not executed or is executed twice. We need to consider zero executions of the loop body, for the case in which the transaction that precedes and the one that follows the loop are both involved in an anomaly. When considering two executions of the loop we cover three different cases: when a transaction that precedes the loop is involved in an anomaly with a transaction in the loop; when a transaction in the loop is involved in an anomaly with the transaction that follows the loop; and when a transaction in the loop is involved in an anomaly with itself in the next iteration of the loop. It is not necessary to check for a single execution of the loop as two loop unrolls generate a super-set of the cases generated by a single loop unroll. It is also not necessary to consider more than two consecutive

Table 1. Experimental results summary.

Test Name	Total Anomalies	Total Warnings	Correct Warnings	False Warnings	Missed Anomalies
Connection [5]	2	3	2	1	0
Coordinates'03 [3]	2	7	2	5	0
Local Variable [3]	1	1	1	0	0
NASA [3]	1	1	1	0	0
Coordinates'04 [4]	1	2	1	1	0
Buffer [4]	0	1	0	1	0
Double-Check [4]	0	1	0	1	0
StringBuffer [10]	1	1	1	0	0
Account [23]	1	1	1	0	0
Jigsaw [23]	1	2	1	1	0
Over-reporting [23]	0	1	0	1	0
Under-reporting [23]	1	1	1	0	0
Allocate Vector [1]	1	2	1	1	0
Knight Moves [21]	1	1	1	0	0
Arithmetic Database [21]	1	2	2	0	1*
Total	**14**	**27**	**15**	**12**	**1**

* This anomaly was partially detected.

executions, since all the anomalies detected with three or more expansions of the loop body are duplicates of those detected with just two expansions.

3.4. Validation of the Approach

We ran a total of 15 tests for detecting high-level anomalies in TM programs. Many of those tests consist of small programs taken from the literature [3, 4, 5, 10, 23] with well studied high-level anomalies. The *Allocate Vector* test was taken from the IBM concurrency benchmark repository [1]. We also developed two of the tests [21]. All the 15 test programs were analyzed with success by our tool.

We measure the effectiveness of our high-level datarace detector by the number of *main anomalies* reported, as defined in Section 3.2. This metric is better than the regular, pattern specific, anomaly count, since it reflects the number of spots that lack inter-transaction atomicity. It is also more meaningful for the user since it point our the transactions that should be merged.

The results are summarized in Table 1. In a total of 14 anomalies present in these programs, 13 were correctly pointed out.

In the *Arithmetic Database* test our tool indicates two anomalies; these anomalies are part of a larger anomaly which was not detected as a whole. This program performs four transactions that should be merged in a single one. Since our detection approach is based on the most common case of anomalies between a pair of adjacent transactions, we fail to see the lack of atomicity of these four transactions. It is also worth noticing that two specific cases of the anomaly were reported.

In addition to the correctly detected anomalies, there were also 12 false positives (45% of total warnings). We group the causes for these imprecisions in 4 different categories.

Out of these 12 false warnings, 2 were due to redundant read operations: when reading `object.field`, we consider that two readings are actually being performed, one to `object` and another to `field`. It makes no sense for two instances of this statement to be involved in an anomaly. It is possible to eliminate these false positives by tailoring the analysis and consider only one read operation in the access to `field`.

Another 4 false positives are related to cases for which additional semantic information would have to be provided by the software developer or somehow inferred. These false warnings could be eliminated with the aid of other available techniques, such as *points-to* and *may-happen-in-parallel* analyses.

Two other false positives could be eliminated by refining the definition of the *anomaly patterns* described in Section 3.2. For example, an *RwR* anomaly could be ignored if the last transaction reads both values involved.

Finally, 4 false warnings which are matched by our anomaly patterns are definite false positives. Further study would be necessary to adapt the anomaly patterns in order to leave out these correct accesses, without compromising the precision of the detector. If we fix all the previous false positives but these last four, we will be able to reduce the percentage of false positives from 45% to 15%, which is much better than what can be observed in related works.

