DOI: 10.2298/CSIS120104026G

A Programming Language Independent Framework for Metrics-based Software Evolution and Analysis

Črt Gerlec[1], Gordana Rakić[2], Zoran Budimac[2], Marjan Heričko[1]

[1] Institute of Informatics
Faculty of Electrical Engineering and Computer Science
University of Maribor
Smetanova ulica 17, 2000 Maribor, Slovenia
{crt.gerlec,marjan.hericko}@uni-mb.si
[2] Department of Mathematics and informatics
Faculty of Science
University of Novi Sad
Trg Dositeja Obradovića 4, 2100 Novi Sad, Serbia
{goca,zjb}@dmi.uns.ac.rs

Abstract. Knowledge about different aspects of software quality during software evolution can be valuable information for developers and project managers. It helps to reduce the number of defects and improves the internal structure of software. However, determining software's quality and structure in heterogeneous systems is a difficult task. In this paper, a programming language independent framework for evaluating software metrics and analyzing software structure during software development and its evolution will be presented. The framework consists of the SMIILE tool for calculation of software metrics, extended with an analysis of software structure. The data are stored in a central repository via enriched Concrete Syntax Tree (eCST) for universal source code representation. The framework is demonstrated in a case study. The development of such a framework is a step forward to consistent support for software evolution by providing a change analysis and quality control. The significance of this consistency is growing today, when software projects are more complex, consisting of components developed in diverse programming languages.

Keywords: Software evolution, software development, software quality, software structure, software metrics, syntax tree

1. Introduction

From the beginning of the application of software engineering, engineers have been striving to develop quality and maintainable software products. Therefore, software evolution and its quality have become an important research discipline. Early versioning systems like the Source Code Control System made it possible to record the sequential versions of software products [41]. Such software

history has been important to understand what, where and when a change was applied. Beside versioning systems, software quality measures have been researched. The first published book that describe software metrics appeared in 1976 [20] but the first attempts at applying software metrics had already taken place in the late 1960s [17]. With the spread of software metrics, a need for the appropriate storage of such data grew. In 1993, Pfleeger described the importance of data collecting and determined their success in a metrics program [37].

Various approaches have been used to analyze software evolution. The majority of them are programming language specific. Their meta-models are not general and do not enable a structural software comparison between different systems. Thus, making a comparison between the structural software evolution of two systems or of two components with the same system that are written in different programming languages (e.g. Java and C#), is not possible. A similar problem can be found in the field of software metrics. Existing approaches that define the metrics and its algorithms are programming language specific. Furthermore, the algorithms usually differ between the tools. Thus, a software metric comparison in heterogeneous systems is not accurate.

The purpose of this study is to deal with the problems surrounding programming language dependent frameworks and approaches that describe software metrics and software structure. Thus, a general framework that allows a programming language independent representation and evaluation of software artifacts has been developed. It consists of three major components. The first component of the framework is a language independent meta-model for representing a source code structure. The main purpose is to provide sufficient data for a further evolutionary analysis based on software structures (e.g. detecting structural source code changes between sequential software versions). The second one is the SMIILE (Software Metrics Independent of Input LanguagE) tool. Its main advantage is to define an universal implementation of metric algorithms (e.g. algorithm for the cyclomatic complexity) built upon the meta-model. Both components are based on the enriched Concrete Syntax Tree (eCST) that represents an internal representation of source code. The eCST is built on "universal" nodes that are common for all programming languages. However, in order to store the software artifacts and conduct a deeper analysis, an appropriate repository is needed. To fulfill this demand, a specific repository was build and integrated as the third component in the framework.

In our case study, the application of the framework will be shown. Its goal is to apply the framework (i.e. meta-models) in practice by using the SMIILE tool for defining and calculating software metric values and by using structural source code representation from different programming languages.

The contribution of this paper is the development of a programming language independent framework for metrics-based software evolution and analysis. This goal was achieved by (1) adjusting the eCST concept in order to support a language-independent source code structure representation that en-

ables evolutionary analysis based on software structure and by (2) integrating the SMIILE tool with a central repository that supports eCST.

This paper is organized as follows: The background, needed for understanding the study, and the motivation are described in section 2. Then, the preliminary works are introduced in section 3. The programming language independent framework for metrics and structural analysis in software evolution is briefly described in section 4. In the section 5, the framework application is presented with a case study. The validity and limitations of this research are stated in section 6. In the section 7, state-of-the-art tools and approaches for analyzing software metrics and software structure are introduced. In the last section, the conclusion and ideas for future work are provided.

2. Background and motivation

This section describes three important notions of the study (i.e. software evolution, software metrics and software repository) and the motivation.

2.1. Software evolution

The field of software evolution has become an interesting area over the last decade, leading to an increase in the amount of research on the subject [14]. Lehman et al. [31] describes two perspectives on software evolution. The first perspective focuses on the questions of "what and why" and describes the nature of software evolution and its properties. On the other hand, the second perspective is focused on the word "how" and covers areas like the theories, abstractions, languages, activities, methods and tools required to evolve software.

Software evolution could also be understood as continuous adaptation. Software changes, that are caused by an adaptation process, are usually partitioned into three general classes [33]. The first class includes corrections that tend to be fixes of source code errors. However, there are also some other error fixes that are related to software design, architecture and requirements. The next class consists of improvements. They tend to include things like increases in performance, usability, maintainability, etc. The last class comprises enhancements that represent new features or functions that are visible to the users of the end system.

Software systems evolve continuously in order to satisfy all users' needs and requirements. The research in [26] showed that the software history is a good indicator for its quality. Therefore, it is vital for companies to ensure mechanism that tracks the changes during the development in order to minimize the risk for potential new bugs.

Software changes are part of software evolution. Thus, it is important to analyze these changes from a structural and qualitative point of view and then compare the results.

Structural source code changes are constant during software development. They are usually made when new functionality is added to the existing software product or during the updates. Moreover, changes are also made in refactoring and debugging processes. In this paper, a structural source code change is defined as an object-oriented change on a class (e.g. the add/remove method) between two sequential versions. The examples of structural source code changes are:

- add parameter, field and method,
- remove parameter, field and method,
- hide and unhide method,
- rename method,
- move attribute, method and class,
- extract superclass, interface and class,
- pull up field and method,
- push down field and method, and
- inline class.

In the sense of software evolution, our study focuses on defining a programming language independent meta-model that is based on the time (i.e. version) component. Its intent is to collect sufficient data about software structure and its quality properties. Such a meta-model enables further evolutionary analysis upon the collected data. However, the change detection process uses several rules in order to identify structural changes between two source code versions. Each rule represents one change type (e.g. add method) and usually accepts two parameters. For example, the first parameter is metadata for a class in version n and the second parameter represents the same class in the next version (i.e. n+1). If the metadata for the same class in two sequential versions fulfills the demands of the rules, the change type that the rule represents, was used on the class. Even though the change detection process has already been implemented, it is out of the scope of this paper.

2.2. Software product metrics

The measuring and continual monitoring of a software product is crucial for success in the software development process. From this perspective, software metrics, the software metrics tool and the software metrics repository are crucial notions.

Software metrics can be defined as numerical values that reflect the properties of a software development processes and software products [34]. There are numerous categorizations of software metrics but when considering the measurements, target metrics can be divided into three main categories: product metrics, process metrics and project metrics [29]. In the rest of the paper, we will deal with product metrics and especially code metrics as a sub-category of product metrics. First, we will specify some of the product metrics used in the rest of the paper:

- Cyclomatic Complexity (CC) - reflects structure complexity based on control-flow structures in the program.
- Halstead Metrics (H) - reflects the complexity of the program based on number of operators and operands.
- Lines of Code (LOC) - represents the length of the source code expressed in the number of lines of source code. It is common to differentiate between the number of lines of comment (CLOC), source code (SLOC), etc.
- Object Oriented metrics (OO) - the family of metrics related to the object orientation of software. On the other hand, the term *design metrics* is often used and usually describes metrics related to characteristics of object oriented development and design. However, some examples of the metrics used in this paper are:
  - Number of Classes (NOC) - reflects the number of classes contained in the package, namespace, project, etc.
  - Number of Interfaces (NOI) - reflects the number of interfaces contained in the package, namespace, project, etc.
  - Number of Methods (NOM) - reflects the number of methods declared in the unit (class, interface, etc).
  - Number of Properties (NOP) - reflects the number of properties declared in the unit (class, interface, etc).
  - Number of Attributes (NOA) - reflects the number of attributes declared in the unit (class, interface, etc).

