UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCES
DEPARTMENT OF
MATHEMATICS AND INFORMATICS

Gordana Rakić

# Extendable and Adaptable Framework for Input Language Independent Static Analysis

- PhD Thesis -

Supervisor: Dr Zoran Budimac

Novi Sad, June 2015

# Contents

**Abstract** **xii**

**Sažetak** **xv**

**Preface** **xvii**

**1 Introduction** **1**

1.1 Outline . . . . . . . . . . 4

**2 Background** **7**

2.1 Software quality - standards and definitions . . . . . . . . . . 7

2.2 Static analysis . . . . . . . . . . 8

2.2.1 Intermediate representations in static analysis . . . . . . . . . . 10

2.2.2 Representative static analysis techniques . . . . . . . . . . 13

2.2.2.1 Software metrics . . . . . . . . . . 13

2.2.2.2 Software networks analysis . . . . . . . . . . 21

2.2.2.3 Code clone detection . . . . . . . . . . 23

2.2.2.4 Software changes analysis . . . . . . . . . . 26

2.3 Computer languages . . . . . . . . . . 27

2.3.1 General purpose languages . . . . . . . . . . 29

2.3.1.1 Procedural languages . . . . . . . . . . 29

2.3.1.2 Object-oriented languages . . . . . . . . . . 32

2.3.1.3 Functional languages . . . . . . . . . . 33

2.3.1.4 Mixed-paradigm langauges . . . . . . . . . . 34

2.3.2 Domain-specific languages . . . . . . . . . . 35

2.3.2.1 Specification languages . . . . . . . . . . 36

2.3.2.2 Modeling languages . . . . . . . . . . 37

2.3.3 Script languages . . . . . . . . . . 41

2.3.4 Legacy languages . . . . . . . . . . 42

2.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

**3 Justification of SSQSA Concept 45**

3.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
3.2 Preliminary investigation . . . . . . . . . . . . . . . . . . . . . . . 46
3.3 Intermediate representations . . . . . . . . . . . . . . . . . . . . . 52
3.4 Related software solutions . . . . . . . . . . . . . . . . . . . . . . 57
3.5 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

**4 SSQSA framework 63**

4.1 SSQSA overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63
4.2 SSQSA Internal source code representation . . . . . . . . . . . . . 69
4.2.1 eCST: enriched Concrete Syntax Tree . . . . . . . . . . . . . . 69
4.2.1.1 Fundamental concept of universal nodes . . . . . . . . . . 75
4.2.1.2 Methodology . . . . . . . . . . . . . . . . . . . . . . . . 77
4.2.1.3 Overview of universal nodes . . . . . . . . . . . . . . . . 79
4.2.2 Internal Representations Derived from eCST . . . . . . . . . . 82
4.2.2.1 eCFG: enriched Control-Flow Graph . . . . . . . . . . . . 82
4.2.2.2 eGDN: enriched General Dependency Network . . . . . . 84
4.3 SSQSA components . . . . . . . . . . . . . . . . . . . . . . . . . . 87
4.3.1 Generators and manipulators of SSQSA internal representations . 87
4.3.1.1 eCST Generator: enriched Concrete Syntax Tree Generator 87
4.3.1.2 Generators of the internal representation derived from eCST 91
4.3.1.3 eCST Manipulators . . . . . . . . . . . . . . . . . . . . . 94
4.3.2 SSQSA static analysers . . . . . . . . . . . . . . . . . . . . . . 97
4.3.2.1 SMIILE:
Software Metrics Independent on Input LanguagE . . . . 97
4.3.2.2 SNAIPL: Software Networks Analyser Independent of Programing Language . . . . . . . . . . . . . . . . . . . . . . 100
4.3.2.3 SSCA: Software Structure Change Analyzer . . . . . . . . 102
4.3.2.4 LICCA: Language Independent Code Clone Anlysis . . . 103
4.3.3 Peripheral tools integrated in SSQSA framework . . . . . . . . . 105
4.3.3.1 External software metrics repository . . . . . . . . . . . . 105
4.3.3.2 Integration with the Testovid . . . . . . . . . . . . . . . . 105
4.4 Adaptability of SSQSA framework . . . . . . . . . . . . . . . . . . 106
4.4.1 Role of the parser generator . . . . . . . . . . . . . . . . . . . . 110
4.4.2 Demonstration od Adaptability . . . . . . . . . . . . . . . . . . 115
4.4.2.1 Procedural languages . . . . . . . . . . . . . . . . . . . . 115
4.4.2.2 Object-oriented languages . . . . . . . . . . . . . . . . . 117
4.4.2.3 Functional languages . . . . . . . . . . . . . . . . . . . . 118
4.4.2.4 Mixed paradigm languages . . . . . . . . . . . . . . . . . 120
4.4.2.5 Script languages . . . . . . . . . . . . . . . . . . . . . . . 121

4.4.2.6 Legacy languages . . . 122
4.4.2.7 Specification languages . . . 123
4.4.2.8 Intermediate languages . . . 125
4.4.2.9 Domain-Specific languages . . . 126
4.5 Extendability of SSQSA framework . . . 127
4.5.1 Demonstration of extendability . . . 128
4.5.1.1 Syntax-independent analyses . . . 128
4.5.1.2 Syntax-based analyses . . . 131
4.5.1.3 Dependency-based analyses . . . 133
4.6 Summary . . . 134

**5 Validation and results** **137**
5.1 Inspection of SSQSA results . . . 137
5.1.1 Internal consistency check . . . 138
5.1.1.1 Internal validation of eCST Generator . . . 138
5.1.1.2 Internal validation of eGDN Generator . . . 143
5.1.1.3 Internal validation of static analysers . . . 144
5.1.2 External correctness check . . . 147
5.1.2.1 External validation of eGDN Generator . . . 147
5.1.2.2 External validation of SMIILE . . . 149
5.2 SSQSA potentials . . . 157
5.2.1 ATSE and project-oriented teaching . . . 158
5.3 Summary . . . 161

**6 Conclusion** **163**
6.1 Future work . . . 165
6.2 Summary . . . 166

**A Catalog of universal nodes** **167**

**B Student example** **183**

**Prošireni sažetak** **189**

**Bibliography** **199**

**Short biography** **211**

**Kratka biografija** **213**

**Key words documentation** **215**

**Ključna dokumentacijska informacija** **219**

# List of Figures

2.1 Z specification of Stack . . . 36
2.2 Ontology example . . . 40

4.1 SSQSA architecture . . . 65
4.2 SSQSA architecture: pipes and filters . . . 68
4.3 CST, AST, and eCST: comparative view . . . 70
4.4 eCST: comparative view (example in Java and Modula-2) . . . 72
4.5 eCST: *BubbleSort* (Java example) . . . 74
4.6 eCST: comparative view (loop statement example in Java and Modula-2) 76
4.7 eCFG: BubbleSort example . . . 83
4.8 eGDN: comparative view (Part 1: Student example in Java) . . . 84
4.9 eGDN: comparative view (Part 2: Student example in C# . . . 84
4.10 SCG: GTK Scheme example . . . 85
4.11 UCN: Delphi example . . . 86
4.12 eCSTGenerator architecture . . . 88
4.13 Languages XML Schema . . . 89
4.14 eCT XML Schema . . . 90
4.15 eCST to eCFG transformation process . . . 92
4.16 eGDN XML Schema . . . 93
4.17 Software metrics XML Schema . . . 99
4.18 Structural changes XML Schema . . . 103
4.19 Code clones XML Schema . . . 104
4.20 Extendability: steps overview . . . 108
4.21 Enrichment of the syntax tree, part 1: before (while example) . . . 109
4.22 Enrichment of the syntax tree, part 2: after (while example) . . . 109
4.23 eCST: XML fragment view (class declaration example) . . . 129
4.24 eCST: XML fragment view (comment example) . . . 130
4.25 eGDN sub-graph: inheritance tree (SSQSA fragment example) . . . 134

# List of Tables

3.1 Software metrics tools overview: part 1 . . . . . . . . . . . . . . . . . . . . . 48
3.2 Software metrics tools overview: part 2 . . . . . . . . . . . . . . . . . . . . . 50

4.1 High-level eCST universal nodes . . . . . . . . . . . . . . . . . . . . . . . . . 80
4.2 Middle-level eCST universal nodes . . . . . . . . . . . . . . . . . . . . . . . . 81
4.3 Low-level eCST universal nodes . . . . . . . . . . . . . . . . . . . . . . . . . 81
4.4 Parser generators overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . 114
4.5 Halstead metrics results (example)) . . . . . . . . . . . . . . . . . . . . . . . 133

5.1 eCSTGenerator results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 143
5.2 Design metrics (eCST and eGDN - an comparative overview): eGDN Generator example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 144
5.3 CC values: QuickSort example . . . . . . . . . . . . . . . . . . . . . . . . . . 146
5.4 eGDN Generator results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 148
5.5 Network extraction results: comparative overview . . . . . . . . . . . . . . . 149
5.6 Size metrics: eGDN Generator example . . . . . . . . . . . . . . . . . . . . . 150
5.7 Halstead metrics (basic) - comparative overview: eGDN Generator example 152
5.8 Halstead metrics (derived 1) - comparative overview: eGDN Generator example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
5.9 Halstead metrics (derived 2) - comparative overview: eGDN Generator example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
5.10 CC metrics - comparative overview: eGDN Generator example . . . . . . 156

A.1 COMPILATION_UNIT universal node . . . . . . . . . . . . . . . . . . . . . . . 167
A.2 PACKAGE_DECL universal node . . . . . . . . . . . . . . . . . . . . . . . . . 168
A.3 INTERFACE_UNIT_DECL and CONCRETE_UNIT_DECL universal nodes . . . . . 169
A.4 IMPLEMENTS and EXTENDS universal nodes . . . . . . . . . . . . . . . . . . . 170
A.5 IMPORT_DECL universal node . . . . . . . . . . . . . . . . . . . . . . . . . . 171
A.6 TYPE_DECL universal node . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171

A.7 ATTRIBUTE_DECL universal node . . . . . . . . . . . . . . . . . . . . . . . . 172
A.8 BLOCK_SCOPE universal node . . . . . . . . . . . . . . . . . . . . . . . . . . 173
A.9 FUNCTION_DECL universal node . . . . . . . . . . . . . . . . . . . . . . . . . 174
A.10 FORMAL_PARAM_LIST and PARAMETER_DECL universal node . . . . . . . . 174
A.11 FUNCTION_CALL universal node . . . . . . . . . . . . . . . . . . . . . . . . . 175
A.12 ARGUMENT_LIST and ARGUMENT universal nodes . . . . . . . . . . . . . . 175
A.13 VAR_DECL universal node . . . . . . . . . . . . . . . . . . . . . . . . . . . . 176
A.14 STATEMENT universal node . . . . . . . . . . . . . . . . . . . . . . . . . . . 176
A.15 INSTANTIATES universal node . . . . . . . . . . . . . . . . . . . . . . . . . . 177
A.16 LOOP_STATEMENT universal node . . . . . . . . . . . . . . . . . . . . . . . 178
A.17 BRANCH_STATEMENT and BRANCH universal nodes . . . . . . . . . . . . 179
A.18 JUMP_STATEMENT universal node . . . . . . . . . . . . . . . . . . . . . . . 180
A.19 CONDITION universal node . . . . . . . . . . . . . . . . . . . . . . . . . . . 181
A.20 EXPRESSION universal node . . . . . . . . . . . . . . . . . . . . . . . . . . . 181
A.21 KEYWORD, OPERATOR, SEPARATOR, and BUILTIN_TYPE universal nodes . . 181
A.22 NAME and TYPE universal nodes . . . . . . . . . . . . . . . . . . . . . . . . 182
A.23 Universal nodes marking comments . . . . . . . . . . . . . . . . . . . . . . 182

# Listings

2.1 *QuickSort* algorithm implemented in Modula-2 . . . . . . . . . . . . . . . 30
2.2 *ArbitraryList* implemented in Java . . . . . . . . . . . . . . . . . . . . . . 32
2.3 *QuickSort* implemented in Erlang . . . . . . . . . . . . . . . . . . . . . . . 33
2.4 *QuickSort* implemented in functional Scala . . . . . . . . . . . . . . . . . . 34
2.5 *QuickSort* implemented in object-oriented Scala . . . . . . . . . . . . . . . 35
2.6 *QuickSort* specification written in Tempura . . . . . . . . . . . . . . . . . 37
2.7 *Contact* ontology written in OWL notation . . . . . . . . . . . . . . . . . . 37
2.8 *QuickSort* implemented in PHP . . . . . . . . . . . . . . . . . . . . . . . . 41
2.9 *QuickSort* implemented in COBOL . . . . . . . . . . . . . . . . . . . . . . . 42
4.1 *BubbleSort* implemented in Java . . . . . . . . . . . . . . . . . . . . . . . 72
4.2 Dynamic instantiation in Java . . . . . . . . . . . . . . . . . . . . . . . . . 94
4.3 Dynamic types in PHP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 95
4.4 Grammar rule for the WHILE statement . . . . . . . . . . . . . . . . . . . . 107
4.5 Rewriting the rule . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
4.6 Enrichment of the tree . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
4.7 Simple example written in Modula-2 . . . . . . . . . . . . . . . . . . . . . . 116
4.8 Records in Modula-2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 116
4.9 Dynamic instantiation in Java . . . . . . . . . . . . . . . . . . . . . . . . . 118
4.12 Example of unit written in Delphi . . . . . . . . . . . . . . . . . . . . . . 120
4.13 Example written in PHP . . . . . . . . . . . . . . . . . . . . . . . . . . . . 122
4.14 Branch statement written in COBOL . . . . . . . . . . . . . . . . . . . . . . 123
4.15 Simple example written in Tempura . . . . . . . . . . . . . . . . . . . . . . 124
4.16 GCD example in Tempura . . . . . . . . . . . . . . . . . . . . . . . . . . . 124
4.19 Simple example written in C# . . . . . . . . . . . . . . . . . . . . . . . . 131
4.20 Simple example written in Delphi . . . . . . . . . . . . . . . . . . . . . . . 132
5.1 *QuickSort* algorithm implemented in Java . . . . . . . . . . . . . . . . . . 140
5.2 *QuickSort* eCST: comparative view (Modula-2 and Java) . . . . . . . . . 141
5.3 Successful eCST Generator execution message . . . . . . . . . . . . . . . . . 142
5.4 Inconsistency of Halstead operators - example . . . . . . . . . . . . . . . . 151

# LISTINGS

B.1 *Student* example implemented in Java . . . . . . . . . . . . . . . . . . . . . . 183
B.2 *Student* example implemented in C# . . . . . . . . . . . . . . . . . . . . . . . 186

# Abstract

In a modern approach to software development, a great importance is given to monitoring of software quality in early development phases. Therefore, static analysis becomes more important. Furthermore, software projects are becoming more complex and heterogeneous. These characteristics are reflected in a diversity of functionalities and variety of computer languages and the technologies used for their development. Because of that, consistency in static software analysis becomes more important than it was earlier.