4. Related Work

Low-level datarace detection, either by observing a program's execution (dynamic approach) or its specification (static approach) has been an area of intense research [6,8,9,13,14,16,17]. We are unaware of any work that specifically targets the detection of low-level dataraces in the TM setting. However, we have shown that current algorithms and tools, which are intended for use with lock-based mechanisms, may as well be applied to transformed TM programs.

There are some relevant works on high-level anomaly detection that, although not targeting the TM setting, share some principles and features with our own work. One of the earliest works on the subject is the one by Wang and Stoller [24]. They introduce the concept of thread atomicity, with *atomicity* having a different meaning than the one stated in the ACID properties provided by TM systems. In this case, thread atomicity is more related to *serializability*, and it means that any concurrent execution of a set of threads must be equivalent to some sequential execution of the same set of threads. Wang and Stoller provide two algorithms for dynamically (i.e., at runtime) finding atomicity violations. Other authors have based on this work to develop other approaches [5,10]. Our approach, being less strict than the one from Wang and Stoller [24], tends to be more precise and generates much less false positives.

An attempt to provide a more accurate definition of anomalies is the work on *High-Level Dataraces* (HLDRs) by Artho et. al [3]. Informally, an HLDR in this context refers to variables that are related and should be accessed together, but there is some thread that does not access that variable set atomically. This is different from thread atomicity, which considers the interaction between transactions, without regard for relations between variables.

Because HLDR is concerned with sets of related variables, some atomicity violations are not regarded as anomalies, such as those concerning only to one variable. On the other hand, it is possible that an HLDR does not incur in an atomicity violation. This work is in some way related to ours, in that it attempts to increase the precision of thread atomicity by reducing the false positives cases. However, while our approach is to simply disregard some atomicity violations as safe, the work by Artho founds a new definition, which still exhibits some false positives, and also introduces some false negatives. This work is also related to our in that they both automatically infer data relationships and do not require processing user annotations which state those relationships.

A different approach has been taken by Vaziri et. al [22]. Their work focuses on a static pattern matching approach. The patterns reflect each of all the possible situations that may lead to an atomicity violation. The anomalies are detected based on sets of variables that should be handled as a whole. To this end, the user must explicitly declare the sets of values that are related. This work is similar to ours in that both approaches are static, and both follow a pattern-matching scheme. However, our approach is intended to be applied to existing programs, and so it assumes that any set of variables may be related. Contrarily, the work by Vaziri demands that the user explicitly declares which sets of variables are meant to be treated atomically, and so it can trigger anomalies on all atomicity violations, without too many false positives.

5. Concluding Remarks

In this paper we proposed to detect low-level dataraces in transactional memory programs by explore the correspondence between low-level dataraces in these programs with equivalent low-level dataraces in lock-based programs. We proposed to convert all memory transactions in a program into synchronized blocks, all synchronizing in a single global lock. This was achieved with static analysis of the source code and a source-to-source transformation.

Application of this technique to well known test programs proved to be effective in the detection of low-level anomalies in transactional memory programs.

We have analyzed common criteria for reporting high-level anomalies, and attempted to provide a more useful criteria by defining three anomaly patterns. A new approach to static detection of high-level concurrency anomalies in Transactional Memory programs was defined and implemented. This new approach works by conservatively tracing transactions and matching the interference between each consecutive pair of transactions against a set of well defined anomaly patterns. Our approach raises false positives, although at an acceptable level; and well known techniques can be applied to prune the false warnings to and even lower level. When compared with the existing reports from literature, these results are, in general, considerably better. We may therefore conclude that our conservative tracing of transactions is a reasonable indicator of the behavior of a program, since our results rival with those of dynamic approaches.

The developed framework can be improved by further refining the error patterns. The addition of *points-to* and *may-happen-in-parallel* analyses would help to improve the tool by reducing the number of states to be analyzed. Other improvements could be achieved by enabling the analysis of standard or unavailable methods, and by solving the issue of redundant read accesses.

Our approach is novel because it is based in static analysis; it extracts conservative trace trees aiming at reducing the number of states to be analyzed; and it detects anomalies using a heuristic based in a set of suspicious patterns believed to be anomalous.