Nowadays, various software metrics tools are used for automatic calculations of software metrics. However, achieveing accuracy of the gathered metric values and the appropriate interpretation of extracted data is often the hardest step.

In section 7, problems in the area of consistent and systematic application of software metrics will be presented. During the exploration, the strong dependency of the applicability of software metrics on an input programming language was recognized as one of the main weaknesses in this field. Introducing an enriched Concrete Syntax Tree (eCST) for intermediate representation of the source code resulted in a step towards programming language independence.

2.3. Software repositories

In order to perform a detailed measurement and analysis and interpretation of numerous software metric values, a repository is needed. Its aim is to collect [24], store and enable access to a wide range of metric values (e.g. product, process and resource metric values) collected from software products, software development processes and project management tools. The collected data, extracted with different tools, helps project leaders and development teams get a better overview of a project.

Software repositories have been recognized as an important tool in the past. Carnegie et al. [27] suggested that software organizations should implement systems to define, collect, store, analyze and use process data. Furthermore,

Basili [12] suggested that data analysis routines should be implemented in order to extract derived data from the row data. Then, all collected data should be stored in a computerized database. In the study conducted by Goeminne et al. [22] the term "repository" is defined as follows: "A data source containing information that is relevant to the software product or process, and that can be accessed and modified by different persons by using their identity." Repositories collect various properties of software systems (e.g the version of the source code). As mentioned earlier, metrics repositories store data about a software product (i.e. software metrics) while other repositories store different data (e.g. properties of software processes). However, with historical insight over the software properties, users become familiar with changes that were made over time. With such knowledge, users are able to predict changes in the future and act if the negative trend is detected. Thus, the establishment of software repositories is sensible in organizations.

2.4. Motivation

Related research has also shown that there is no fully consistent tool support for measurement and analysis during software development and maintenance. The tools used for these purposes have some limitations (e.g. limited programming language support, weak and inconsistent usage of metrics and/or testing techniques, etc).

Large software systems are written in several programming languages. In order to ensure a high level of software quality, we have to know the condition of every part of a system. Furthermore, in order to evaluate such systems, different tools have to be used. However, these tools usually provide inconsistent values for software metrics [36], [32], [43] and therefore, a comparison between different parts of a system, written in different programming languages, is not applicable.

In the field of software evolution, which enforces techniques such as advising, recommending and the automating of refactoring and reengineering, solutions that are based on a common intermediate structure can be a key supporting element. This support could be based on metrics, testing and deeper static and structure analysis. The development of such support would introduce new values into the field of software engineering. For all of these reasons, a proposed universal tree could be an appropriate internal representation applicable toward all stated goals. Universality of internal structure is important for meeting consistency in all fields.

By realization of this idea a key benefit could be made from language independence of eCST and its universality and broad applicability.

3. Preliminary work

In this section, the preliminary work for developing a tool for change analysis during software evolution will be described. Furthermore, a description of eCST

and the original idea of an application of underlying trees in the development of the SMIILE tool will be presented.

3.1. A tool for mining software repositories

The tool for identifying structural source code changes was presented in [19]. Its aim is to extract data from software repositories (e.g. subversion) and store them into the meta-model in order to identify structural source code changes between sequential versions. The change identification process is based on a set of change rules. They are applied between different versions represented by the meta-models. If demands of the rule are fulfilled, the change type is found. In this study, 26 different rules for detecting change types were used. The results showed that the tool could be used to analyze source code changes in software repositories. On the other hand, the tool also has some limitations. The main weakness is a programming language dependency. The current tool only supports C# and VisualBasic programming language. The main problem is direct relation between source-code and the meta-model. In order to overcome this limitation, a universal intermediate representation of source code is needed.

3.2. Introducing of eCST

The motivation for introducing eCST as a new intermediate representation of the source code is described in section 2.4.

Originally, tools used a Concrete Syntax Tree (CST) for the representation of source code. This tree is usually an intermediate product of a parser generator. It takes language grammar as an input and returns a language scanner and parser as output. The grammar rules determine the manner in which the syntax tree, as an intermediate structure, will be generated [23].

A CST represents concrete source code elements attached to a corresponding construction in a language syntax. Although this tree is quite rich, it is still unaware of sophisticated details about the meaning of syntax elements and their role in certain problems (e.g. algorithms for the calculation of software metrics). We enriched CST by adding universal nodes to mark elements to become a recognizable independent for input programming language. The catalog of universal nodes used in the prototype can be found in the appendix, in table 8.

To illustrate this technique and to achieve the independence of a programming language, we provide the following simple example [38]. It illustrates the problems in the calculation of a CC metric via the predicate counting method.

The simple loop statement (REPEAT), written in Modula-2, and the corresponding one (do-while), written in Java, are stated as in table 1.

Although the given statements have different syntax, they express the same functionality: some statements in the code will be repeated until parameter *i* becomes greater than parameter *j*. In addition to the different syntax, a condition for leaving the loop is oppositely stated. First, the condition expresses what