In this dissertation SSQSA: Set of Software Quality Static Analyzers is described. The aim of the SSQSA framework is consistent static analysis. This goal is reached by introducing new intermediate source code representation called eCST: enriched Concrete Syntax Tree. The dissertation mostly focuses on eCST, intermediate representations derived from it, and their generation with description of the tools involved in it.

The main characteristic of eCST is language independence which gives to SSQSA framework two-level extensibility: supporting a new language and supporting a new analysis. This leads to efficiency of adding support on both levels and consistency of added functionalities.

To prove the concept, support for more than 10 characteristic languages was introduced. Furthermore, characteristic static analysis techniques (software metrics calculation, code-clone detection, etc.) were implemented and integrated in the framework.

Established SSQSA framework provides the infrastructure for the further development of the complete platform for software quality control.

# Sažetak

U modernim pristupima razvoju softvera veliki značaj pridaje se kontroli kvaliteta softvera u ranim fazama razvoja. Zbog toga, statička analiza postaje sve značajnija. Takođe, softverski proizvodi postaju sve kompleksniji i heterogeni. Ove karakteristike se ogledaju u raznovrsnosti jezika i tehnologija koje se koriste u procesu razvoja softvera. Zbog toga, konzistentnost u statičkoj analizi dobija veći značaj nego što je to bio slučaj ranije.

U ovoj disertaciji opisan je SSQSA: skup statičkih analizatora za kontrolu kvaliteta (eng. Set of Software Quality Static Analyzers). Namena SSQSA okvira je konzistentna statička analiza. Cilj se postiže uvođenjem nove međureprezentacije izvornog koda nazvane eCST (obogaćeno konkretno sintaksno stablo, eng. enriched Concrete Syntax Tree). Naglasak disertacije je primarno na eCST reprezenataciji koda, reprezentacijama izvedenim iz eCST i procesu njihovog generisanja, sa opisom oruđa angažovanim u ovim procesima.

Osnovna i najbitnija karakteristika eCST reprezenatacije je nezavisnost od jezika u kom je izvorni kod pisan, što SSQSA okviru daje proširivost na dva nivoa: kroz podršku za nove jezike i kroz podršku za nove analize. Ovo dovodi do efikasnog uvođenja funkcionalnosti na oba navedena nivoa, kao i do kozistentnosti uvedenih funkcionalnosti.

Kao dokaz ispravnosti koncepta, podrška za više od 10 ulaznih jezika je uvedena. Takođe, implementirane su karakteristične tehnike statičke analize (izračunavanje oftverskih metrika, otkrivanje duplikata u kodu, itd.) i integrisane u SSQSA okvir.

Na opisani način, postavljanjem SSQSA okvira, obezbeđena je infrastruktura za dalji razvoj kompletne platforme za kontrolu kvaliteta softvera.

# Preface

In the modern approach to software development, a great importance is given to monitoring and control of software quality in the early stages of development. Therefore techniques applicable during design and implementation phase become more important. These techniques applied to an intermediate representation of design or implementation make integral parts of static analysis.

Nowadays software projects are becoming more complex and heterogeneous. These characteristics are reflected in a wide variety of functionality as well as in a wide variety of used computer languages and the technologies used for the development of these functionalities. Therefore, consistency in static analysis becomes more important than it was earlier.

In this thesis SSQSA: Set of Software Quality Static Analysers has been described. The aim of the SSQSA framework is consistent software quality monitoring and control on code level. This goal is reached by introducing new intermediate source code representation called eCST: enriched Concrete Syntax Tree. The thesis mostly focuses on eCST, intermediate representations derived from it, and their generation with description of the tools involved in it.

The main characteristic of eCST is language independence. This characteristic gives to SSQSA framework two-level extensibility: adding support for a new language and adding support for new analysis in a a straightforward way. Advantages included in this extensibility are efficiency of adding support on both supported levels and consistency of added functionalities.

To prove the concept, support for more than 10 characteristic languages of different paradigms (object-oriented, procedural, functional, etc.) was introduced. Furthermore, characteristic static analysis techniques such are software metrics calculation, code clone detection, structural changes analysis, and software network analyses were implemented and integrated in the framework. Usability of this framework is additionally demonstrated by integration with external tools to use the results of SSQSA analysers.

Motivation for developing a new framework begins with the intention to fulfil gaps in the field of systematic application of software metrics by improving characteristics of software metric tools. One of the important weaknesses of available metric tools is the lack of support for calculation of metric values independently on input programming language. In order to fill this gap a new, language-independent software metrics tool SMIILE: Software Metrics Independent on Input LanguagE has been developed.

The general idea for building SSQSA framework originated from SMIILE tool. The development of this tool started in 2007 and its first prototype was finalized in 2010. It was based on eCST as an intermediate representation of the source code independent of input language.

Afterwards, the framework was gradually extended by integration of new tools. Some tool already existed but they were independently developed for specific purposes and each supported only one programming language. Some of analyses introduced in this manner are software networks extraction and analysis, and structural changes analysis. For example, before inclusion of software network extractor and analyser in the SSQSA system, a similar in-house software network extraction tool existed, but it was limited only to Java programming language. Similarly tool for analysis of structural changes of applications written in C# was previously implemented. By accepting the eCST as an intermediate representation of the source code and adapting the implementation, those tools were extended to support all languages that are supported by the framework. Tools included in the framework are applicable also to the software developed in combination of supported languages. In this way the Set of Software Quality Static Analyzers (SSQSA) framework was gradually built up to meet described goals.

Together with new static analysers, analysis specific internal representations were built. The main characteristic of all these representations and their generators is that they are based on eCST representation and thus also language independent.

During four-year period (since 2010), support for more than ten new languages and several new analyses were added. More than 30 students of bachelor, master, and doctoral studies were included and more than 15 research papers related to the SSQSA framework were published. In this way SSQSA project finds application in education and academy, apart from its potential application in industry.

The work descried in this thesis was partially supported by the Serbian Ministry of Education, Science and Technological Development through national projects "Intelligent Techniques and Their Integration into Wide-Spectrum Decision Support" no. OI174023 and "Infrastructure for technology enhanced learning in Serbia" no. III47003, but also through bilateral projects. This support enabled exchange of ideas and experiences with colleagues from different institutions. Similarly, exchanges through DAAD project, CEEPUS network, and COST action helped in gaining valuable results of the project. Special thanks also go to SQAMIA initiative [1].

Author is also grateful to all master and PhD students from Novi Sad, but also from Skopje, Maribor, and Budapest for their work to enable support of various programming languages and analyses. Special gratitude goes to Miloš Savić (Novi Sad), Jozef Kolek (Novi Sad), Melinda Toth (Budapest), and Črt Gerlec (Maribor) for their willingness to participate in SSQSA project and to refer their research to it. Thanks to all colleagues who participated in any way in SSQSA project.

The most important role in SSQSA project plays supervisor of this thesis. Therefore, there are no words which can express gratitude of author to professor dr Zoran Budimac for his ideas, advices and absolute support. Furthermore, thanks to committee members, professor dr Mirjana Ivanović and professor dr Marjan Heričko (Maribor), and professor dr Vladimir Kurbalija, for invested time and effort, and for their valuable comments.

[1]SQAMIA http://www.sqamia.org/

Thanks to all friends who remained present through all these years. Finally, the greatest gratitude belongs to family, but specially to my parents. This doctorate is dedicated to them for their unlimited love and support.

# Chapter 1

# Introduction

The quality of each product, and therefore the quality of the software product can be described as the degree to which a given product meets the needs and requirements of users. The main quality attribute of each software product is correctness. Without correctness it is pointless to talk about other quality attributes such are functionality, usability, reliability, efficiency, portability, and maintainability.

Mentioned attributes of software quality can be monitored, evaluated, and controlled at early phases of software development by examining the source code and other static artifacts, or during the execution and testing process. Assessment of software quality attributes that is made on the source code or any of its internal representations without executing the program is called *static analysis*, while analysis of the program during execution time is called *dynamic analysis*. In the modern approach to software development, a great importance is given to monitoring and quality control in the early phases of development. Therefore static analysis becomes more important.

Static analysis can be described as a set of techniques for program analysis without its execution. Each of these techniques involves the implementation of analysis algorithms on a static representation of the code.

Nowadays software projects are becoming more complex. A number of projects are going on for decades, and through their life cycle software becomes complex and heterogeneous [Wagner, 2014]. These characteristics are reflected in a wide variety of functionality as well as in a wide variety of used computer languages and the technology used for the development of these functionalities. Primarily, this heterogeneity complicates the analysis and quality control over the product life cycle and affects the consistency of the static analysis results. Furthermore, some components have been developed in legacy programming languages such are COBOL and FORTRAN. These components, as well as the languages they were written in, are often older then these analyses so that they usually remain left out of process monitoring and quality control during maintenance. Therefore, when considering these components, the consistency of measurement results for the whole project can not even be a topic of analysis and discussion [Capers, 1996]. Under described conditions it is extremely important to provide a consistent quality of static analysis independent of language and technology used for software development [Ben-Menachem and Marliss, 1997].

The overall subject of the research described in this thesis is consistency of software quality analysis when applied to the source code written in different computer languages. This research focuses on creating a framework for the consistent static analysis of the software product.

Advanced characteristic of created framework are based on the intermediate representation of source code: enriched Concrete Syntax Tree (eCST) [Rakić and Budimac, 2011]. This representation of the source code, as well as the process of its generation, are independent of the input language. In this way, independency of all other integrated components of the input language has been achieved as a basic contribution of the research.

In particular, when a unique representation of a software code regardless of the language in which it was written in is available, it is possible to use a single implementation of the analysis algorithms. Consequently, for each new input language the whole set of integrated analyses is readily available.

The developed framework consists of:

- central component which is responsible for generating eCST representation of the source code;

- generators of alternative representations of source code and design derived from eCST;

- components responsible for analyses based on generated representations of source code;

- external tools integrated to use results of performed analyses.

Described approach contributes to the SSQSA framework with two crucial features [Kolek et al., 2013]:

**Adaptability** or flexibility in supporting new languages and

**Extendability** or scalability in supporting new analyses.

In that way the consistency of results and reliable analysis of software product quality regardless of the input language has been achieved. This is the ultimate contribution of the thesis, especially considering described trends in software development: size, complexity, and heterogeneity of projects.

After adding a new language all available analyses are immediately applicable for this language. In case of adding a new analysis, a single implementation of the analysis algorithm is enough for all supported languages, which means that after integration of a new analysis in the framework it is applicable to all supported languages.

Currently, SSQSA framework support the following input languages: Java, C#, Delphi, C, COBOL, Pascal, Modula-2, Scheme, Erlang, Python, PHP, JavaScript, OWL, WSL, and Tempura. Furthermore, the following analyses were implemented and integrated in the framework: software metrics calculation, software network analyses, code clone detection, and structural changes analysis.

## 1.1 Outline

This thesis describes created framework called SSQSA: Set of Software Quality Static Analyzers. Chapter organisation of the rest of the thesis is as follows.

**Chapter 2: Background** provides the basic concepts and terms to be used in the text. It introduces software quality and related terminology (section 2.1). Afterwards, focus is moved on static analysis and its role in software quality monitoring (section 2.2). Finally (in section 2.3) term of computer languages is introduced and their impact on static analysis is explained. Special attention is paid on diversity of possible input languages with focus on ones supported in SSQSA framework.