References

1. IBM's Concurrency Testing Repository, `https://qp.research.ibm.com/concurrency_testing`
2. Ansari, M., et al.: Lee-TM: A non-trivial benchmark suite for transactional memory. In: Proceedings of ICA3PP '08. pp. 196–207. Springer-Verlag, Berlin (2008)
3. Artho, C., Havelund, K., Biere, A.: High-level data races. Softw. Test., Verif. Reliab. 13(4), 207–227 (2003)
4. Artho, C., Havelund, K., Biere, A.: Using block-local atomicity to detect stale-value concurrency errors. In: Wang, F. (ed.) ATVA. Lecture Notes in Computer Science, vol. 3299, pp. 150–164. Springer (2004)
5. Beckman, N.E., Bierhoff, K., Aldrich, J.: Verifying correct usage of atomic blocks and typestate. SIGPLAN Not. 43(10), 227–244 (2008)
6. deok Choi, J., Loginov, A., Sarkar, V., Logthor, A.: Static datarace analysis for multithreaded object-oriented programs. Tech. rep., IBM Research Division, Thomas J. Watson Research Centre (2001)
7. Dias, R., Teixeira, B.: Ajex: A source-to-source java stm framework compiler. Tech. rep., DI-FCT/UNL (Apr 2009)
8. Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. SIGPLAN Not. 26(12), 85–96 (1991)
9. Flanagan, C., Freund, S.N.: Type-based race detection for Java. SIGPLAN Not. 35(5), 219–232 (2000)
10. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proceedings of POPL'04. pp. 256–267. ACM, New York, NY, USA (2004)

11. Herlihy, M., Luchangco, V., Moir, M., Scherer, I.W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of PODC'03. pp. 92–101. ACM, New York, NY, USA (2003)
12. Herlihy, M., Luchangco, V., Moir, M., William N. Scherer, I.: Software transactional memory for dynamic-sized data structures. In: PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing. pp. 92–101. ACM, New York, NY, USA (2003)
13. Marino, D., Musuvathi, M., Narayanasamy, S.: Literace: effective sampling for lightweight data-race detection. In: Proceedings of PLDI'09. pp. 134–143. ACM, New York, NY, USA (2009)
14. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: PLDI. pp. 308–319. ACM Press (2006)
15. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for java. In: CC. pp. 138–152 (2003)
16. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. SIGPLAN Not. 38(10), 167–178 (2003)
17. Qi, Y., Das, R., Luo, Z.D., Trotter, M.: Multicoresdk: a practical and efficient data race detector for real-world applications. In: Proceedings of the 7th Workshop on Parallel and Distributed Systems. pp. 1–11. ACM, New York, NY, USA (2009)
18. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of PODC'95. pp. 204–213. ACM, New York, NY, USA (1995)
19. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of PODC'95. pp. 204–213. ACM, New York, NY, USA (1995)
20. Teixeira, B., Sousa, D., Lourenço, J., Dias, R., Farchi, E.: Detection of transactional memory anomalies using static analysis. In: Proceedings of PADTAD'10. pp. 26–36. ACM, New York, NY, USA (2010)
21. Teixeira, B.: Static Detection of Anomalies in Transactional Memory Programs. Master's thesis, Universidade Nova de Lisboa (Apr 2010)
22. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: Proceedings of POPL'06. pp. 334–345. ACM, New York, NY, USA (2006)
23. von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. In: Journal of Object Technology. p. 2004 (2003)
24. Wang, L., Stoller, S.D.: Run-time analysis for atomicity. Electronic Notes in Theoretical Computer Science 89(2), 191–209 (2003), RV '2003, Run-time Verification (Satellite Workshop of CAV '03)

Joao Lourenço is an Assistant Professor at the Computer Science Department of New University of Lisbon, Portugal. He received his MSc and PhD in 1995 and 2004 respectively, both from the New University of Lisbon. His current research interests focus on Software Transactional Memory, more specifically in Transactional Memory run-time and programming language support, and also software development environments and tools for TM, including testing and debugging tools. He is currently a member of the Transactional Systems Research Team (TrxSys) at the Center for Informatics and Information Technologies (CITI).