Table 1. Loop statements

```REPEAT ... Some statements ... UNTIL(i > j);```	```do{ ...Some statements... }while(i <= j);```
(a) REPEAT (Modula-2)	(b) do-while (Java)

condition should be fulfilled to leave the loop, while the second one states the condition to continue looping.

Simplified syntax trees representing these given statements are illustrated in Figure 1.

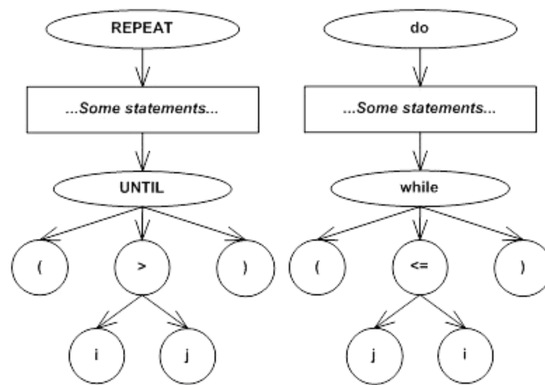

Fig. 1. Simplified CST for REAPEAT-UNTIL (left) and do-while (right) statements

For the implementation of a CC algorithm, a *REPEAT* and a *WHILE* loop have to be recognized and then increment the current CC value by 1. It is clear that by using CST for source code representation, two implementations or at least two conditions to recognize these loops in the tree are needed. By adding universal nodes (i.e. *LOOP_STATEMENT*) as a parent of sub-trees, that represent these two segments of source code, the goal by only one condition in the implementation of the CC algorithm is met. A universal node, CONDITION, was also added in order to mark the condition for leaving the loop repetition (Figure 2).

By adding all the needed universal nodes [40], the algorithms for the CC metric could be implemented independently of a programming language. The only requirement is that there is a language grammar to modify and generate an appropriate parser that is then used for generating eCST.

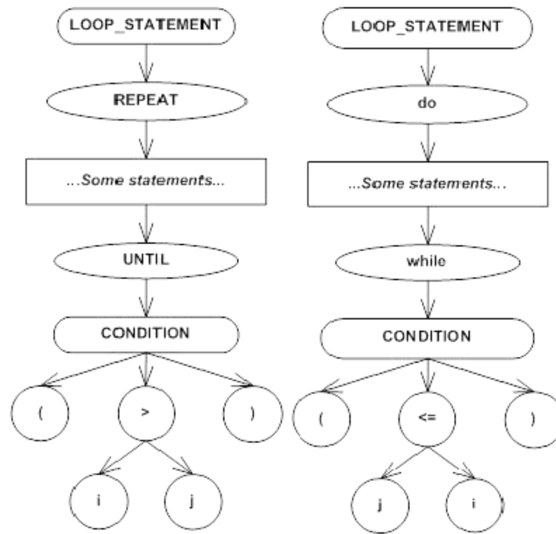

Fig. 2. Simplified eCST for REAPEAT-UNTIL (left) and do-while (right) statements

Enriching the CST by adding universal nodes is done at a grammar level. In the input language grammar, in the corresponding rule, we simply build in an imaginary node. For example, in the rule where a control structures (e.g. if, case, switch, etc.) are defined, an appropriate universal node is created. In the ANTLR [9] (the compiler generator that was used in our study) this is possible via a simple extension of the appropriate rule in the form of a declaration of a new node that is automatically added to the syntax tree during its creation process. The list of universal nodes required for implementing CC algorithm is given in [40], while a full description of the eCST, generating process and storing is presented in [39]. The possible broader applicability of eCST in different software engineering fields is described in [38].

It should be noted that the CC metric was chosen as a characteristic example for presenting the usefulness of the eCST in the sense of language independence. The LOC metric is less sensitive to the syntax of a programming language. However, a generated eCST is stored in an XML file based on a recursive definition [39]. Each node contains information about the location of the element in the source code (line and column), its text and node name, an index of the element and nodes that contain children (i.e. sub-trees). In such a structure, we can find all the necessary data for calculating the LOC metric. From the first element in the underlying sub-tree we can identify the starting line number, and from the last element in the last sub-tree we can identify the ending line number. Using the first and last line, we can easily calculate the LOC metric for a certain unit.

3.3. The SMIILE Tool

The SMIILE tool is a software metrics tool with the following general goals:

- independence of an input programming language,
- broad set of software metrics supported and
- support of software metrics history.

For an input source code, the SMIILE tool will execute the steps in two phases (Figure 3).

- Phase 1:
  - Recognition of the input programming language based on the input file extension.
  - Reading data about the language.
  - Calling an appropriate scanner and parser. Scanner and parser is generated by an ANTLR parser generator [9] from grammar containing rules for extending CST to eCST.
  - Tree generation that represents the provided source code and translates it into XML format. This process forms the basis for applying different algorithms (e.g. algorithms for the calculation of software metrics)
- Phase 2:
  - Reading the tree structure form XML to eCST.
  - Calculating software metric values.
  - Storing software metric values in XML.

The SMIILE was used on several different programming languages (object-oriented Java and C#, procedural Module-2 and Pascal and legacy COBOL). Furthermore, several metrics were used and implemented. We have chosen two of them (LOC and CC) in order to demonstrate the universality of the model. The LOC metric calculation algorithm is executable on a lexical level, while the CC metric is sensitive to input language syntax (illustrated by the example in the previous subsection). To implement algorithms for calculating the CC by predicate counting and at the same time to meet the language independence of this implementation, we introduced universal nodes for each element of language syntax figuring in the algorithm. However, the eCST is designed in such a way as to support any programming languages.

The catalog of universal nodes used in the implementation of the CC metric is specified in [40]. The full catalogue of universal nodes used in the current prototype of SMIILE tool can be seen in the appendix (in table 8). In the following table (table 2) we will only introduce those universal nodes referred to in this paper.

The storage of the SMIILE tool's source code representation and metrics history can be divided into two parts. In the first part, the eCST representation of a source code is stored in an XML file that represents the basis for metric calculations. In the next part, software metrics are calculated and stored in a separate XML file that contain metric values. In other words, for each version of a software the SMIILE tool generates two xml files (i.e. eCST representation and metric values). However, the aim of this study is to integrate software metrics history with a repository.

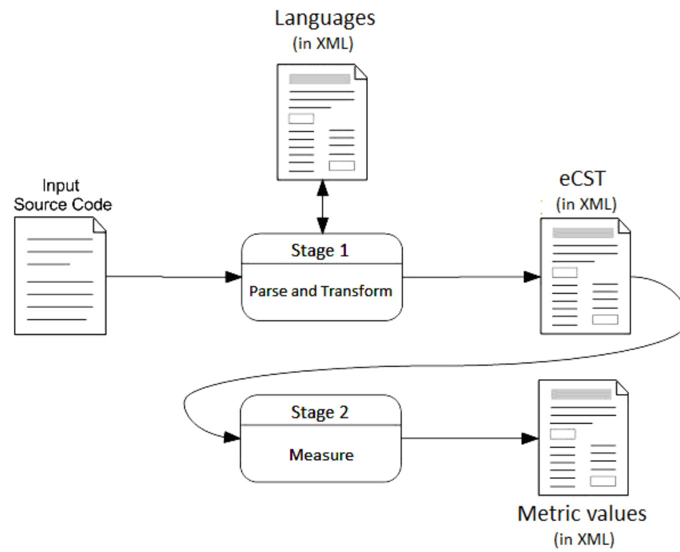

Fig. 3. SMIILE Tool Architecture

Table 2. Catalog of universal nodes used for integration eCST with the framework.

Universal node	coresponding element of language syntax
PACKAGE_DECL	package, workspace,. . .
CONCRETE_UNIT_DECL	class, implementation module,. . .
ABSTRACT_UNIT_DECL	abstrat class, etc.
INTERFACE_UNIT_DECL	interface, definition module,. . .
EXTENDED_BASE_UNITS	extended class