**Chapter 3: Justification of SSQSA concept** in its first section describes the starting motivation for the development of the first tool in SSQSA framework, SMIILE, justifying it by preliminary research on weaknesses of available software metric tools at that point of time (in section 3.2). Furthermore, section 3.3 describe the state of the art in the field of intermediate source code representations and their usage in the static analysis techniques related to SSQSA framework. Finally, review of related software solutions is given in the last section of this chapter (section 3.4).

**Chapter 4: SSQSA framework** provides full picture of SSQSA framework and its architecture. *SSQSA overview* (section 4.1) provides architecture with brief overview of the SSQSA components. Section *SSQSA intermediate representations* (4.2) describes the central intermediate representation in SSQSA framework - eCST: enriched Concrete Syntax Tree, but also intermediate representations derived from it. Section *SSQSA components* (4.3) describes constituents of SSQSA framework at different levels: generators and manipulators of intermediate representations, static analysers, higher-level external tool integrated to use results of the SSQSA analysers, etc. The last two sections ot this chapters (4.4 and 4.5) explain adaptability and extendability of the framework as its main characteristics.

**Chapter 5: Validation and Results** describes results of the thesis on two levels. On the first level, section 5.1 describes representative results of testing SSQSA components and shows SSQSA applicability and consistency and correctness of results. Following section, describes in which way SSQSA can find application in industry and education, while applicability in science is obvious.

**Chapter 6: Conclusion** summarises results of the thesis with conclusion and possible further direction of SSQSA development and research.

Each of described chapters ends with brief summary of its contents. Furthermore, as history of SSQSA framework and its components importantly affected contribution of this dissertation. Some sections contain boxed text with important historical facts in the development of SSQSA framework.

**Appendix A: Catalog of universal nodes** contains detailed overview of universal nodes with description and their usage in supported languages.

**Appendix B: Examples** containes larger fragments of source code related to examples used in the dissertation.

# Chapter 2

# Background

## 2.1 Software quality - standards and definitions

The quality of each product, and therefore the quality of the software product can be described as the degree to which a given product meets the needs and requirements of users. This is the definition of the software quality used by ISO (the International Organization for Standardization)[1] in ISO 9126[2] standard. This definition refers to the level of compliance of requirements.

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) work actively in a field of worldwide standardisation. As a result of this cooperation ISO 9126 standard was replaced by ISO/IEC 25000 [3]. ISO/IEC 25000 is a series of standards under the general title Systems and Software Quality Requirements and Evaluation (SQuaRE). New standardisation uses new definition of software quality and related terms.

> *Software Quality* expresses degree to which a software product satisfies stated and implied needs when used under specified conditions.

[1]ISO, 2014 https://www.iso.org

[2]ISO 9126, 2014 http://www.sqa.net/iso9126.html

[3]ISO 25000, 2014 https://www.iso.org/obp/ui/#iso:std:iso-iec:25000:ed-1:v1:en

Software quality standardisation mainly refers to a presence of software quality attributes in final product.

> *Software Quality Attribute* can be described as an inherent property or characteristic of software product that can be distinguished quantitatively or qualitatively by human or automated means.

Required quality attributes affect software quality which place these categories to common categories.

> Category of software quality attributes that bears on software quality is defined as *Software Quality Characteristic.*

Thus, software quality attributes are grouped in characteristics of software quality, but also into sub-characteristics which constitute characteristics.

Software quality model defined by standard ISO 9126-1 distinguished six attributes of software quality. These are: functionality, usability, reliability, efficiency, portability, and maintainability. Currently, ISO 25010 defines the following eight software product quality characteristics: functional suitability, reliability, performance efficiency, usability, security, compatibility, maintainability, and portability. Each of these characteristics is composed of a set of related sub-characteristics [4].

Software quality assurance and control play an important role in software development. During these processes mentioned attributes of software quality are monitored, evaluated, and controlled at different phases of software development. This is done by determining values of different quality characteristics measures in process of software quality measurement.

## 2.2 Static analysis

Software quality monitoring and control is crucial throughout the whole life cycle of a software product. Furthermore, it is very important to take care

[4]ISO 25010, 2014 https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

of the quality from the very early phases of software development process by examining the source code and other static artifacts, or during the execution and testing process. Assessment of quality attributes of software that is made on the source code or any of its internal representations without executing the program is called *static analysis*, while analysis of the program during execution time is called *dynamic analysis.* In the modern approach to software development, a great importance is given to monitoring and quality control in the early phases of development. Therefore static analysis becomes more important.

> *Static analysis* can be defined as a set of techniques for program analysis without its execution.

Each technique involves the implementation of analysis algorithms on an intermediate static representation of the source code. Examples of source code representation used in static analysis are syntax trees and different graph representations. Algorithms implemented on these representations can be used for various purposes.

During the evolution of a software product it changes in various ways. Some of these changes are made on request of users and usually refer to adding a new or change of existing functionality. This affects functionality of the product, but also its quality attributes. On the other hand, improvement of features and characteristics of software product is continuous activity in its evolution. Such activities are reengineering and especially refactoring as its characteristic part. Changes made in this direction usually do not affect functionality of the program but affect its quality. All these changes are to be monitored and their effects on quality of software product are to be supervised.

In software evolution static analysis techniques play multiple roles. Static analysis can be applied to detect weak points of software product by preparation of the data for testing, detection of potential errors, exploring possible execution scenarios, timing analysis and execution time estimation, etc. Furthermore, by static analysis different flaws in design and implementation (so called ‘bad smells‘) can be located. Special case of bad smell to be considered are duplicates or clones in the source code. Static analysis and its role in locating of bad smells in the source code are crucial in preparation stage for improvement changes in software product. Furthermore, static analysis is very important in observing quality attributes during the changes and improvement of each software product. Static analysis are used in detecting, monitoring, and directing the changes by following their effects on observed quality attributes during the evolution.

Techniques of static analysis usually rely on software metrics - it is used as the basic technique of static analysis to measure the attributes of software quality. Since it is not possible to manage what you can not measure [DeMarco, 1986], the measurement or numerical expression of facts about events and objects in the software development process, is used as an elementary process in process monitoring and quality control.

### 2.2.1 Intermediate representations in static analysis

Many concepts and approaches in software analysis (primarily techniques of static analysis) are based on tree representations of the source code. There are two widely used tree representations of a source code: concrete syntax tree (CST) and abstract syntax tree (AST). We introduce eCST as an innovation in a field of tree representations of the source code.

Syntax trees are usually secondary products of language processing tools such are parser, compiler, etc. These tools could be produced automatically by generators or developed manually by implementing the language rules. Parser generators take a language specification usually provided as a language grammar as its input and return parser for that language as an output. This grammar is provided in some default form (e.g. EBNF: Extended Backus-Naur form) or in some generator-specific notation. These generators usually have embedded mechanisms to generate syntax trees as internal structures. Additionally, these mechanisms can be extended with mechanism for enrichment of syntax trees with additional information about language or input source code.

The concrete syntax tree (CST) representation shows how a programming language construct is derived according to the context-free grammar of the language. The root node of a CST represents starting non-terminal symbol of the grammar. Interior nodes in CST correspond to syntactical categories of the language identified by non-terminal symbols of the grammar, while leaf nodes represent tokens of the represented construct.

Abstract syntax tree (AST) is an alternative and more compact way to represent language constructs. While Concrete Syntax Tree (CST) represents all constituent of source code at a very concrete level, without any abstraction, Abstract Syntax Tree (AST) is an abstraction of this representation. It usually consists of abstract elements of language syntax and often does not contain all parts of concrete syntax. AST representation retains the hierarchical structure of language constructs, while omitting details that are either visible from the structure of AST or unimportant for a language processing task. Imaginary nodes are introduced for this abstraction of syntax.

Imaginary nodes do not correspond to concrete tokens, which means that node labels do not explicitly appear in the source code. These nodes are usually root nodes of a sub-tree representing specific program constructs. Thus, imaginary nodes can help in marking the semantics of syntactic constructs. For example, imaginary node VAR_DECL is a convenient root node for the declaration of a variable.

Important characteristic of AST is that its content and structure is not uniquely defined. For example, imaginary nodes can be added freely, depending on the purpose.

Besides syntax trees, static analysis often relies on different graph representation of source code. The commonly used internal representation of the source code for static analysis is Control Flow Graph. Control Flow Graph (CFG) represents the flow trough the control structure within a single function where nodes are basic blocks or statements while directed edges follow the possible execution paths.

> Let G = (V, E, i) be a directed graph, where nodes V correspond to basic program blocks and edges $E \subseteq V \times V$ connect two nodes $v_i, v_j \in V$ iff $v_j$ is executed immediately after $v_i$. This graph G is called *CFG: Control Flow Graph.* Each node $v_i \in V$ which

has no incoming edges ($\nexists v_i \in V : (v, v_i) \in E$) represents the start node or the program.

On the other hand, program can be represented by graphs of various types of dependencies between different software units and entities.

Complex network theory is based on graph theory and statistical analysis. Complex real-world systems represented by typed and/or attributed graphs form different kinds of complex networks. Statistical methods applied on these graphs provide powerful mechanism in network analysis. Complex networks theory has an application in many areas where complex systems are observed. Application areas vary from biology and physics to social networks, computer networks, etc. In a context of software engineering and software development, software networks as a special type of complex networks can be observed [Cohen and Havlin, 2010], [Newman, 2003], [Boccaletti et al., 2006].

Software networks are directed graphs representing relationships or dependencies between software entities (packages, classes, modules, methods, functions, procedures, etc.). Software network can be observed as a static representation of source code and can be used in analysis of the quality of software development process and software product with particular application in a field of large-scale software systems. Each node represents software entity (package, class/module, procedure/function/method, etc.) defined in the source code of a program. Each link denotes different types of dependencies among them (calls, uses, references, contains, etc.) [Myers, 2003], [Šubelj and Bajec, 2012].

Depending on dimension and the level on which software network represents the software product, several types of networks can be distinguished [Savić et al., 2014].

- In horizontal dimension three levels of networks can be extracted:

**Low level:** network representing collaboration inside class/interface/module (SCG: static call graph / MCN: method collaboration network, FUGV: function uses global variables, etc.);

**Middle level:** network representing collaboration between classes/interfaces/modules;

**High level:** package collaboration network.

- Vertical dimension reflects dependencies between software entities on different abstraction levels represented by hierarchy tree.

Graph including all described dependency networks is called GDN: General Dependency Network [Heckerman et al., 2001], [Savić et al., 2014].

> Let G = (V, E, i) be a directed graph, where nodes V correspond to software entities and edges $E \subseteq V \times V$ connect two nodes $v_i, v_j \in V$ iff $v_j$ depends on $v_i$. This graph G is called *GDN: General Dependency Network.*

By filtering appropriate information from GDN described graph representations can be extracted.

### 2.2.2 Representative static analysis techniques

In this section, representative static analysis techniques will be described. Techniques are carefully selected to vary from basic (such are software metrics) to advanced (such is hybrid code-clone detection). The most of described analyses are integrated in the SSQSA framework.

#### 2.2.2.1 Software metrics

The measuring and continual monitoring of a software product is crucial for success in the software development process. The main instrument in these processes are software metrics which can be defined as values that reflect the status of a software product or its specifications.

> *Software metrics* can be defined as numerical values that reflect the properties of a software development processes and software products [N. Fenton, 1996].

The first software metrics have been introduced over a half of century ago. Software metrics were changing along with the changing needs and demands of the market of software products. The first published book that describe software metrics appeared in 1976 [Gilb, 1976], but the first attempts at applying software metrics had already taken place in the late 1960s [Fenton and Neil, 1999]. Originally, measuring was applied in controlling the time and resources required for execution of the observed programs. Later, when the programs became more complex and possibility of errors increased, metrics were used to measure the complexity of a program and assess the possibility of the occurrence and frequency of errors. Afterwards, the quality of software design became more important due to the increased amount of errors that occurred in the early stages of software development. Therefore, metrics for monitoring the characteristics of the design were introduced. With growth of costs of software development, an important place in this system of measurements take metrics for monitoring the entire process of development of a software product.

Nowadays, software metrics can be categorised into three major subgroups: process metrics, project metrics, and product metrics [Kan, 2002].

Process metrics are related to the characteristics of the process. They are used with intention to improve the observed software development process. Frequently used measure is the amount of effort needed to invest in the entire software development in specific phases or in specific activities during the development phase. These efforts are reflected by some of the standard measures such as man-months or man-days. Other examples of the process metrics are frequency of repetition of errors, the time required to eliminate or reduce errors, etc.

Project metrics represent certain values related to resources and costs, their allocation to the individual stages in the evolution of products, productivity, and other items related to project planning and management.

Product metrics reflect characteristics of software product itself. Some software features are visible to the user, while others are only visible to a development team. In accordance to availability of observed attributes to participants in development process and end user, software metrics can be divided into internal and external.

Characteristics of the software product reflected by external metrics are available not only to the development team but also to an end user of the product. It is not a rare case that external characteristics are expected to meet the subjective attitude or feeling of a user. Therefore, it is a hard task to define and measure external metrics.

Internal metrics reflect those characteristics visible only to the team involved in a development. By subject of measurement they reflect the size, complexity, structure, and architecture of the software product.

Focus in the dissertation is on **internal product metrics**.