Diogo Sousa is a BSc student at the Computer Science Department of New University of Lisbon, Portugal. He will conclude his BSc in July 2011 and plans to enroll in the MSc immediately after. He has been collaborating in the research activities of the Transactional Systems Research Team (TrxSys) at the Center for Informatics and Information Technologies (CITI) since his first year of the BSc, always under the supervision of Dr. Joao Lourenço. He also frequently participates in programming contests and is a supporter of Free Software.

Bruno Teixeira is currently working as an IT Consultant at a private corporation in Lisbon, Portugal. He received his MSc in 2010 from the New University of Lisbon with a dissertation entitled "Static Detection of Anomalies in Transactional Memory Programs" and was advised by Dr. Joao Lourenço.

Ricardo Dias is a PhD student at the Computer Science Department of New University of Lisbon, Portugal. He received his MSc in 2008 from the New University of Lisbon. His current research interests focus on Software Transactional Memory, more specifically in static verification of isolation anomalies in transactional programs.

Received: January 10, 2011; Accepted: March 2, 2011.

CIP – Каталогизација у публикацији
Народна библиотека Србије, Београд

004

COMPUTER Science and Information Systems : the International journal / Editor-in-Chief Mirjana Ivanović. – Vol. 8, No 2 (2011) - . – Novi Sad (Trg D. Obradovića 3): ComSIS Consortium, 2011 - (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
Library administration: M. Petković
Printed by: Sigra star, Beograd

ComSIS Computer Science and Information Systems

Volume 8, Number 2, Special Issue, May 2011

Contents

Editorial

Invited Papers

225 Challenges and Directions in Formalizing the Semantics of Modeling Languages
Barrett R. Bryant, Jeff Gray, Marjan Mernik, Peter J. Clarke, Robert B. France, Gabor Karsai

255 Software Agents: Languages, Tools, Platforms
Costin Bădică, Zoran Budimac, Hans-Dieter Burkhard, Mirjana Ivanović

Regular Papers

299 SPEM Ontology as the Semantic Notation for Method and Process Definition in the Context of SWEBOK
Miroslav Líška, Pavol Navrat

317 Ontology Driven Development of Domain-Specific Languages
Ines Čeh, Matej Črepinšek, Tomaž Kosar, Marjan Mernik

343 Domain-Specific Language for Coordination Patterns
Nuno Oliveira, Nuno Rodrigues, Pedro Rangel Henriques

361 From DCOM Interfaces to Domain-Specific Modeling Language: A Case Study on the Sequencer
Tomaž Kos, Tomaž Kosar, Jure Knez, Marjan Mernik

379 A DSL for PIM Specifications: Design and Attribute Grammar based Implementation
Ivan Luković, Maria João Varanda Pereira, Nuno Oliveira, Daniela da Cruz, Pedro Rangel Henriques

405 UML Profile for Specifying User Interfaces of Business Applications
Branko Perišić, Gordana Milosavljević, Igor Dejanović, Branko Milosavljević

427 Formalizing Business Process Specifications
Andreas Speck, Sven Feja, Sören Witt, Elke Pulvermüller, Marcel Schulz

447 An Approach to Assess and Compare Quality of Security Models
Raimundas Matulevičius, Henri Lakk, Marion Lepmets

477 GammaPolarSlicer
Sérgio Areias, Daniela da Cruz, Pedro Rangel Henriques, Jorge Sousa Pinto

501 Animation of Tile-Based Games Automatically Derived from Simulation Specifications
Jan Wolter, Bastian Cramer, Uwe Kastens

517 Solving Difficult LR Parsing Conflicts by Postponing Them
C. Rodriguez-Leon, L. Garcia-Forte

533 Detecting Concurrency Anomalies in Transactional Memory Programs
João Lourenço, Diogo Sousa, Bruno Teixeira, Ricardo Dias

ISSN: 1820-0214

Computer Science and Information Systems

Vol 8, No 2, May 2011