IMPLEMENTED_INTERFACE_UNITS	implemented interface, corresponding definition module,. . .
ATTRIBUTE_DECL	attribute, field,. . .
PROPERTY_DECL	property
FUNCTION_DECL	method, procedure, function
PARAMETERS_DECL	parameters of the method, procedure, function,. . .
NAME	name of any element (unit, function, attribute,. . .)
TYPE	type of any element (unit, function, attribute,. . .)

4. Framework for analyzing software evolution

In this section, the programming language independent framework for analyzing software structure and metrics is presented (figure 4). In order to overcome the existing problems of meta-models and approaches for software evolution analysis, our framework focuses on the following aspects:

- A programming language independent framework for analyzing software evolution built on the eCST.
  - An eCST-based meta-model that provides sufficient meta-data for analyzing software structure and its changes.
  - An eCTS-based meta-model for representing software syntax that enables metric calculations based on universal nodes.
- A software repository that stores meta-data and enables further analysis.

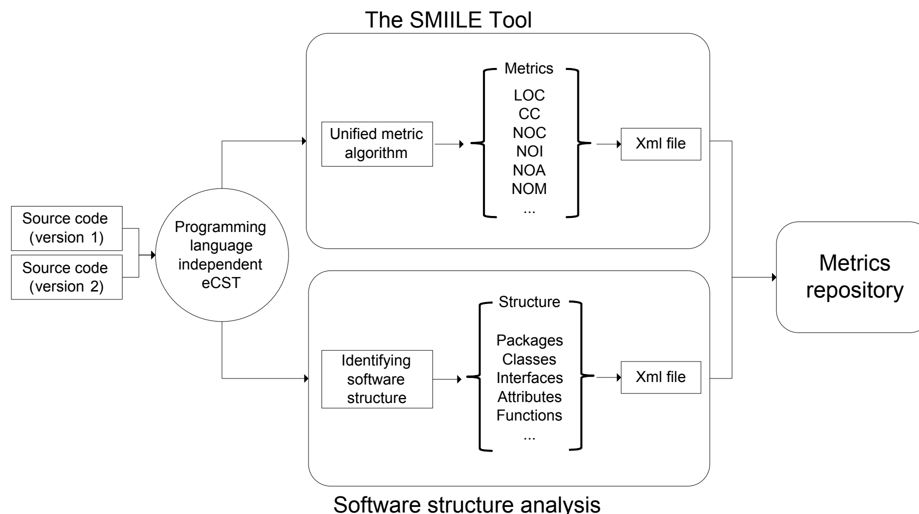

Fig. 4. The programming language independent framework for analyzing software structure and metrics.

The framework is built upon the eCST that includes ”universal” nodes, which are common for various programming languages. It consists of three components. The first component is responsible for defining time in the software development life cycle and is represented with the *version* entity. The second component deals with a software structure. It describes the structure of software at a certain time in the software development process and is represented in a dedicated part of the eCST. The last entity deals with software *metrics* and provides a mechanism for the software quality analysis. Similar to a structure definition,

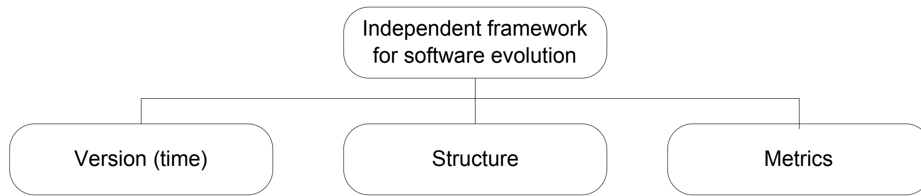

Fig. 5. Three fundamental components of the framework.

the properties needed for evaluating software metrics are also defined in the special part of eCST. All tree core components are shown in Figure 5.

The version entity is a central part of our framework. It determines the certain state of software in a development life cycle. In order to provide meaningful data to our meta-model and to enable a reasonable analysis in the future, the version entity consists of two elements: *DateTime* for determining the date and time of the software snapshot and *VersionName*, which describes the version with a unique identifier (e.g. version number, release name). The latter is usually specified by the product owner (e.g. "my software version 1.5.3").

4.1. Programming language independent meta-model for describing software structure

In order to detect structural software changes between software versions, a special meta-model is needed. However, one of the purposes of this study was to build a programming language independent meta-model that ensures sufficient data and represents the basis for approaches that deal with techniques for detecting structural software changes.

The basis for defining this meta-model were changes defined by Fowler et al. [18]. The authors actually defined refactoring techniques that are similar to source code changes. By their definition, refactoring improves the internal structure of a software system via source code changes. On the other hand, new functionality is not allowed to be added to the end system during the refactoring process [18]. However, each refactoring is a source code change while the opposite relation is not true. Our programming language independent meta-model is defined to provide sufficient data for detecting the changes below.

- Add parameter, field and method
- Remove parameter, field and method
- Hide and unhide method
- Rename method
- Move attribute, method and class
- Extract superclass, interface and class
- Pull up field and method
- Push down field and method
- Inline class

Programming languages differ from each other. Besides object-oriented constructs that are similar, they also have some that are unique or different between languages. For example, Java and C# use properties. In Java, they are implemented with get and set methods. On the other hand, the C# programming language has a unique construct for the same functionality. However, the idea behind a source code representation is to take a snapshot of the software's structure as it is. No additional logic is used that could identify, for example, properties (i.e. getter and setter methods) in Java code. In order to cover as many programming languages as possible, additional changes have been added to the list above. Additional types are written bellow.

- Add property
- Remove property
- Move property
- Pull up property
- Push down property
- Method body change

In order to provide sufficient data for approaches that deal with identifying code changes, the appropriate extent of data should be extracted from the raw source files. This extent of data is called the information level and represents the minimal amount of data that is necessary in order to identify structural changes. However, changes from the list were analyzed and for each change an information level for detecting it from two sequential versions were defined.

For example, figure 6 shows the extract interface change type. In the *version 1*, the class *Employee* has 3 methods: *getRate, getName and getSurname*. After the change process in *version 2*, the *Employee* class implements a new interface *Billable*. The new interface contains a method *getRate* that was "transferred" from the *Employee* class.

To be able to automatically detect such changes from software history, the appropriate data (i.e. information level) should be stored into the meta-model. The information level for the extract interface type is shown in table 3.

Table 3. Information level for extract interface change detection.

	Version 1	Version 2	State (Added/Deleted/Updated)
Package	x	x	x
Class	x	x	x
Base class	-	-	-
Interface	x	x	x
Method	x	x	x
Properties	-	-	-
Attributes	-	-	-

The adequate information for detecting the extract interface change type between *version 1* and *version 2* are *package*, *class*, *interface* and *method*.

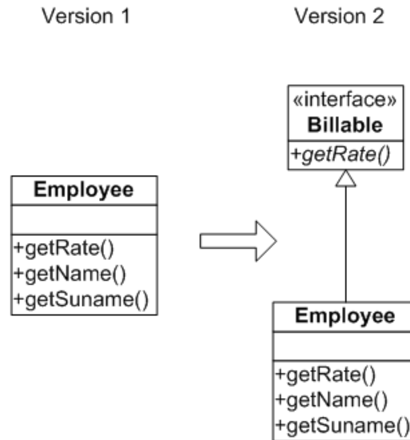

Fig. 6. The extract interface change type.