Size as an internal attribute of a software product is usually measured by number of lines of code. Lines of Code (LOC) family of metrics consists of several modifications of this basic measure of length of source code such are [Fenton and Neil, 2000], [Bhatt et al., 2012]:

- SLOC: Source Line of Code reflecting number of lines containing source code without empty lines and lines containing comments;

- CLOC: Comment Line of Code reflecting number of lines containing comments;

- BLOC: Blank Lines of Code reflecting number of lines containing nor source neither comments;

- PLOC: Physical Line of Code reflecting size as it is without taking into account programming style or code formatting;

- LLOC: Logical Line of Code reflecting size after applying codding style rules or formatting and logical restructuring of statements over the lines.

Maurice Halstead [?] introduced one of oldest product size and complexity metrics set. Halstead Metrics (H) reflects the size and the complexity of the program calculated based on number of operators and operands. Metrics from the Halstead's set are following:

- Basic Halstead metrics

  - $n_1$ - the number of distinct operators;
  - $n_2$ - the number of distinct operands;
  - $N_1$ - the total number of operators;
  - $N_2$ - the total number of operands.

- Derived Halstead metrics

  - $n$ - Program vocabulary ( $n_1 + n_2$ );
  - $N$ - Program length ( $N_1 + N_2$ );
  - $\hat{N}$ - Estimated program length ( $n_1 log_2 n_1 + n_2 log_2 n_2$ );
  - $V$ - Volume ( $N log_2 n$ );
  - $D$ - Difficulty ( $\frac{n_1 * N_2}{2n_2}$ );
  - $E$ - Effort $(V * D)$;

Halstead program volume (V) expresses the amount of the information to be absorbed in order to understand the program, while program difficulty (D) expresses how difficult is to understand the program after necessary information are absorbed. Program effort (E) reflects how much effort should be invested in rewriting the program.

Nowadays there exist different modifications and approximations of these metrics, but also some process and project metric are derived from above measures.

CC: Cyclomatic Complexity reflects complexity of the control-flow structures in the program. It was introduced by McCabe [McCabe, 1976] at the same period as Halstead introduced his metrics. CC is currently one of the mostly used metrics.

Value of CC represents the number of linearly independent (base) path of program execution. Any execution path through the program sequence can be created as an linear combination of these base paths. From this we conclude that, when testing the program, it is enough to test only the base paths. For this reason this metric is used also in the software testing as a boundary of the number of needed test cases.

Graph which is representative of all possible execution paths of the program is called the CFG: Control Flow Graph. Control flow graph, CFG (N, E), consists of a set of nodes (N) and the set of edges (E). Each node represents a single statement, while the branches represent the flow between the commands, or the possible execution direction.

> Let each control flow graph refers to a basic block (procedure, method, ...), out of which each has exactly one input and exactly one output, and for each path of program execution there is exactly one path in the control flow graph. Let the CFG (N, E) be the control flow graph, where N is the designated set of nodes, and E is set of edges. Let $e =\mid E \mid$ - the number of branches in the graph and $n =\mid N \mid$ number of nodes in the graph. Then *CC: Cyclomatic Complexity* of graph CFG (N, E) is defined as:

$$CC = v(CFG) = e - n + 2$$

For a non-empty graph the minimal CC is equal to 1. CC in the graph increases with increasing number of branching, and there are some recommendations from the experience that the CC should never exceed 10. If such a great value occurs, it is possible to split the function to the multiple functions with a smaller CC. Such functions are easier to test and maintain.

CC can be calculated by creating the CFG and counting the nodes and the branches and using the formula, but also on some of the alternative ways, such are counting control-flow predicates or counting graph areas.

When counting predicates, the basic idea relies on the fact that each branching point in the graph represents a conditional statement, or an appropriate predicate. Thus, for each conditional predicate with one condition CC is increased by one, and for those with multiple conditions for each new condition CC is also increased by 1. CC is also increased for each logical operator (AND, OR, ...) under conditions of a observed statement.

In order to apply the method of counting the area of the graph, the graph need to be a planar one. This means that it does not contain branches that intersect each other. Then, when observing the infinite area in which the graph lies, where number of areas of the graph is labeled with R, CC can be defined as $CC = v(CFG) = R$.

CC measure has its good and bad characteristics, and there are different arguments pro and contra its usage. The fact is that it is widely used in practice for decades and adaptable to new techniques in programming, which adds value to this metric. Furthermore, there are numerous metrics derived from CC used in data-flow analysis, program complexity analysis, or in measurement for object-oriented systems.

Different design metrics are also widely used in static analysis (general ones or specially devoted to object oriented design). Many variations of design metrics were developed but some general groups can be selected as mostly used:

**Lorenz metrics** [Lorenz and Kidd, 1994]

- AMS: Average Method Size (expressed in LOC);
- ANMC: Average Number of Comments per Method (expressed in CLOC);
- ANMC: Average Number of Methods per Class;
- ANIVC: Average Number of Instance Variables per Class;
- CHNL: Class Hierarchy Nesting Level (counting starts from the top level class);
- NSSR: Number of Subsystem-to-Subsystem Relationships;
- NCCR: Number of Class-to-Class Relationships in Each Subsystem;
- IVU: Instance Variable Usage expresses the usage of instance variables by methods. It can signalise some design problems;
- NPRC: Number of Problem Reports per Class (higher number of reported problems can signalise that class should be redesigned);
- NTCR: Number of Times Class is Reused;

- NCMTA: Number of Classes and Methods Thrown Away expresses level of redesigning in the development process when classes are redesigned and methods and whole classes can be replaced or excluded.

**Morris metrics** [Morris, 1989]

- NMC: Number of Method per Class;
- NID: Number of Inheritance Dependencies;
- DCBO: Degree of Coupling Between Objects express level of dependency between objects. High coupling makes maintenance difficult and decreases reusability;
- DCO: Degree of Cohesion of Objects expresses level of dependency between elements of objects. Low cohesion can signalize that object design should be redesigned;
- OLE: Object Library Effectiveness expresses the level of reusage of objects of library classes;
- FE: Factoring Effectiveness expresses contribution of unique methods in total number of methods. Higher factoring means lower level of code duplication;
- DRIM: Degree of Reuse of Inheritance Methods (potential and real, overriding, etc);
- AVC: Average Method Complexity (where method complexity is measured as CC);
- AG: Application Granularity reflects how small peaces of functionality are objects responsible for. Finely grained applications are easier for maintenance and reusage.

**Chidamber & Kemerer metrics** [Chidamber and Kemerer, 1994]

- WMC: Weighted Methods per Class (sum of method complexity expressed as CC values);
- DIT: Depth of Inheritance Tree;

- NOC: Number of Children;
- CBO: Coupling Between Object Classes;
- RFC: Response for a Class;
- LCOM: Lack of Cohesion in Methods;

**MOOD: Metrics for Object Oriented Design** [?]

**Encapsulation metrics** :

- MHF: Method Hiding Factor expresses contribution of hidden methods in total number of methods;
- AHF: Attribute Hiding Factor expresses level of attribute encapsulation. It is contribution of hidden attribute in total number of attributes.

**Inheritance metrics** :

- MIF: Method Inheritance Factor expresses how much is inheritance applied to methods;
- AIF: Attribute Inheritance Factor expresses how much is inheritance applied to attributes.

**Polymorphism metrics** :

- PF: Polymorphism Factor expresses level of usage of polymorphism.

**Coupling metrics** :

- CF: Coupling Factor expresses level of communication between classes. This value should be low in good object design.

Nowadays, various software metrics tools are used for automatic calculations of software metrics. In chapter 3, problems in the area of consistent and systematic application of software metrics will be presented. During the exploration the strong dependency of software metrics of an input programming language was recognized as one of the main weaknesses of the applicability in this field.

#### 2.2.2.2 Software networks analysis

As described in section 2.2.1, software network is a graph that represents different aspects of software architecture and design. Each node represents one software entity (package, unit, function, etc.) while links reflect the dependencies between them (package collaboration, class collaboration, function calls, etc.). Obviously, the most of design metrics can be calculated based on these graph representations of the source code, while size and complexity metrics can be used to reflect internal characteristics of observed nodes. As software networks rely on formal theory of complex networks the formal background for the calculation of design metrics can be stated.

Important properties in complex networks analysis are [Newman, 2010], [Cohen and Havlin, 2010], [Boccaletti et al., 2006], [Newman, 2003]:

**Size** expressed by number of nodes;

**Density** reflecting the ratio between number of existing links in the network and number of all possible links (up to complete graph);

**Node degree** where distribution and correlation of in, out and total degrees are observed, existence of scale-free property (power low degree), etc.;

**Path length** with shortest path lengths, diameters (path with maximum length from the set of shortest paths), betweenness, small-worldness (tendency of nodes to be neighbours to each others), etc;

**Connectedness** observed through the characteristics of connected components;

**Community structures** grouping the nodes which are more densely interconnected with respect to the rest of nodes in the network;

**Clustering** observed as local and global clustering through the clustering coefficients;

**Motifs** as patterns or frequently repeated sub-graphs.

When software networks are observed these attributes can be mapped to different design metrics [Savić et al., 2014]. This mapping depends on viewpoint of observation and abstraction level of observed network. For example, when vertical software network representing entities hierarchy is observed, the node degree has multiple meanings. If CONTAINS links between a package and classes or interfaces are observed, the node degree represents number of classes or interfaces (respectively) in the package.

Similarly, if relationships between classes and methods or attributes are observed, the node degree represents number of methods or attributes in the class.

Meaning of node degree in horizontal networks representing dependencies in source code is based on referencing between packages, classes, or methods. Number of ingoing links is fan-in and number of outgoing links is fan-out of package, class, or method respectively. These values influence coupling between entities.

If inheritance tree or, more precisely, sub-graph of class collaboration network filtered by EXTENDS links is observed, the in-degree of a class expresses number of children for that class. Furthermore, if length of paths in the inheritance tree is observed, the shortest path from observed class to the root node expresses the value of DIT metric for the observed class.

A very important step in network analyses is detection of connected components. Connectedness is observed in calculation of cohesion metrics. Number of connected components in SCG and FUGV restricted to methods and attributes participate in calculation of LCOM metrics.

All example metrics are defined in the field of classical software metrics while mapping from complex network analysis to software metrics calculation is an additional benefit to the subject.

Some properties of complex network applied on the software networks give improvements of metrics defined in the field of software metrics. For instance, graph clustering applied on software network result with an improved measure of cohesion. GCE: Graph Clustering Evaluation metrics determines the quality of clusters obtained by community detection algorithms. While classical cohesion estimation algorithms consider only internal dependencies of software entities, GCE takes into account both, internal and external links [Savić and Ivanović, 2014].

Software networks satisfy requirements for scale-free networks - node degree follows some power-low distribution. High degree in class collaboration network means high complexity in the software design. It can signalise flaws in software design because class functionality used by observed class node is partially implemented in some other class. On the other hand, high out degree implies high reusability of functionality of observed class because its implementation is often used by other class. As some nodes in the networks can be source of faults, decreasing of complexity and reusability of the classes can decrease vulnerability of the observed system. Described interventions can be done globally by affecting overall node degrees or locally by following and affecting characteristics of selected node[Šubelj and Bajec, 2012].

Detection of giant connected component, observing strongly and weakly connected components in network by following number of nodes, the number of links, diameter, the small-worldness, the clustering, etc. can provide important information about software quality attributes. Small-worldness of the network is related to high clustering. It can be observed through the average distance between nodes. These parameters express characteristics od software structure and design [Šubelj and Bajec, 2012].

Motifs and other patterns in the network have an application in design patterns detection, algorithmic patterns recognition, clone detection etc. [Myers, 2003]

#### 2.2.2.3 Code clone detection

Clones or duplicates in source code are one of the important factors affecting maintenance of software products. They usually appear as a result of reuse of existing source code by copy-paste with or without modifications. This is known as cloning the code. Duplicates in the source code do not have to be identical copies, but parts or fragments of the source code similar to some extent. [Roy et al., 2009] introduces definition of code fragment in order to define code clone.

> *CF: Code fragment* is any sequence of code lines (with or without comments). It can be of any granularity, e.g., function definition, begin-end block, or sequence of statements. A CF is identified by its file name and begin-end line numbers in the original code base and is denoted as a triple (CF.FileName, CF.BeginLine, CF.EndLine).
>
> A code fragment CF2 is a *CC: Code Clone* of another code fragment CF1 if they are similar by some given definition of similarity. This implies f(CF1) = f(CF2) where f is the similarity function.

Two clone fragments form a clone pair, and more than two clone fragments form a clone class or clone group.

Similarity function depends on similarity definition which is determined by the viewpoint from which similarity is observed. Fragments of the code can be similar by syntax or semantics. According to that clones can be divided in four basic categories [Roy et al., 2009], [Rattan et al., 2013]:

**Exact clones** are real clones or identical segments of source code possibly different with respect of white spaces

**Renamed/parametrized clones** are identical copies according to the syntax, but with possible differences in identifiers, literals, types, layout, and comments.

**Near miss clones** are syntactically similar copies with more important changes such are statement insertions and deletions in addition to changes in identifiers, literals, types, and layouts.

**Semantic clones** or functional clones are fragments of the source code similar in their meaning or functionality but without important similarities in syntax.

Additionally, [Rattan et al., 2013] introduce higher level clones:

**Function clones** reflect similarities in granularity of a function, procedures, or method.

**Structural clones** reflect similarities in software design.

**Model based clones** are duplicates in models of software systems.

Article [Roy et al., 2009] provides overview, comparison, and evaluation of available techniques for code clone detection. Observed approaches are categorised as:

**Textual approach** in most cases uses directly raw source code in the clone detection process. Selected segments (usually lines) of code are compared. As a comparison criteria hash code of the text segment or some other value representing it can be used.

**Lexical approach** or token-based clone detection relies on lexical analysis of source code for splitting the source code into a sequence of tokens. Afterward, sub-sequences are observed and compared in order to detect duplicates.

**Syntactic approach** relies on syntactic analysis of source code. Parser is used to produce an intermediate tree representation of the source code (AST or CST). A duplicates detection algorithm can be applied on this representation. There are two mostly used approaches:

> **Tree-matching approach** for finding cloned sub-trees
>
> **Metrics-based approach** relies on low-level code metrics gathered on syntax level such are size and complexity metrics and comparison of their values in finding similar fragments in code.

**Semantic approach** relies on other static analysis techniques. It often uses graph or network representation of the program to find duplicate dependencies in it. This approach can also use metric results for clone detection based on higher-level metrics gathered from semantics of the program.

There are various modifications or extensions of these basic approaches by involving improved algorithms for comparison, different visualisations, mining techniques, etc. In order to gain better results the basic techniques are combined and new hybrid approaches relying on good properties of isolated techniques are derived [Roy et al., 2009].

Lexical approaches are generally more robust over minor code changes such as formatting, spacing, and renaming than textual techniques. In syntactic approach tree representation allows abstraction of variable names, literal values, and other tokens in the source code which enable more sophisticated duplication detection. Syntactic and semantic approaches give even more freedom in duplicate detection.

Furthermore it can be noted that metric-based approach can be observed as separate category applicable at two levels (syntactic and semantic) and that this approach is very suitable to be combined with other approaches to gain better results.

#### 2.2.2.4 Software changes analysis

Software evolution could be understood as continuous adaptation of observed software product. Software changes that are caused by this process, are usually partitioned into three general categories [Madhavji et al., 2006]:

- fixes of errors in the source code, software design, architecture, etc.
- improvements of performances, usability, maintainability, etc.
- enhancements that involve new features or functionality

It is important to analyze evolutional changes from a quantitative and qualitative point of view. By comparing these aspects of changes effectiveness of changes can be estimated and discussed.

Structural source code changes are continuous during software development. They are usually made during introduction of a new functionality or in reengineering, refactoring, and debugging processes. These are situation when it is needed to add, remove, or modify some unit (class, module, etc.), function (method, procedure, etc), etc. All these changes are to be monitored and analysed already on code level. For these purposes some other techniques of static analysis such are software metrics, and clone detection can be involved.

## 2.3 Computer languages

The term *computer languages* can be used for a broad set of artificial languages which can be processed by computer. Computer languages differ depending on form of appearance, paradigm, purpose, phase of software development in which they will be used, level of abstraction, etc.

Computer languages can appear in a graphical or textual form, while textual form can be based on natural languages (linguistic) or rely on mathematics (formal). Textual code representation of the visual notation (graphical or not) is always generated.

Following the software development process, different computer languages or notations are used for system and software requirement specification, static (e.g. ADL: Architecture Description Language) and dynamic software design description (e.g activity diagrams), construction of software solution, etc. Construction includes coding, testing, debugging and all activities related to producing executable solution of the problem. Construction languages can be categorised based on their purpose: configuration languages, toolkit languages, scripting languages, and programming languages. Languages designed to be used for a specific purpose are known as domain specific languages, while languages designed for general usage are general purpose languages. [Bourque et al., 2014]

Languages used in early phases of software development are often formal (specification languages) or graphical (modeling languages), while graphical languages can also have formal background. Special set of languages are designed for data and knowledge description and manipulation (OWL, XML, SQL, etc.).

Languages can also be categorised based on a level of abstraction. Low-level languages are, by syntax and semantic, close to the machine languages, while high level languages are much more readable by humans. Middle-level languages are usually designed to be used in maintenance phase while reenginering, refactoring, (software) migrating, and other transformation techniques.

Furthermore, computer languages can be categorised by paradigm or based on supported programming style. Some of programming paradigms are: unstructured, structural (procedural, imperative), declarative (logic, functional, etc.), object oriented, aspect oriented, etc.

Programming languages are subset of computer languages. They are designed for implementation of an executable solution. The most of described categorisations of computer languages (purpose, level, etc.) are applicable on programming languages. Sometimes, the term *programming language* is used as a synonym of the term *computer language* if programming is defined to involve analysis, design, implementation, testing, debugging, and maintenance of software in which case programming languages can be categorised also based on development phase (same as computer languages). Otherwise, this categorisation is not applicable because usage of programming languages is then strictly related to the implementation phase.

The history of computers and related languages begins in the middle of 19th century with the Analytical Engine, mechanical general-purpose computer designed by mathematician Charles Babbage. First electronic computers and first programming languages appear the whole century later. Some of the oldest languages which are still present in the practice are FORTRAN (1957) designed for scientific and engineering applications, functional language Lisp (1958), and business-oriented COBOL (1959). Fundamental paradigms were established a decade later (1970s), while the following decade (1980s) is characterised by consolidation in the field of imperative programming. 1990s are known by first appearance of internet and development of script, mark-up, object-oriented, multi-paradigm and domain-specific languages. 21st century is a period of enormous growth in the field of information technologies. Software becomes large and complex while spectrum of languages mixed in development of software products is wide and diverse.

In this dissertation the term *source code* denotes a code written in any computer language independently of notation, purpose and paradigm. Diversity of computer languages, their syntax and semantic makes static analysis to be a difficult task. The most affected is a consistency of results of static analysis. Consistent static analysis is the main goal of SSQSA framework, while the proof of concept will be done on the example of some representative computer languages.

In the rest of this section, an overview of computer languages by categories is provided, with a focus on languages supported in the SSQSA framework. Furthermore, the overview enables a reader to gain an insight into the differences between the languages.

### 2.3.1 General purpose languages

The main characteristic of general purpose languages is that they are designed to be used in development of wide range of software products. They do not contain constructs dedicated to some concrete domain or if they do, these constructs are introduced only to enrich the language but the supported domain is not in the main focus of the language. Based on applied programming style or paradigm, general purpose languages can be categorised as procedural, object-oriented, functional, mixed-paradigm languages, etc. This categorisation is not very strict because nowadays general purpose languages tend to become mixed-paradigm languages applicable in many areas. Therefore, this categorisation is based mainly on the initial design and the main characteristics of the languages. For example, many languages (e.g. Java that is primarily object-oriented language) are introducing support for lambda calculus which is characteristic of functional languages, while some other languages (e.g. Scala) originally supported multiple paradigms including functional and object-oriented.

#### 2.3.1.1 Procedural languages

Procedural languages are derived from the structured programming but follows the idea of splitting the program functionality into smaller functional blocks, so-called procedures. In that way programming in these languages consists of splitting the problem to sub-problems, writing the solutions of the sub-problems in a form of the procedures, and finally writing the main program consisting of calls of the implemented procedures. Obviously, procedural programs are written in imperative style of programming.

Procedures can be grouped into modules. This facility is supported by the modular languages. It is not the rare case that procedural languages support modularisation.

Procedural languages were introduced in 1950s. Some of the first widely used procedural languages were Fortran and BASIC. Nowadays, some of the most present procedural languages are Pascal, Modula-2, C, etc.

Listing 2.1 contains *QuickSort* algorithm implemented in Modula-2 as an illustrative example of source code written in a procedural language.

Listing 2.1: *QuickSort* algorithm implemented in Modula-2
Listing 2.1: *QuickSort* algoritam implementiran u Moduli-2

```
MODULE Quick; FROM IO IMPORT WrCard, RdCard, WrLn, WrStr, OK;

  CONST
    Max = 10;
  TYPE
      Index = [0 .. Max+1];
      Arr = ARRAY [1 .. Max] OF CARDINAL;
  VAR
      arr: Arr;

  PROCEDURE ReadInput(VAR arr: Arr);
    VAR
      i: Index;
    BEGIN
      FOR i:= 1 TO Max DO
          REPEAT
              WrStr('Input ');
              WrCard(i, 2);
              WrStr('. array member: ');
              arr[i]:= RdCard();
              WrLn;
          UNTIL OK;
        END;
    END ReadInput;

    PROCEDURE WriteOutput(VAR arr : Arr);
    VAR
      i: Index;
      temp : CARDINAL;
  BEGIN
      FOR i:= 1 TO Max-1 DO
        temp := arr[i];
        WrCard(temp, 1);
        WrStr(', ')
      END;
      temp := arr[Max];
      WrCard(temp, 1);
  END WriteOutput;

PROCEDURE Sort(VAR arr : Arr; Left, Right : Index);
  VAR
    i, j : Index;
      middle : Index;
      temp : CARDINAL;

  BEGIN
      i := Left;
      j := Right;
      middle := arr[(i + j) DIV 2];
      REPEAT
        WHILE (arr[i] < middle ) DO
            INC(i);
        END;
        WHILE (arr[j] > middle) DO
            DEC(j);
        END;
        IF (i <= j) THEN
      temp := arr[i];
      arr[i] := arr[j];
      arr[j] := temp;
      INC(i);
            DEC(j)
        END;
      UNTIL (i > j);
      IF (Left < j) THEN
        Sort(arr, Left, j);
      END;
      IF (i < Right) THEN
        Sort(arr, i, Right);
      END
END Sort;

  PROCEDURE QSort(VAR arr: Arr);
  BEGIN
    Sort(arr, 1, Max)
  END QSort;
```

```
BEGIN
  ReadInput(arr);
    WrStr('Original array: ');
    WriteOutput(arr);
  WrLn;
  WrStr('Sorted array: ');
    QSort(arr);
    WriteOutput(arr)
END Quick.
```

#### 2.3.1.2 Object-oriented languages

Historically and fundamentally, object-oriented languages follow procedural ones keeping imperative style, modularisation, and solution divided into well defined operations. While procedural languages were focused on implementing operations on data structures as functional blocks packed in procedures, introducing of objects is based on idea of packing data structures and their operations into the logical units. Therefore it can be concluded that object-oriented programming is derived from procedural one and implicitly from structural programming. It keeps good characteristics of previous approaches and introduces their improvements in the form of objects. New approach brought numerous associated improving concepts such is encapsulation, inheritance, and polymorphism.

Examples of mostly used object-oriented languages are currently Java, C++, and C#. An illustrative example of source code is given in Listing 2.2. This Java implementation of some arbitrary list (by usage of Array data structure) provides constructor for creating the list object and only two basic operation over this data structure: insertion of new element and access to the element at $i^{th}$ position. It can contain implementations of sorting algorithms or any other needed operation over the list.

Listing 2.2: *ArbitraryList* implemented in Java
Listing 2.2: *ArbitraryList* klasa implementirana u Javi

```
import java.util.Arrays;

public class ArbitraryList<Element> {
    private int size = 0;
    private static final int DEFAULT_CAPACITY = 10;
    private Object elements[];

    public ArbitraryList() {
      elements = new Object[DEFAULT_CAPACITY];
    }

    public void add(Element e) {
      if (size == elements.length) {
          extendCapacity();
      }
      elements[size++] = e;
    }

  public Element get(int i) {
      if (i >= size || i < 0) {
          throw new IndexOutOfBoundsException("Out_of_bounds!");
      }
      return (Element)elements[i];
    }

    private void extendCapacity() {
      int currentSize = elements.length * 2;
      elements = Arrays.copyOf(elements, currentSize);
    }
}
```

#### 2.3.1.3 Functional languages

Functional programming is based on the lambda calculus. Functional languages can be described as syntactic interfaces to lambda calculus with less or more improvements by additional constructs. Declarative style of programming is characteristic of programs written in functional languages.

Functional languages were introduced and developed in parallel with structured and procedural languages. One of the first programming, and thus, one of the first functional languages was Lisp. It appeared in the middle of 20th century. Some of the most actual functional languages are Erlang, Lisp, Haskell, Scheme, etc.

*QuickSort* algorithm implemented in Erlang is given in Listing 2.3.

Listing 2.3: *QuickSort* implemented in Erlang
Listing 2.3: *QuickSort* algoritam implementiran in Erlangu

```
sort([Pivot|T]) ->
    sort([ X || X <- T, X < Pivot]) ++
    [Pivot] ++
    sort([ X || X <- T, X >= Pivot]);
sort([]) -> [].
```

### 2.3.1.4 Mixed-paradigm langauges

Mixed-paradigm or multi-paradigm languages are designed to support more than one paradigm. As has been described earlier, actual trend is to use multiple languages in software development, to apply different programming styles, to mix paradigms, etc. Following these trends, language designers tend to make languages as general as possible widening the set of language constructs and mixing paradigms. Furthermore, some of the languages have grown from single-paradigm to mixed-paradigm, while some others were initially designed to support different styles and paradigms.

Historical development of mixed-paradigm languages can not be clearly determined, but definitely last decades shown the growth of this phenomenon. Illustrative examples of mixed-paradigm languages widely used at present are: Delphi that combines procedural and object-oriented approach, Scala mostly combining object-oriented and functional paradigm, etc.

In Scala, it is possible to implement some algorithm (for example *QuickSort*) by applying functional style (Listing 2.4) or in more object-oriented way (Listing 2.5).