In addition, the information about the state of this construct is also needed. The state defines if a construct was added, deleted or updated in an observed version. This information is needed in order to identify parts that were actually changed.

The information level needed to fulfill all demands of our list of changes are shown in the table 3. The root element is *concrete_unit_decl* which represents the main entity. From an object-oriented perspective, this element corresponds to the *class* construct. The next important elements are *unit_state*, *name* and *package_decl*. The first one describes the state of an entity in the observed version and identifies if it was added, deleted or updated. For example, if a method is added to an existing class in a version, the *unit_state* node will be set to *updated*. On the other hand, the latter two elements identify a name and a package for a *concrete_unit_decl*. *Extended_base_units* and *implemented_interface_units* represents the lists of extended classes and implemented interfaces of the observed entity.

An additional set of meta-data elements are *attribute_decl*, *property_decl* and *function_decl*. The first two elements represent the attributes and properties of a class. They are composed by two sub-elements that identify their types (*type*) and names (*name*). The last element is *function_decl*, which represents the functions or methods. It consists of the function name (*name*), the access element (*access_decl*), the return type that is represented by *type*, a list of parameters (*parameters_decl*) with underlying elements (*type*, *name*) and tokens that describes the function body. However, tokens are represented in the eCST and therefore they are extracted from it.

Interfaces are described with the *interface_unit_decl* node, which is similar to *concrete_unit_decl*. However, the interface declaration has less universal nodes.

4.2. Software repository for storing framework meta-data

A software repository is a fundamental tool for analyzing software evolution. Software products are represented by raw source code that has to be reshaped in order to achieve a deeper analysis. Special techniques are required to extract meta-data from software products and store them into central storage. In this research, a special software repository was built. Its intent is to fully support the whole process for analyzing the software structure and evaluation of software metrics. In addition to this, a special mechanism for evaluating software quality has been added. The mechanism based on a composed metric, called the Quality index [25].

The repository consists of several modules (Figure 7).

- Basic metrics list
- Composed metrics list
- Defining a Quality index
- Data import
- Metric values presentation through versions

The fundamental module of our repository is a *basic metric list*. Its role is to mark each metric with a unique internal identifier. If the metric is not defined, it is skipped during the data import process. The reason for this is to unify metric names across the repository and to ease further analysis. The next module is *composed metrics list* where custom metrics are defined. For example, C is a composed metric derived from A and B (basic metrics). Furthermore, the repository enables custom calculations using simple mathematical operations. Currently, the repository supports all basic arithmetic operations (e.g. addition, subtraction, multiplication, divisions). However, if we would like to calculate the ratio between the CC and the LOC, then we can manually define the ratio metric as follows: Ratio = CC/LOC. The repository takes all the necessary metrics' data from out of storage and then applies mathematical operations on them. Beside the composed metrics, the repository supports calculating a quality index and its underlying parameters.

The data import process transfers data from outer sources into the repository. The process uses a special mechanism that transforms the original structure of a source into the internal (xml notation). For example, meta-data (represented in meta-models) goes through the process of reshaping its structure in order to import the data into the repository. On the other hand, the repository also allows for the importing of data from third-party sources. The only requirement is to provide the data in an appropriate structure. However, such a procedure guarantees that the data is always imported in the same way.

The last module is responsible for visualizing metric values. The repository supports the storage of software metrics for different project versions. Thus, the metric values could be historically analyzed and shown on a graph. With such a representation, researchers and project managers can analyze the history of the metrics and observe their changes between project versions. For example, if the rise of the cyclomatic complexity is recognized, additional actions may be required to lower the complexity in the next version.

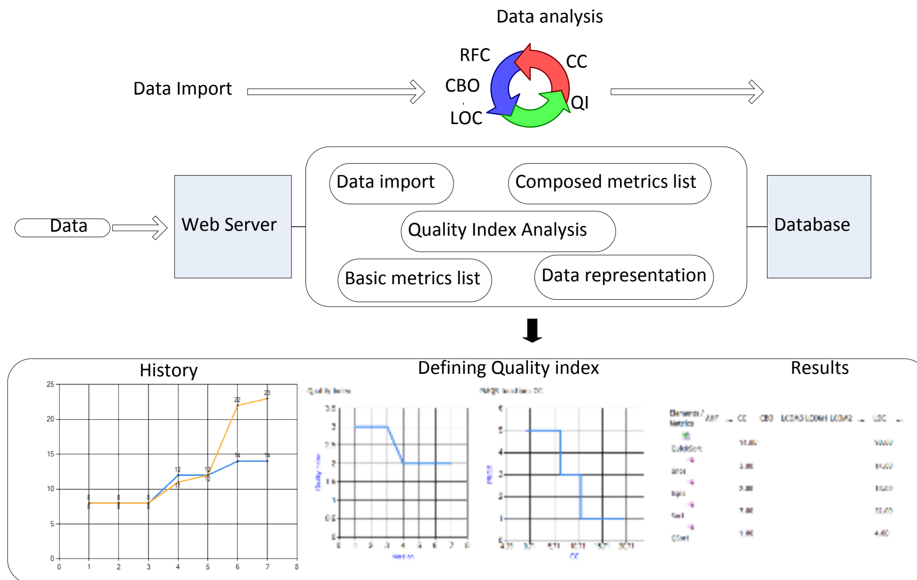

Fig. 7. The metrics repository framework.

5. A case study

In this section, the applicability of the framework is shown. The application is divided into two parts. In the first part, the programming-language independent mechanism for extracting meta-data from raw source code files is described. The mechanism is based on the meta-model that represents a software structure. In the second part, a general approach for calculating software metrics is presented. However, the meta-model with universal nodes is the frame for the software metrics' evaluation.

5.1. Extracting meta-data from a source code

In the case study, two object-oriented programming languages (Java and C#) were used in order to show the applicability of the meta-model for representing software structure. To extract data from raw source code and to fulfill demands of the meta-model, a special tool was developed. The fundamental part of the tool is a mechanism for analyzing source code files. It uses the ANTLR language tool[9] for language recognition and manipulation. However, Java and C# grammars were used in order to construct an abstract syntax tree from the source code. Then, a tree is used to identify meta-data and to fulfill demands of the meta-model.

The example below shows two classes implemented in C# and Java programming language. Both classes (*Student*) have the same behavior and the only distinction is the programming language syntax.