Listing 2.4: *QuickSort* implemented in functional Scala
Listing 2.4: *QuickSort* algoritam implementiran u Scali (funkcionalni pristup)

```
Array[Int]):
Array[Int] = { if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
      xs filter (pivot ==),
      sort(xs filter (pivot <))
    )
  }
}
```

Listing 2.5: *QuickSort* implemented in object-oriented Scala
Listing 2.5: *QuickSort* algoritam implementiran u Scali (objektno-orientisani pristup)

```
def sort(xs: Array[Int]) {
  def swap(i: Int, j: Int) {
    val t = xs(i);
    xs(i) = xs(j);
    xs(j) = t
  }

  def sort1(l: Int, r: Int) {
    val pivot = xs((l + r) / 2);
    var i = l;
    var j = r;
    while (i <= j) {
      while (xs(i) < pivot) i += 1;
      while (xs(j) > pivot) j -= 1;
      if (i <= j) {
        swap(i, j);
        i += 1;
        j -= 1
      }
    }
    if (l < j) sort1(l, j);
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

### 2.3.2 Domain-specific languages

Domain-specific languages are designed to enable easier software development in some specific domain. Main characteristics of these languages are the following ones: language is simple, syntax is based on thin domain vocabulary, and constructs are raised to the higher level of of abstraction. Thus, developers are relaxed of redundant details which are out of their domain. It is easier to learn these languages and development is quicker. On the other hand, possibilities are very restricted, usually only to the specific domain (e.g. specification, modeling, etc.) While some languages are very pure, dedicated to the certain domain (e.g. BNF, Latex, Matlab, VHDL, HTML, etc.), for some languages it is pretty hard to determine if they belong to general-purpose or domain-specific languages (e.g. legacy language COBOL, Script languages such are JavaScript and PHP, etc.). [Fowler, 2010], [Raja and Lakshmanan, 2010], [Mernik et al., 2005]

#### 2.3.2.1 Specification languages

Specification languages are used in phases of analysis and design to specify system at high level of abstraction. These languages are usually formal, which means that some formal theory (e.g. logic) is applied in their design. Specification refinement and writing executable specifications enables easier validation, verification, and further refinement up to the implementation. A proper specification language will make these processes easier and more efficient. Examples of specification languages are: Z notation, VDM, ITL, Tempura, etc. Specification of a *stack* with two operations (*new* and *push*) is given in Figure 2.1, while Listing 2.6 contains *QuickSort* specification written in Tempura.

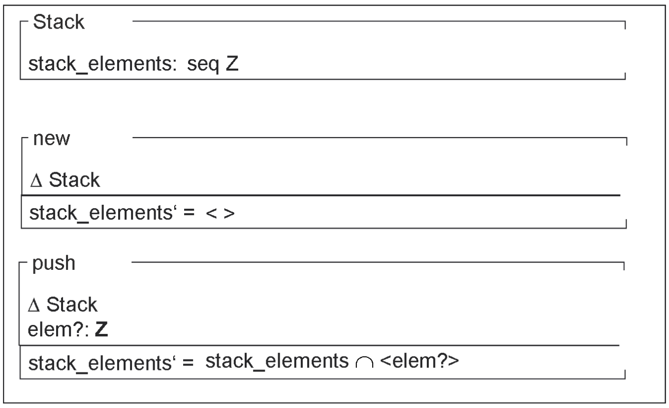

```
Stack
stack_elements: seq Z

new
Δ Stack
stack_elements‘ = < >

push
Δ Stack
elem?: Z
stack_elements‘ = stack_elements ⁀ <elem?>
```

Figure 2.1: Specification of Stack written in Z notation
Slika 2.1: Specifikacija za Stek napisana u Z notaciji

Listing 2.6: *QuickSort* specification written in Tempura
Listing 2.6: Specifikacija *QuickSort* algoritma napisana u Tempuri

```
define serial_quicksort(L) = {
    if |L| <= 1 then
      empty
    else
      exists pivot : {
      quick_partition(L,pivot);
      {serial_quicksort(L[0 to pivot])
        and stable L[pivot to |L|]};
      {serial_quicksort(L[(pivot+1) to |L|])
        and stable L[0 to (pivot+1)]}
  }
}.
```

### 2.3.2.2 Modeling languages

Modeling languages are used to represent structure of the system or knowledge and to define roles of its constituents. This is done through the set of rules. Modeling languages can be textual and graphical. Graphical representation through the different diagrams is very popular nowadays. Diagrams always have their textual representation (often XML which is also represent of modeling languages) but to be categorised as graphical language, it has to have graphical interface. Spectrum of application of modeling languages is very wide. Some well-known examples of modeling languages are ADL: Architecture Description Language, different ontology description languages such is OWL: Web Ontology Language, markup languages such are XML: Extensible Markup Language and HTML: HyperText Markup Language, Latex, etc. An illustrative example[5] of ontology diagram is given in Figure 2.2, while fragment of corresponding OWL code is given in Listing 2.7.

Listing 2.7: *Contact* ontology written in OWL notation
Listing 2.7: Ontologija Contact napisana u OWL notaciji

```
Ontology(
  Declaration(Class(:ContactInformation))

  Declaration(DataProperty(:city))
```

[5]Ontology examples, 2015: http://ebiquity.umbc.edu/ontology/

```
Declaration(DataProperty(:company))
Declaration(DataProperty(:country))
Declaration(DataProperty(:department))
Declaration(DataProperty(:email))
Declaration(DataProperty(:phone))
Declaration(DataProperty(:postalCode))
Declaration(DataProperty(:room))
Declaration(DataProperty(:state))
Declaration(DataProperty(:street))
Declaration(DataProperty(:url))

AnnotationAssertion(rdfs:label :ContactInformation "ContactInfo")
  SubClassOf(:ContactInformation DataMaxCardinality(1 :city))
  SubClassOf(:ContactInformation DataMaxCardinality(1 :company))
  SubClassOf(:ContactInformation DataMaxCardinality(1 :country))
  SubClassOf(:ContactInformation DataMaxCardinality(1 :department))
  SubClassOf(:ContactInformation DataMaxCardinality(1 :postalCode))
  SubClassOf(:ContactInformation DataMaxCardinality(1 :room))
  SubClassOf(:ContactInformation DataMaxCardinality(1 :state))
  SubClassOf(:ContactInformation DataMaxCardinality(1 :street))
AnnotationAssertion(rdfs:label :city "ContactCity")
  DataPropertyDomain(:city :ContactInformation)
  DataPropertyRange(:city xs:string)
AnnotationAssertion(rdfs:label :company "ContactCompany")
  DataPropertyDomain(:company :ContactInformation)
  DataPropertyRange(:company xs:string)
AnnotationAssertion(rdfs:label :country "ContactCountry")
  DataPropertyDomain(:country :ContactInformation)
  DataPropertyRange(:country xs:string)
AnnotationAssertion(rdfs:label :department "ContactDepartment")
  DataPropertyDomain(:department :ContactInformation)
  DataPropertyRange(:department xs:string)
AnnotationAssertion(rdfs:label :email "ContactEmail")
  DataPropertyDomain(:email :ContactInformation)
  DataPropertyRange(:email xs:string)
AnnotationAssertion(rdfs:label :phone "Contact␣Phone")
  DataPropertyDomain(:phone :ContactInformation)
  DataPropertyRange(:phone xs:string)
AnnotationAssertion(rdfs:label :postalCode "ContactPostalCode")
  DataPropertyDomain(:postalCode :ContactInformation)
  DataPropertyRange(:postalCode xs:string)
AnnotationAssertion(rdfs:label :room "ContactRoom")
  DataPropertyDomain(:room :ContactInformation)
  DataPropertyRange(:room xs:string)
AnnotationAssertion(rdfs:label :state "ContactState")
  DataPropertyDomain(:state :ContactInformation)
  DataPropertyRange(:state xs:string)
AnnotationAssertion(rdfs:label :street "ContactStreet")
```

```
    DataPropertyDomain(:street :ContactInformation)
    DataPropertyRange(:street xs:string)
  AnnotationAssertion(rdfs:label :url "ContactURL")
    DataPropertyDomain(:url :ContactInformation)
    DataPropertyRange(:url xs:anyURI)
  )
)
```

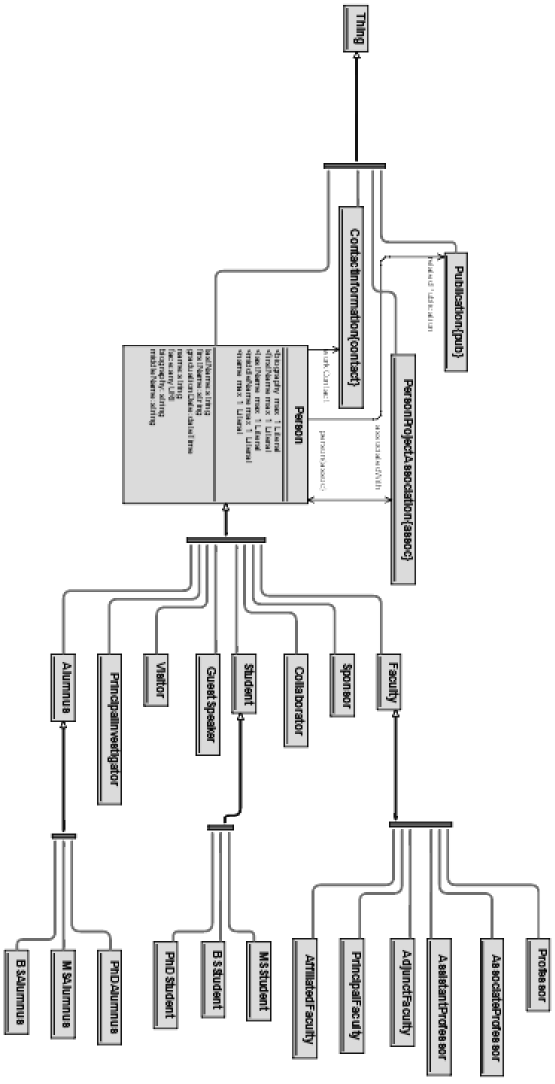


Figure 2.2: Graphical representation of an ontology
Slika 2.2: Grafička reprezentacija ontologije

### 2.3.3 Script languages

Script languages are designed to be interpreted by some virtual machine integrated in a specific environment as an extension of the basic language. Illustrative examples are JavaScript and PHP which are extensions of HTML for writing client and server side applications, respectively. However, purpose of Script languages differs from general purpose such is Python to domain-specific purpose (e.g. JavaScript) while some domain specific script languages such is PHP aim to broader usage. Concerning the paradigm, most of script languages combine paradigms.

Implementation of QuickSort algorithm in PHP is given in Listing 2.8.

Listing 2.8: *QuickSort* implemented in PHP
Listing 2.8: *QuickSort* algoritam implementiran u PHP-u

```
function quicksort($arr){
  $loe = $gt = array();
  if(count($arr) < 2){
    return $arr;
  }
  $pivot_key = key($arr);
  $pivot = array_shift($arr);
  foreach($arr as $val){
    if($val <= $pivot){
      $loe[] = $val;
    }elseif ($val > $pivot){
      $gt[] = $val;
    }
  }
  return array_merge( quicksort($loe),
            array($pivot_key=>$pivot),
            quicksort($gt)
          );
}

$arr = array(1, 3, 5, 7, 9, 8, 6, 4, 2);
$arr = quicksort($arr);
echo implode(',',$arr);
```

### 2.3.4 Legacy languages

Legacy languages were created in 1950s and 1960s. These languages are primarily imperative, unstructured languages, but during the decades they were enriched and extended by language constructs to support structural and object oriented programming, still keeping some traditional constructs and characteristics. Observing contemporary versions of these languages they could be categorised as multi-paradigm languages.

These languages are still present in current projects. Usually, components written in these languages are integrated with modern components and maintained. These components are in the most projects *black boxes* concerning quality analysis techniques because the tool support is very weak. In the best case quality analyses are done by usage of separate tool dedicated to the specific language. Therefore, the results are not very usable or comparable considering overall picture of the project or product.

COBOL: Common Business-Oriented Language is nowadays one of the most present legacy languages. It has all characteristic of language specific for business domain. Another illustrative examples are FORTRAN: Formula Translation designed for scientific and engineering applications, and BASIC: Beginner's All-purpose Symbolic Instruction Code. These are general-purpose languages.

An illustrative example of code written in COBOL is given in Listing 2.9. It contains an implementation of *QuickSort* algorithm. Realisation of *QuickSort* in traditional COBOL would require additional space for simulation of recursion because COBOL originally did not support it, but nowadays it is supported.

Listing 2.9: *QuickSort* implemented in COBOL
Listing 2.9: *QuickSort* implementiran COBOL-u

```
IDENTIFICATION DIVISION.
PROGRAM-ID. quicksort RECURSIVE.

  DATA DIVISION.
    LOCAL-STORAGE SECTION.
    01  temp                    PIC S9(8).

    01  pivot                   PIC S9(8).
  01  left-most-idx          PIC 9(5).
  01  right-most-idx         PIC 9(5).

  01  left-idx               PIC 9(5).
  01  right-idx              PIC 9(5).

  LINKAGE SECTION.
  78  Arr-Length             VALUE 50.

  01  arr-area.
  03  arr                PIC S9(8) OCCURS Arr-Length TIMES.

  01  left-val               PIC 9(5).
  01  right-val              PIC 9(5).

PROCEDURE DIVISION USING REFERENCE arr-area, OPTIONAL left-val,
    OPTIONAL right-val.

      IF left-val IS OMITTED OR right-val IS OMITTED
        MOVE 1 TO left-most-idx, left-idx
          MOVE Arr-Length TO right-most-idx, right-idx
      ELSE
        MOVE left-val TO left-most-idx, left-idx
          MOVE right-val TO right-most-idx, right-idx
      END-IF

      IF right-most-idx - left-most-idx < 1
        GOBACK
      END-IF

      COMPUTE pivot = arr ((left-most-idx + right-most-idx) / 2)

      PERFORM UNTIL left-idx > right-idx
        PERFORM VARYING left-idx FROM left-idx BY 1
          UNTIL arr (left-idx) >= pivot
          END-PERFORM

          PERFORM VARYING right-idx FROM right-idx BY -1
            UNTIL arr (right-idx) <= pivot
          END-PERFORM

    IF left-idx <= right-idx
            MOVE arr (left-idx) TO temp
              MOVE arr (right-idx) TO arr (left-idx)
              MOVE temp TO arr (right-idx)
      ADD 1 TO left-idx
              SUBTRACT 1 FROM right-idx
            END-IF
        END-PERFORM

          CALL "quicksort" USING REFERENCE arr-area,
            CONTENT left-most-idx, right-idx
          CALL "quicksort" USING REFERENCE arr-area,
            CONTENT left-idx, right-most-idx