```
/* C# example */
namespace CSharpExample
{
  using System;
  public class Student : Person, IStudent
  {
    private Mark _mark;
    public int StudentNumber { get; set; }
    private decimal CalculateAverageMark(int level){
    ...
    }
  }
  ...
}
```

```
/* Java example */
package JavaExample;
import java.util.ArrayList;
public class Student extends Person implements IStudent
{
  private Mark _mark;
  private int studentNumber;
  public int getStudentNumber(){
  ...
  }
  public void setStudentNumber(int studentNumber){
  ...
  }
  private decimal calculateAverageMark(int level){
  ...
  }
}
...
```

The difference between the classes is the implementation of the *student number* property. In the C# programming language, a special construct is used in order to describe the property. On the other hand, in the Java programming language, the property is implemented with the attribute *studentNumber*, which actually stores a value, and two additional methods. The first method is *getStudentNumber* that returns the value and the second method is *setStudentNumber* that sets the value.

The table 4 shows the meta-models for the extracted source files. However, the meta-models' elements are similar for both languages. The *unit_state* elements were set to 'added' because the analyzed classes were treated as new

Table 4. The meta-model for classes implemented in C# and Java.

	C# class	Java class	is eCST node
UNIT_STATE	Added	Added	no
PACKAGE_DECL	CSharpExample	JavaExample	yes
CONCRETE_UNIT_DECL	x	x	yes
NAME	Student	Student	yes
EXTENDED_BASE_UNITS	Person	Person	yes
IMPLEMENTED_ _INTERFACE_UNITS	IStudent	IStudent	yes
ATTRIBUTE_DECL	x	x	yes
- TYPE	Mark	Mark/ int	yes
- NAME	_mark	_mark/ _studentNumber	yes
PROPERTY_DECL	x	/	yes
- TYPE	int	/	yes
- NAME	StudentNumber	/	yes
FUNCTION_DECL	x	x	yes
- NAME	CalculateAverageMark	CalculateAverageMark/ getStudentNumber/ setStudentNumber	yes
- ACCESS_DECL	private	private/ public/ public	yes
- RETURN_TYPE	x	x	yes
- - TYPE	decimal	decimal/ int/ void	yes
- PARAMETERS_DECL	x	x	yes
- - TYPE	int	int/ - / int	yes
- - NAME	level	level/ - / studentName	yes
- TOKENS	...	... / ... / ...	no

classes in the case study. Similar to previous elements, the *name* elements also have the same value for both languages. On the other hand, the *package_decl* elements are different and are set to 'CSharpExample' and 'JavaExample'. The extended class (*base_unit_decl*) and implemented interface (*interface_unit_decl*) have the same values for both meta-models ('Person' and 'IStudent'). Both classes have one attribute and therefore the *attribute_decl* elements are set with the name '_mark' and type 'Mark'. As expected, the differences are in the definition of the *property_decl* and *function_decl* elements. The C# programming language has special constructs for properties. Therefore the *property_decl* element is set to 'int' for the *type* and 'StudentNumber' for the *name* sub-element. In Java, one additional *attribute_decl* and two additional *function_decl* are defined. Besides the 'getStudentNumber' and 'setStudentNumber' definitions in *function_decl*, the 'calculateAverageMark' method is also defined. It has the 'private' access modifier (*access_decl*), void return type (*return_type*) and one method parameter (*parameter_decl*) with the type 'int' and name 'level'. A similar method is also defined in the meta-model that is the basis of the C# programming language.

The difference between the languages related to properties are reflected in the calculated values of the LOC metric. Therefore, the source code written in Java is approximately twice as long as one written in C# (tables 5 and 6). Furthermore, the properties in C# account for the difference in the number of attributes, properties and methods (the so-called functions in the universal model). These values are presented in table 6. The number of classes (i.e. concrete units) and interfaces (i.e. interface units) are the same in both examples 5. In this case study, the CC metric has no importance because the CC values for all methods are equal to one.

Software metrics have the same relation between different programming languages. For example, the CC metric counts the branches in a source code. If the source code consists of more branches, its complexity will be higher. Therefore, a high complexity will always be indicated with a high CC value in all programming languages and the opposite relation does not exist.

Table 5. Metrics related to the workspaces.

	C#	Java
Number of concrete units	2	2
Number of interface units	1	1
LOC	32	66

When the source files are analyzed, the meta-models and the results are exported in an xml format. Then, only the final step is required. This step imports the prepared meta-models into the software repository.

Table 6. Metrics related to the class Student.

class Student	C#	Java
Number of attributes	1	2
Number of properties	1	0
Number of functions	1	3
LOC	15	22

6. Validity and limitations

This section is focused on an internal and external validity [28] and on the limitations of the study. In order to guarantee the accuracy of the extracted data from the raw source files, the ANTLR language tool was used. Furthermore, the C# and Java grammar, which describes a programming language, were used. The tool requires a syntactically correct source code in order to build an abstract syntax tree. However, the tree constitutes the basis for extracting the meta-data into the meta-models used by framework.

The metrics' value calculations rely on the nodes defined in the abstract syntax tree. Therefore, the metrics' equations are unified and defined only once in the higher level of abstraction. For example, if the metric for calculating cyclomatic complexity is defined upon the universal nodes in a syntax tree then their values could be easily calculated for C# and Java programming language without knowing the specifics of the programming language. Furthermore, such an approach enables the calculation of software metrics in the same way between programming languages. Thus, comparing the quality of heterogeneous systems in such an environment is more accurate.

In this research, a programming language independent framework for analyzing software evolution was successfully applied in a case study and partly in the preliminary work. In the first part, the source code of two programming languages (i.e. C# and Java) were used in order to analyze its structure. The results showed that the meta-model for describing software structure was successfully populated with the meta-data. In the second part, the SMIILE tool that evaluates software metrics was used. Several software metrics were defined using the eCST and tested with different programming languages.

The metric values were correctly calculated for all cases. This was proven by using independent and language-specific software metrics tools.

The research showed that the framework is general (i.e. programming language independent) and can be used for more programming languages.

The limitation of the framework is the extent of meta-data that are described in the meta-model for representing software structures. The study has been limited to the subset of changes defined in Fowler's book [18]. However, the book contains numerous refactorings for resolving bad smells in code and design. From this perspective, this limitation is not a real limitation. Furthermore, software metrics' support gives additional value to the framework because the quality of changes can be tracked.

Different structural source code changes cannot always be compared among programming languages. Some of them have special object-oriented constructs (e.g. properties in C#) or behaviour (e.g. some languages support multiple inheritance). Another limitation is that the LOC metric has to be compared with caution. For example, a high LOC value in a class (e.g. 10,000 lines of code) will always indicate a large class and vice versa. On the other hand, we cannot predict that the LOC values of the classes with the same behaviour, written in C# and Java, will be equal. However, even if we can not strictly compare every aspect among languages, we still provide consistent monitoring of software evolution. Furthermore, adding weights to some metric values (e.g. the LOC metric) can lead to better comparability among languages. However, such an approach can improve the framework in the future.

7. Related work

In the last decade, different approaches for evaluating software artifacts have been used. Therefore, this section will focus on existing tools and approaches for evaluating software metrics and software structure through the software development process. As the analisys of related work will show, the integrated approach to application of software metrics algorithms and analysis of software evolution that are independent on programming languages are not existant in usable form. The first part describes the approaches for evaluating software metrics, the second part describes the approaches for representing software structure and the third part is focused on programming language independence.

The majority of problems are related to programming language dependency. However, the last part of the analysis describes some more or less successful approaches in order to overcome this issue. To improve existing approaches, new framework that is based on internal representation of source code with improved characteristics was developed.

7.1. Software metric approaches

In this section, the findings of current problems in the application of software metrics in practice are described [39]. Some preliminary observations of the field show that the main problem lies in the weaknesses of available metric tools and techniques. These observations are based on numerous reports on the weaknesses of existing tools in both practice and in the academic world ([30] and [32]).

Our analysis included 20 tools, with six of them being representative examples. The tools were analyzed with respect to two groups of criteria.

The first group of criteria is related to the usage range of a tool and by the nature and structure of the software product being measured. However, the group of criteria consists of: platform independence, input language independence and a list of supported metrics. The following metrics were considered:

- the cyclomatic complexity - CC,

- the Halstead metrics - H,
- lines of code - LOC (if a tool calculates any of the LOC (SLOC, CLOC, etc.) metric, then the corresponding cell contains the '+' symbol),
- the object-oriented metrics - OO (if a tool supports any of the OO metrics, then the corresponding cell contains the '+' symbol. The mark '*' next to the symbol '+' means that a tool only partially satisfied specified criteria) and
- the others (if a metric is supported and it does not belong on the list above, then the criteria is marked with a '+').

The results for six representative tools can be seen in Table 7.

Table 7. Software metric tools and observed criteria

Tool	Producer [see ref]	Platform indep.	Language indep.	CC	H	LOC	OO	Other metrics
SLOC	D. Wheeler [47]	-	+	-	-	+	-	-
Code Counter Pro	Geronesoft [2]	-	+	-	-	+	-	-
Source Monitor	Campwood Software [8]	-	-	+	-	+	+	-
Understand	ScientificToolworks [1]	+	-	+	+	+	-	-
RSM	MSquared Technologies [7]	+	-	+	+	+	-	-
Krakatau	Power Software [5], [6]	-	+*	+	+	+	+	-

The important conclusions of this analysis are below.

- The analyzed tools could be divided into two categories.
  - The first category includes tools that only calculate simple metrics (i.e. the LOC metrics) but for a wide set of programming languages.
  - The second category of tools is characterized by a wide range of metrics but limited to a small set of programming languages. There were attempts to bridge the gap between these categories, but without success. This is a limitation because there are many legacy software systems written in ancient languages, whereas modern metric tools cannot be applied uniformly.
- Even if the tools support some object-oriented metrics, the amount of supported metrics is fairly small. This is especially true when compared to the broad application of the object-oriented approach within current software development.

However, we have demonstrated on representative languages that the SMIILE is language independent for currently implemented software metrics (section 3.3). The process of calculating them can be strictly connected with the language syntax (e.g. the CC metric) or it can be less sensitive to its syntax and lexical analysis because we have enough data in the universal nodes (e.g. the LOC metric with the first and the last line). The object-oriented metrics are still decently supported in tools. However, we have an internal representation of the source code and its design. This is the basis for metric calculation and our next task is to extend the set of algorithms for calculating software metrics[38].

Furthermore, the analysis considered support for processing and interpreting the calculated metric results via the given tools. The criteria were: the history of the source code, the metric results' storing facility, a graphical representation of the calculated values and an interpretation of the calculated values including suggestions for improvements based on the calculated values. The general conclusion was that many techniques and tools compute numerical results with no real interpretation of their meaning. The only interpretations of numerical results that can be found are graphical. These results possess little or no value for practitioners, who need suggestions or advice on how to improve their project based on the metrics' results. Recommendations for an improvement, or even the automation of an improvement based on the obtained metrics results, would be significantly useful for the way to the real practical usability of software metrics.

Today, complex software projects are developed in several programming languages while available software metric tools are not language independent. When taking these facts into account, we can conclude that the use of several software metric tools in one project is required. An additional problem is that different software tools often provide different values for the same metric, calculated on the same product or its component [36],[32]. One of the reasons for this is the fact that the rule for metrics calculations could be differently interpreted and implemented with different tools [43]. On the other hand, our approach uses a common internal representation of the source code and meta-model for all programming languages that represents a basis for metrics calculation. Such an approach enables the same metrics calculation algorithms across different programming languages.

7.2. Approaches for analyzing software structure

Software evolution analysis covers different aspects of software development. ¿From the granularity level, two major approaches exists. The version-centered approach considers versions to be a representation of granularity, while the history-centered approach considers history to be a representation of granularity [21].

The research conducted by Gîrba et al. [21] focused on the set of requirements that an evolution meta-model should have. Therefore, they defined a meta-model where history is modeled as an explicit entity. A time component was set as the basis for structural information, which thus provides a common

infrastructure for expressing and combining evaluation analysis and structural analysis. The authors also focused on different abstraction and detailed levels, the ability to compare and combine property evolutions and the ability of history navigation between relations. Our meta-model differs from this study in three aspects: the meta-model is programming-language independent (supports object-oriented and procedural languages), it has the ability to represent software metrics and it provides a sufficient basis for detecting structural code changes between versions.

Tichelaar et al. [45] investigated similarities between refactorings for Smalltalk and Java programming languages. They derived a language independent meta-model for object-oriented source code and showed that it is feasible to build a language independent engine for refactorings on top of this meta-model. Our study is similar in the context of an independent meta-model and differs in the ability to provide sufficient data to analyze different structural source code changes between versions over the software evolution. However, some refactorings are composed by one or several structural source code changes.

Studies conducted in [21, 45] are based on a language independent and extensible model for modeling object oriented software systems, called FAMIX [44].

7.3. Programming language independence

This section focuses on various universal software tools that strive to achieve the independence of an input programming language.

The FAMIX meta-model [44] boasts one of the most similar general goals with our project. Its strength is mainly in language independence. It supports OO design (at the interface level of abstraction) for various input programming languages and is supported by separate tools for filling in the meta-model with sources in different programming languages. Our approach is more general - it is based on (enriched) syntax trees representing all aspects of source code, instead of just the design. It is thus equally appropriate for supporting a broader set of software metrics. However, it also fully supports procedural languages, including legacy ones (e.g., COBOL).

Arbuckle[11] presented an interesting approach for the measuring evolution of a multi-language software system. He avoids difficulties related to syntax, semantics and language paradigms by looking directly at relative shared information content. His approach measures a relative number of bits of shared binary information between artifacts of consecutive releases. However, our approach uses source code changes from software repositories to analyze software evolution.

The ATHENA tool for assessing the quality of software [15] was based on the parsers that generate abstract syntax trees as a representation of a source code. The generated trees were structured in such a way that the metric algorithms were easily applied. The final goal of the tool was to generate a report that describes the quality. However, it was only executable under the UNIX operating system and its official support is not available anymore. Our approach

is also based on usage of specific parser for generating of syntax trees, but our parers are not manually implemented but generated by a parser generator. This makes the process of adding support for a new programming language easier. Furthermore, the eCST has a richer representation than AST. At the end, the SMIILE tool is implemented in Java and can therefore be used on a broader range of platforms.

The development of the CodeSquale metrics tool was based on a similar idea. The early results were published on the project website [3], [4]. The authors developed a system, based on the representation of a source code through abstract syntax trees, and implemented one object-oriented metric for the Java source code. Furthermore, an idea for the additional implementation of other metrics and opportunities for extending the tree to other programming languages was described. However, their final goal was programming language independence. Unfortunately, later results were not published. However, a weakness of the project was the use of an AST to represent source code. By using the eCST we are able to implement algorithms that are independent of programming language.

The Wide Spectrum Language (WSL) [10] is used for the intermediate representation of software programs in translating legacy to maintainable code (eg. assembly code to C/COBOL code). The main characteristics of WSL is a formal background and the application of formal transformations of code internally represented by using abstract syntax trees. Even the WSL is (by definition) independent of programming languages. Nevertheless, it still does not support object-oriented languages. In the process of program transformation, a small set of software metrics is used to measure the effects of transformations. In comparison with WSL, our approach supported a broader scope of languages and metrics (including object-oriented).

Static analysis usually includes some metrics calculation and further analysis of the obtained values. Such a study was presented in [46] where the authors used a static analysis for student programs written in Java. The study is based on an abstract syntax tree (AST) to represent the code. The XML format was used in order to represent the data.

The AST representation of the source code also led to language independence in some related areas of software engineering. The tool described in [16] uses the abstract syntax tree for the representation of source code in a duplicated code analysis. The tree has specific mechanisms designed for the easier implementation of algorithms and comparisons. A similar approach was described in [13] but a more complex algorithm for comparison was implemented.

An approach for detecting similar classes in Java source code was presented in [42]. Furthermore, ASTs were also used to monitor software changes [35]. The specified tool was implemented for the analysis of code written in the C programming language. Its significance is in its ideas for change analysis based on AST.

8. Conclusion and future work

The programming language independent framework for analyzing software structure and metrics during software development and evolution was presented in the study. The framework consists of three modules. The first one defines the meta-model for providing sufficient data, which constitutes the basis for approaches dealing with software change detection processes. The second module uses a mechanism for evaluating software metrics. Both modules are built on the eCST for the unified representation of a source code. The last module contains an approach for collecting evolutionary software artifacts that then enables further analysis.

The integration of improved characteristics of eCST into a framework for metrics calculation and the framework for software evolution extended by software metrics and the changes repository lead to the following important benefits.

- Usage of the eCST leads to language independence.
- The storing of software metrics in the software metrics repository enables a better interpretation of acquired data.
- Integration with the repository additionally gives opportunities to extend it in such a way as to store data about structural software changes.
- Enables the further analysis of stored data (e.g. custom metrics) and provides the opportunity to give recommendations to users about the improvement of a product and the development process or even the automation of some of the suggested improvements (e.g. automatic refactoring). The need for this analysis is examined in the related work. An advanced calculation on metric values and visualization are enabled by the software metrics repository and the rest of an intelligent analysis are planned for future work.

These features distinguish the framework from existing techniques and approaches and provide it with significant prospects in the field of software development and evolution. Furthermore, a special engine for detecting structural source code changes is already being implemented, but it is out of the scope of this paper.

The framework was successfully presented in the case study. In the first part, the software structure was analyzed from source code written in two different programming languages (i.e. C# and Java). The data extracted from the eCST into the meta-model fulfilled all described requirements. However, programming language independency of the eCST has been also shown on other case studies [39] and [40]. Therefore, the extraction is independent of an input language. In the second part, several software metrics were evaluated based on the same source code. The algorithms defined upon the universal nodes correctly calculated the values for *lines of code* and *cyclomatic complexity* and also for some other design metrics (i.e. *NOC, NOA, NOM*). In the last part, the results were successfully imported into the software repository for collecting and storing meta-data.

The framework and its components are still on prototype level. In future work, the framework will be tested with more programming languages and adjustments will be made if necessary. Also, support for additional algorithms for calculating software metrics will be added. Furthermore, all data will be stored in a repository with the intention of analyzing the correlation between the changes and software quality and to provide more useful information to the user or even to develop an automated quality improvement.

Acknowledgments. Work of the second and third author is partially supported by the Serbian Ministry of Science and Technological Development through project no. OI174023 "Intelligent Techniques and Their Integration into Wide-Spectrum Decision Support". Bilateral project between Slovenian Research Agency and Serbian Ministry of Science and Technological Development (Grant BI-SR/10-11-027) enabled the exchange of visits and ideas between authors of this paper and their institutions.

References

1. Understand 2.0 user guide and reference manual. online (2008), http://www.scitools.com
2. Code counter pro. online (2010), http://www.geronesoft.com/
3. Codesquale. online (2010), http://code.google.com/p/codesquale/
4. Codesquale. online (2010), http://codesquale.googlepages.com/
5. Krakatau essential pm (kepm)- user guide 1.11.0.0. online (2010), http://www.powersoftware.com/
6. Krakatau suite management overview. online (2010), http://www.powersoftware.com/
7. Rsm. online (2010), http://msquaredtechnologies.com/
8. Sourcemonitor,. online (2010), http://www.campwoodsw.com/sourcemonitor.html
9. Antlr - another tool for language recognition (2011), http://www.antlr.org
10. Wsl - wide spectrum language (2012), http://www.smltd.com/wsl.htm
11. Arbuckle, T.: Measuring multi-language software evolution: a case study pp. 91–95 (2011)
12. Basili, V.R.: Data collection, validation and analysis, p. 143160. MIT Press (1981)
13. Baxter, I.D., Yahin, A., de Moura, L.M., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: ICSM. pp. 368–377 (1998)
14. Breivold, H.P., Crnkovic, I., Larsson, M.: A systematic review of software architecture evolution research. Information and Software Technology 54(1), 16 – 40 (2012)
15. Christodoulakis, D., Tsalidis, C., van Gogh, C., Stinesen, V.: Towards an automated tool for software certification. In: Tools for Artificial Intelligence, 1989. , IEEE International Workshop on Architectures, Languages and Algorithms. pp. 670–676 (1989)
16. Ducasse, S., Rieger, M., Demeyer, S.: A language independent approach for detecting duplicated code. In: ICSM. pp. 109–118 (1999)
17. Fenton, N.E., Neil, M.: Software metrics: successes, failures, and new directions. Journal of Systems and Software (1999)
18. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1 edn. (Jul 1999)
19. Črt Gerlec, Andrej Krajnc, M.H.J.B.: Mining source code changes from software repositories. Central and Eastern European Software Engineering Conference in Russia (2011)

20. Gilb, T.: Software Metrics. Chartwell-Bratt (1976)
21. Gîrba, T., Ducasse, S.: Modeling history to analyze software evolution: Research articles. J. Softw. Maint. Evol. 18, 207–236 (May 2006)
22. Goeminne, M., Mens, T.: A comparison of identity merge algorithms for software repositories. Science of Computer Programming (2011)
23. Grune, D., Bal, H.E., Jacobs, C.J.H., Langendoen, K.: Modern Compiler Design. John Wiley (2002)
24. Harrison, W.: A flexible method for maintaining software metrics data: a universal metrics repository. Journal of Systems and Software 72(2), 225–234 (2004)
25. Heričko, M., Živkovič, A., Porkolb, Z.: A method for calculating acknowledged project effort using a quality index. Informatica 31(4), 431–436 (2007)
26. Illes-Seifert, T., Paech, B.: Exploring the relationship of a files history and its fault-proneness: An empirical method and its application to open source programs. Information and Software Technology 52(5), 539 – 558 (2010)
27. Institute, C.M.U.S.E., Martin, R., Carey, S., Coticchia, M., Fowler, P., Maher, J.: Proceedings of the Workshop on Executive Software Issues, August 2-3 and November 18, 1988. Technical report, Carnegie Mellon University, Software Engineering Institute (1989)
28. Jedlitschka, A., Ciolkowski, M., Pfahl, D.: Reporting experiments in software engineering. Guide to advanced empirical software engineering (1), 201–228 (2008)
29. Kan, S.: Metrics and Models in Software Quality Engineering Second Edition Boston. Addison-Wesley (2003)
30. Lanza, M., Marinescu, R.: Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer (2006)
31. Lehman, M., Ramil, J.F., Kahen, G.: Evolution as a noun and evolution as a verb. In: Proc. Workshop on Software and Organisation Co-evolution (SOCE) (July 2000)
32. Lincke, R., Lundberg, J., Löwe, W.: Comparing software metrics tools. In: ISSTA. pp. 131–142 (2008)
33. Madhavji, N.H., Fernandez-Ramil, J., Perry, D.: Software Evolution and Feedback: Theory and Practice. John Wiley & Sons (2006)
34. N. Fenton, S.L.P.: Software Metrics: A Rigorous and Practical Approach. Thomson Computer Press (1996)
35. Neamtiu, I., Foster, J.S., Hicks, M.W.: Understanding source code evolution using abstract syntax tree matching. In: MSR (2005)
36. Novak, J., Rakić, G.: Comparison of software metrics tools for :net. In: Proc. of 13th International Multiconference Information Society - IS, Vol A. pp. 231–234 (2010)
37. Pfleeger, S.: Lessons learned in building a corporate metrics program. Software, IEEE 10(3), 67 –74 (may 1993)
38. Rakić, G., Budimac, Z.: Introducing enriched concrete syntax trees. In: Proc. of 13th International Multiconference Information Society - IS, Vol A. pp. 211–214 (2011)
39. Rakić, G., Budimac, Z.: Problems in systematic application of software metrics and possible solution. In: Proc. of The 5th International Conference on Information Technology (ICIT) (2010)
40. Rakić, G., Budimac, Z.: Smiile prototype. AIP Conference Proceedings 1389(1), 853–856 (2011)
41. Rochkind, M.J.: The source code control system. IEEE Transactions on Software Engineering 1(4), 364–370 (1975)
42. Sager, T., Bernstein, A., Pinzger, M., Kiefer, C.: Detecting similar java classes using tree algorithms. In: MSR. pp. 65–71 (2006)

43. Scotto, M., Sillitti, A., Succi, G., Vernazza, T.: A non-invasive approach to product metrics collection. Journal of Systems Architecture 52(11), 668–675 (2006)
44. Tichelaar, S., Ducasse, S., Demeyer, S.: Famix and xmi. In: Reverse Engineering, 2000. Proceedings. Seventh Working Conference on. pp. 296 –298 (2000)
45. Tichelaar, S., Ducasse, S., Demeyer, S., Nierstrasz, O.: A meta-model for language-independent refactoring. In: Principles of Software Evolution, 2000. Proceedings. International Symposium on. pp. 154 –164 (2000)
46. Truong, N., Roe, P., Bancroft, P.: Static analysis of students' java programs. In: ACE. pp. 317–325 (2004)
47. Wheeler, D.A.: Sloccount user's guide, version 2.26. online (2004), http://www.dwheeler.com/sloccount/sloccount.html

Appendix

Table 8. Catalog of universal nodes.

Universal node	coresponding element of language syntax
PACKAGE_DECL	package, workspace, etc
CONCRETE_UNIT_DECL	class, implementation module, etc
ABSTRACT_UNIT_DECL	abstrat class, etc
INTERFACE_UNIT_DECL	interface, definition module, etc
EXTENDED_BASE_UNITS	extended class
IMPLEMENTED_INTERFACE_UNITS	implemented interface, corresponding definition module, etc.
INSTANTIATED_UNIT	instantionation of a new object
IMPORT_DECL	unit or function import
ATTRIBUTE_DECL	attribute, field, etc.
PROPERTY_DECL	property
FUNCTION_DECL	method, procedure, function, etc
FUNCTION_CALL	call of a function
PARAMETERS_DECL	parameters of a function
ARGUMENT_LIST	parameters passed to a function
VAR_DECL	local variable defined in functions
MAIN_BLOCK	main block of program
STATEMENT	Any statement
BRANCH_STATEMENT	Any Branch Statement (each branch will be additionally marked)
BRANCH	Branch in Branch Statement
LOOP_STATEMENT	Any Loop Statement
JUMP_STATEMENT	Any Jump Statement
CONDITION	Condition (in loop, branch,. . . statements)
CONDITION_BRANCH	each branch of condition separated by logical operator
LOGICAL_OPERATOR	logical operator (in condition)
OPERATOR	any operator
OPERAND	any operand
NAME	name of any element (unit, function,etc.)
TYPE	type of any element (unit, function,etc.)

Črt Gerlec is a researcher and PhD student associated with the Faculty of Electrical Engineering and Computer Science, Institute of Informatics at the University of Maribor. His research interests are mining software repositories, software evolution, software quality, software metrics, information systems and

more. He is experienced software developer on Microsoft.NET platform and expert for software architecture, design patterns and best practices.

Gordana Rakić has received her MSc degree in 2010 from Faculty of Sciences, University of Novi Sad. Currently she is the PhD student and works as assistant at Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad. Her fields of interest are Software Engineering, Software Metrics, Software Maintenance, etc.

Zoran Budimac Since 2004 holds position of full professor at Faculty of Sciences, University of Novi Sad, Serbia. Currently, he is head of Computing laboratory. His fields of research interests involve: Educational Technologies, Agents and WFMS, Case-Based Reasoning, Programming Languages. He was principal investigator of more then 20 projects. He is author of 13 textbooks and more then 220 research papers most of which are published in international journals and international conferences. He is/was a member of Program Committees of more then 60 international Conferences and is member of Editorial Board of Computer Science and Information Systems Journal.

Marjan Heričko is a Full Professor at the University of Maribor, Faculty of EE&CS, Institute of Informatics. He received his M.Sc. (1993) and Ph.D. (1998) in computer science from the University of Maribor. His research interests include all aspects of IS development with emphasis on metrics, software patterns, process models and modeling.

Received: January 04, 2012; Accepted: May 31, 2012.