        GOBACK
          .
```

## 2.4 Summary

This chapter decribed terminology and background related to software quality monitoring. Software quality expresses the level to which a software product satisfies the needs of its users. Software quality is observed through quality attributes by application of static or dynamic analysis techniques. Dynamic analysis requires execution of the program, while static analysis techniques (e.g. software metrics, software networks analysis, code clone detection and structural changes analysis) are applicable on static intermediate representation of the source code.

Input computer (or programming) language, or more precisely, programming paradigm and style, have strong influence on implementation of static analysers. This chapter provided terminology related to computer and programming languages, categorisation of computer languages, described characteristic ones representing specific categories with focus on languages supported in SSQSA framework.

# Chapter 3

# Justification of SSQSA Concept

This chapter describes the starting motivation for the development of the first tool in SSQSA (Set of Software Quality Static Analyzers framework), software metrics tool called SMIILE (Software Metrics Independent on Input LanguagE), justifying it by preliminary research on weaknesses of available software metric tools at that point of time. Furthermore this chapter provides state of the art in specific fields related to SSQSA framework. Finally, review of related software solutions is provided in the last section of this chapter.

## 3.1 Motivation

Systematic application of static analysis techniques can significantly improve the quality of a software product. Software metrics are one of the basic techniques of static analysis. Appropriate tool support for software metric calculation, as well as static analysis in general, may constitute important step toward a success of software projects in general. Preliminary investigation of the state of the art in the field shows that there is no wider acceptance of these techniques in real life product quality monitoring. The main problem in wider application of software metrics and other static analysis techniques lays in limitations and inappropriateness of involved tools. Tools are generally not independent on input programming language or underlying platform, different tools often give inconsistent results for the same metric algorithm, tools usually support only a selection of possible metrics, support for object-oriented metrics is still weak, tools usually do not give explanations or suggestions of possible improvements of code etc.

Motivation for development of a new framework begins with the intention to fulfil gaps in the field of systematic application of software metrics by improving characteristics of software metric tools. Review of the weaknesses of software metric tools which is made as a part of preliminary research to whole SSQSA project is presented in section 3.2.

The general idea for the solution originated from a language-independent software metrics tool SMIILE [Rakić and Budimac, 2011]. It was based on eCST [Rakić and Budimac, 2011] as a universal intermediate representation of the source code. Overview of the usage of different intermediate representations in static analysis and related techniques is given in section 3.3.

The framework was gradually extended by integration of new tools by redirecting them to use eCST as an intermediate representation. In this way the Set of Software Quality Static Analyzers (SSQSA) framework was built up to meet the described goals. Review of related software solutions is provided in chapter 3.4.

## 3.2 Preliminary investigation

In this subsection the justification of starting goals is provided. Preliminary exploration of the problems existing in the field of application of software metrics in practice [Rakić and Budimac, 2010] shows 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 [Lincke et al., 2008], [Lanza and Marinescu, 2006].

Following described assumptions, in order to determine certain weaknesses an inspection of the available tools has been done. Analysis included more than 20 tools, with six of them chosen as representative examples. The tools were analyzed with respect to two groups of criteria. The first group was related to the possible broadness of application of a tool and to the nature and structure of the software product being measured. Second group of the criteria referred to the way of storing and processing the product history and metric results by observed tools.

**The first group of criteria** consists of: platform independence, input language independence, and a list of supported considered:

- LOC: Lines of Code (any metrics from the LOC family: SLOC, CLOC, etc.),
- CC: Cyclomatic Complexity,
- H: Halstead metrics,
- OO: object-oriented metrics (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 to the list above, then the criteria is marked with a '+').

From the described analysis the following important conclusions can be made:

- 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.

| Tool | Producer | Platform indep. | Language indep. | Supported metrics | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | LOC | CC | H | OO | Others |
| SLOCCount | D. Wheeler [a] | - | + | + | - | - | - | - |
| Code Counter Pro | Geronesoft [b] | - | + | + | - | - | - | - |
| Source Monitor | Campwood Software [c] | - | - | + | - | - | + | - |
| Understand | ScientificToolworks [d] | + | - | + | + | - | - | - |
| RSM | MSquared[e] Technologies | + | - | + | + | + | - | - |
| Krakatau | Power Software [f] | - | +* | + | + | + | + | - |

Table 3.1: Overview of software metric tools by the first group of observed characteristics
Tabela 3.1: Pregled oruđa za softversku metriku prema prvoj grupi posmatranih karakteristika

[a] David A. Wheeler,SLOCCount User's Guide, online 2015, http://www.dwheeler.com/sloccount/
[b] Code Counter Pro, online 2015, http://www.geronesoft.com/
[c] SourceMonitor, online 2015, http://www.campwoodsw.com/sourcemonitor.html
[d] Understand, Scientific Toolworks online 2015, http://www.scitools.com
[e] RSM, MSquared Technologies, online 2015, http://msquaredtechnologies.com/
[f] Krakatau Suite Management Overview, online 2015, http://www.powersoftware.com/

  - 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 legacy languages, where 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 in current software development.

**The second group of criteria** was related to storing, processing, and interpreting the calculated metric results by the given tools through the history of the product by involving versions of the source code.

The criteria were (table **??**):

- the history of the source code storing facility;
- the metric results storing facility;
- visualization of the calculated values;
- an interpretation of the calculated values including suggestions for improvements.

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 metric results. Recommendations for an improvement, or even the automation of an improvement based on the obtained metrics results, would be significant contribution to the real practical usability of software metrics.

Currently, situation in the field of application of the software metrics and supporting tools in software product development is similar to described one. During last years, support for design and object-oriented

| Tool | Producer | Source code history | Metrics storage | Graphical represent. | Interpretat. & recommend. |
|---|---|---|---|---|---|
| SLOCCount | D. Wheeler[a] | - | + | - | - |
| Code Counter Pro | Geronesoft [b] | - | + | - | - |
| Source Monitor | Campwood Software [c] | - | + | + | - |
| Understand | ScientificToolworks [d] | - | + | - | - |
| RSM | MSquared Technologies [e] | + | + | - | - |
| Krakatau | Power Software [f] | - | + | + | - |

Table 3.2: Overview of software metric tools by the second group of observed characteristics
Tabela 3.2: Pregled oruđa za softversku metriku prema drugoj grupi posmatranih karakteristika

[a]David A. Wheeler,SLOCCount User's Guide, online 2015, http://www.dwheeler.com/sloccount/
[b]Code Counter Pro, online 2015, http://www.geronesoft.com/
[c]SourceMonitor, Campwood Software, online 2015, http://www.campwoodsw.com/sourcemonitor.html
[d]Understand, Scientific Toolworks (SciTools), online 2010, http://www.scitools.com
[e]RSM, MSquared Technologies, online 2015, http://msquaredtechnologies.com/
[f]Krakatau Suite Management Overview, Power Software, online 2015, http://www.powersoftware.com/

metrics was improved, but still tools limited only to specific languages are important issue. Furthermore, integration of software metrics tools with Integrated Development Environments (IDE) and versioning systems solved the problem with storing of development history, visualization, etc. Problem of analysis of the gained information is still weak, but some progress has been made. One of tools that prove this is the SQUALE: Software QUALity Enhancement[1][Bergel et al., 2009]. SQUALE applies new quality models, aggregates information contained in gained numerical values into high level quality factors and provide users with more useful feedback. This framework is designed to support quality enhancement in the multilingual projects, but it has a potential weak point: collecting of the numerical values is done by the usage of various language-specific tools which can reduce the consistency and credibility of the overall feedback.

Nowadays, 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 are 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 [Lincke et al., 2008], [Novak and Rakić, 2010]. One of the reasons for this is the fact that the rule for metrics calculations could be differently interpreted and implemented with different tools [Scotto et al., 2006]. This confirms a possibility of mentioned weakness of the SQUALE framework. On the other hand, SSQSA's approach is to use a common internal representation of the source code and meta-model for majority of programming languages that represents a basis for metrics calculation. Such an approach enables the same metrics calculation algorithms across different programming languages.

Following this basic idea, a preliminary research was conducted. It was based on an analysis of possibility to implement metric calculation algorithms uniquely for different languages. Research methodology was directed on splitting research subject in two segments:

[1]SQUALE: Software QUALity Enhancement, online 2015, http://www.squale.org/

- Comparative overview how is one metric applied on different languages. This exploration has been repeated or broad spectrum of metrics;
- Comparative exploration on application of wide spectrum of metrics on a single language. This observation was repeated on all languages included in the research.

Focus of the research was on metric calculation algorithms and preconditions for their implementations with respect to observed language. Another aspect of the research was tool support for observed combination of language and metric. Main contribution of conducted research was basic concept for designing new, language independent intermediate representation of source code used in development SMIILE tool.

It was demonstrated on more than 10 representative input languages that the SMIILE is a language independent for currently implemented software metrics (section 4.3.2.1). This means that for each metric algorithm one implementation for all supported languages was enough. The process of the metric calculation 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, but still eCST contains enough information for these calculations (e.g. the LOC metric based on lines data of the first and the last token below the universal node). Moreover, suitable internal representation of the source code and its design has been derived from eCST which is the basis for the calculation of design and object-oriented metrics calculation.

## 3.3 Intermediate representations

The history of intermediate program representations is long - it starts more than half of the century ago. Basic goal at that time was to represent the program with the goal of cross-platform software migration. Nowadays, intermediate program representations are used in compiler engineering, reverse engineering, software modernization, etc. Intermediate representations vary from the intermediate code written in some intermediate languages to the full meta-model for representation of different aspects of the program. Sometimes these representations are used to meet language independency of some specific analysis. Also there are some approaches tending to reach language independency of full functionality based on specific intermediate program representation.

Intermediate languages are *pivot* languages and they are widely used in source code translation and analysis. Some intermediate languages are initially designed as domain-specific languages for program representation to fulfill specific criteria. On the other hand, some general purpose languages are used as intermediate languages. For example C, as an abstraction of assembly language, is used as an intermediate language in the implementations of some other languages (Eiffel, Haskell, etc.).

Program translations can be done between the languages on the same or different levels of abstractions, in different directions, and with various aims. Sometimes, source code is only translated to an intermediate language to enable some specific analysis. For example ALF language [Gustafsson et al., 2009] is used as an intermediate representation of source code written in different languages (C, PowerPC assembly, etc.) to enable static timing analysis. More precisely this language enables control-analyses which is precondition of WCET: Worst Case Execution Time estimation [Lisper, 2014], [Wilhelm et al., 2008]. Even the language is well designed and could represent code written in any language, translation process from source languages to ALF is not language independent.

In compiler engineering intermediate languages are used in compiler construction, decompiling or in source to source translations. These translations usually encompass some analyses, transformations, optimisations, and refinements of the translated code.

Decompiler will translate low-level (assembly) code to some higher-level language in the reverse engineering process. The WSL: Wide Spectrum Language [2] is used for the intermediate representation of software programs in translating legacy to maintainable code (eg. assembly code to C/COBOL code). The main characteristic of WSL is a formal background and the ap-

[2]WSL, 2014 http://www.smltd.com/wsl.htm

plication of formal transformations of code internally represented by using abstract syntax trees. WSL is meant to be independent of programming languages. However, 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 to WSL, approach based on eCST supports a broader scope of languages and metrics (including object-oriented).

Source to source translation (transcompilation) usually means translation between two languages of higher level of abstraction. For example, LLVM: Low-Level Virtual Machine [3] translates any of languages supported by GCC compiler[4] (Pascal, Modula-2, Modula-3, C, C++, Java, Fortran, etc.) to C, C++ and MSIL: Microsoft Intermediate Language. MSIL is also an intermediate language and is also known as CIL: Common Intermediate Language. LLVM uses an intermediate language at the level of human-readable assembly language. [Lattner and Adve, 2004]

Represent of intermediate languages mostly used in compiler engineering is byte-code (e.g. Java byte code). Source code is translated to this intermediate representation which is specific for interpreter and later interpreted by a virtual machine. Another specific category of languages are higher level assembly languages. They are at the similar level of abstraction as a byte code but this code is usually further translated to executable machine code. One such language is already mentioned CIL which is used as a common intermediate languages for all compilers in .NET framework. Furthermore, XTRAN[5] environment includes various analysers relying on intermediate representation written in a C-like intermediate rule-based language. Contrary to all mentioned intermediate languages, eCST is intermediate representation based on syntax tree.

Another intermediate representation usually used in compiler construction, but also in static analysis and related techniques is a syntax tree. Mentioned GCC compiler uses: GENERIC (an intermediate language), and

[3]LLVM: Low-Level Virtual Machine, online 2015, http://llvm.org/

[4]GCC: GNU Compiler Collection, online 2015, https://gcc.gnu.org/

[5]XTRAN, online 2015, http://www.xtran-llc.com/xtran.html

GIMPLE (an internal abstract syntax tree representation) [Merrill, 2003]. Both representations are language independent, idea is similar to idea behind SSQSA and eCST, but results are not really comparable. Areas of application of GCC compiler and SSQSA are complementary. While GCC is oriented to compiling the code and possible static analyses techniques to support it, SSQSA expects compilable code, generates internal representation of it without additional checks and applies supported analyses which are complementar to ones provided by GCC compiler and supporting tools.

Syntax trees, abstract or concrete, are broadly used in numerous fields of software engineering. Abstract Syntax Tree (AST) is used as a representation of source code usually in static analysis which includes different techniques for code clone detection, software metrics calculation, changes monitoring, etc. In [Fischer et al., 2007] the role of AST as a representation of model in Model Driven Engineering is described. Even if the construction of AST is language independent, the content of these trees is always strongly related to language syntax. That can be clearly concluded from all papers related to usage of AST referred in this thesis.

In [Baxter et al., 1998] and [Ducasse et al., 1999] authors use abstract syntax trees for representation of the source code for duplicated code analysis. Those trees have some additional features designed for easier implementation of the algorithm for comparison. [Koschke et al., 2006] propose similar but fresh idea for code clone detection using abstract syntax suffix trees.

Software metrics tools often use AST as an internal representation. The development of ATHENA [Christodoulakis et al., 1989], CodeSquale[6], and many other tools are based on AST representation of the source code. Additionally, graph representation are often used to represent facts on lower or higher level of abstraction. An illustrative example is Bauhaus [Raza et al., 2006] tool which combines two intermediate representations: IML: InterMediate Language, and RFG: Resource Flow Graph. Following the trend of combining several internal representations at different levels of abstraction in one complex internal representation, meta-models have been

[6]CodeSquale, 2015 http://code.google.com/p/codesquale/

introduced. Meta-models tend to include all required information needed for intended analyses. FAMIX is a very prospective family of meta-models developed as a part of MOOSE platform[7]. More information about mentioned tools can be found in the following section (section 3.4) dedicated to related software solutions.

Finally, in parallel with efforts to establish eCST representation and SSQSA architecture, OMG: Object Management Group [8] was working on *modernization meta-models*[9]. They propose standard meta-models based on similar ideas [Pérez-Castillo et al., 2011]. ASTM: Abstract Syntax Tree Meta-model is conceptually similar to our eCST representation. Its idea is usage of common imaginary nodes in enriching AST and it represents program below the function level. KDM: Knowledge Discovery Meta-model by the level of represented data corresponds to some aspects of network representation used in SSQSA. It represents the program on the high (interface) level. This is a good confirmation that approach applied in the development of SSQSA framework has a future, primarily because these meta-models are products of standardization initiatives. Still, there are some differences in two approaches which give validity to our research direction. The main one is that OMG tends to establish very broad set of imaginary nodes which differs through generations of programming languages. In SSQSA framework there is a tendency to minimize this set and to keep the universality of the nodes whenever possible. This is done by adding nodes at the higher levels first and only if necessary adding them in deeper levels and by sharing the nodes between generations. This minimization and universality gives simplicity to implementation of the requested algorithms and provides easier maintenance of the catalogue of nodes.

[7]MOOSE, 2014 http://moosetechnology.org/
[8]OMG: Object Management Group, 2014 http://www.omg.org/
[9]Architecture-driven modernisation, 2014 http://adm.omg.org/

## 3.4 Related software solutions

This section reviews available software solutions developed with similar goals, following similar ideas, or applying similar approaches. Furthermore, each analyzed software solution i compared with SSQSA framework.

One of the earliest attempts to reach language independency of software metrics tool is made by development of the **ATHENA** software certification tool [Christodoulakis et al., 1989]. This tool for assessing the quality of software was based on the parsers that generate abstract syntax trees as a representation of a source code. Used parsers were manually implemented for each language to be supported and algorithms for calculation of software metrics were partially built in parser implementation. 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 to generate parsers by parser generator in order to automate process of adding support for a new language. Furthermore eCST is richer representation then AST. In the end, SSQSA component for software metrics calculations - the SMIILE tool is implemented in Java and therefore it can be used on broader range of platforms.

The development of the **CodeSquale**[10] metrics tool was based on a similar idea to the idea behind SSQSA framework but with usage of AST as intermediate representation. The authors developed the first prototype of the system 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. The final goal was an analyses which i independent of input language. Unfortunately, later results were not published. However, week point of this project was usage of AST for representing the source code. By involving eCST, SSQSA get broader set of algorithms implementable independently of programming language and uniqueness of the

[10]CodeSquale, 2015 http://code.google.com/p/codesquale/

implementation of the algorithms.

[Arbuckle, 2011] presented an interesting approach for the measuring evolution of a multi-language software system. Difficulties related to syntax, semantics, and language paradigms are avoided here by looking directly at relative shared information content. In this approach author measures a relative number of bits of shared binary information between artifacts of consecutive releases. However, SSQSA approach uses source code changes from software repositories to analyze software evolution.

**Rigi** [Kienle and Mller, 2010] is a reverse engineering environment that allows the visual exploration of software systems. Visualisation is based on graph representation of software entities and their relationships. It provides the fact extractors for C, C++, and COBOL, an interactive editor of extracted graphs called Rigiedit, and language independent exchange format known as RSF: Rigi Standard Format. RSF is based on a graph-based data model which is capable to represent software entities at architectural level of abstraction and dependencies between them. In the Rigi architecture, the fact extractors and the graph editor are autonomous components communicating through the exchange of RSF files.

The fact extractors are parsers built with the help of Yacc parser generator. Those parsers identify software entities and their dependencies in a source code and store extracted information in the textual RSF exchange format. For the reverse engineering of software systems written in other languages, users are expected to produce RSF files. The authors of Rigi advocate usage of lightweight fact extractors based only on lexical analysis which produce imprecise, but useful fact bases for analyses of legacy software systems, because those systems are often in the state that the source code cannot be compiled (due to missing files) or contains syntax errors. However, there is no support in Rigi to build parser-based or lightweight fact extractors. In other words, Rigi is capable to analyze and visualize software networks representing software systems written in different programming languages but their extraction is not strictly established and based on language-independency which differs it from SSQSA framework.

**Gupro**: Generic Understanding of PROgram [Ebert et al., 2002] is an integrated workbench to support program understanding of heterogeneous software systems on different levels of granularity. In Gupro, software artifacts are stored in a graph repository which reflects relationships between defined software entities, and abstraction is done by graph queries. Analyzed source code is parsed by parsers generated by the PDL: Parser Description Language [Dahm, 1998]. PDL is parser generator which extends the Yacc parser generator by EBNF syntax and by notational support for translating source code into TGraphs [Ebert et al., 2008].

TGraphs are directed graphs whose nodes and edges may be attributed, typed, and ordered. These graphs are used to conceptually represent software systems: software entities are represented by nodes, relationships among entities are represented by edges, a common type is assigned to similar objects and relationships, and ordering of relationships is expressed by edge order. TGraphs are produced by individualy developed PDL parsers, and therefore the fact extraction in Gupro is not language-independent and unified which decreases the consistency of the analyses. This makes an important weakness of Gupro with respect to the SSQSA framework.

**Bauhaus** [Raza et al., 2006] is a tool suite that supports program understanding and reverse engineering on all layers of abstraction, from source code to architecture. It is capable to analyze programs in Ada, C, C++ and Java. In Bauhaus two separate program representation exist: InterMediate Language (IML) and Resource Flow Graph (RFG) representation. The IML representation is defined by the hierarchy of predefined classes, where each class represents a certain universal programming language construct. IML is generated from source code by a language-specific front-end. While IML represent the system on a very concrete and detailed level, the architectural aspects of the system are modeled by means of RFG. An RFG is a hierarchical graph, which consists of typed nodes and edges.

Nodes represent architecturally relevant elements of software systems. In other words, RFG is, similarly as GDN (that exists in SSQSA), a union of software networks at different levels of abstractions. Using different granularity filters, a different aspects of the architecture can be obtained (call graph, hierarchy of modules, etc.). For C and C++, an RFG is automatically generated from IML representation, whereas for other languages RFG is generated from other intermediate representations (such as Java class files) or compiler supported interfaces (such as Ada Semantic Interface Specification).

Therefore, fact extraction in Bauhaus is not fully language-independent, since RFGs for some languages are not constructed directly from IML. This is an important weakness in comparison to SSQSA framework and its intermediate representations.

XTRAN[11] is another try to achieve language independent environment for various types of analyses. Contrary to SSQSA, whose intermediate structure is based on syntax tree, its intermediate representation is based on a C-like intermediate rule-based language.

It supports several versions of assemblers, C, C++, Java, COBOL, FORTRAN, Pascal, PL/I, several 4th generation languages (SAS, RPG, ...), HTML, and XML. While supported languages may be regarded as analogous to those supported by SSQSA, it does not support functional languages (e.g., Scheme, Erlang, ...) , mixed paradigm languages (e.g., Delphi), and script languages (e.g., PHP), which means that SSQSA is more general.

The goals of XTRAN and SSQSA are however different with respect to analyses they are meant to perform - having a language as its intermediate representation XTRAN is more oriented to syntax-level analyses - construction of function trees, translation from one language to another, location of dead code, cross-reference documents, etc. SSQSA and XTRAN have in common only code-clone analysis and calculation of CC metrics, and in this respect SSQSA is also more general and capable of implementing more analyses than XTRAN.

**MOOSE**[12] is a platform for software and data analysis in reverse engineering and re-engineering of object-oriented software systems. It consists of a repository to store language-independent models of software systems, and provides query and navigation facilities. Moose models are instances of the members of the **FAMIX** meta-models family [Tichelaar et al., 2000a] and capture architectural elements of software systems: defined entities

[11]XTRAN, online 2015, http://www.xtran-llc.com/xtran.html

[12]The MOOSE book, 2014 http://www.themoosebook.org/book

and their dependencies. MOOSE and FAMIX have the most similar general goals to the SSQSA project. Their joint goals were established by the European project FAMOOS [Bär and Ducasse, 1999] and their strength is mainly in language independency. They support OO design at the interface level of abstraction for various input programming languages. Different input languages are supported by separate tools for filling in the meta-model with the information from the input source code [Tichelaar et al., 2000b]. SSQSA approach is more general. It is based on eCST representing all aspects of source code and not only the design. It is thus equally appropriate to support broader set of software metrics. However, it also fully supports procedural languages, including the legacy ones (e.g. COBOL).

## 3.5 Summary

Motivation for development of SSQSA framework started with development of SMIILE tool for calculation of software metrics. The aim of SMIILE tool were improved characteristics with intention to bridge the gap in the field of application of software metrics. This chapter described starting motivation and preliminary investigation of the field. This investigation resulted with conclusion that the main weaknesses are in strong dependency of available tools on input language which is caused by used intermediate representation. Afterwards, chapter contained review of alternative intermediate representations and available related software solution. Final conclusion is that concept used in development of SSQSA framework makes it innovative with planed characteristics.

# Chapter 4

# SSQSA framework

This chapter describes SSQSA framework, the central contribution of the dissertation. *SSQSA overview* (section 4.1) provides architecture with brief overview of the SSQSA components. Section 4.2: *SSQSA intermediate representations* describes the central intermediate representation in SSQSA framework - eCST: enriched Concrete Syntax Tree, but also intermediate representations derived from it. Section 4.3: *SSQSA components* describes constituents of SSQSA framework at different levels: generators and manipulators by intermediate representations, static analysers, higher-level external tool integrated to use results of the SSQSA analysers, etc. Last two sections (4.4 and 4.5) describe two crucial characteristics of SSQSA framework: adaptability and extendability, respectively.

## 4.1 SSQSA overview

Set of Software Quality Static Analyzers (SSQSA) [Rakić et al., 2013b] is the set of software tools for static analysis that are incorporated in the framework developed to target the common aim: consistent software quality analysis. The main characteristic of all integrated tools is the independency of the input computer language. This characteristic gives the tools more generality comparing to the other similar static analyzers. Each of incorporated analyzers can be uniformly applied to any software systems that are written in different computer languages. Furthermore, integration of the analyzers into the framework enables consistent analysis of software product from different viewpoints combining benefits of different techniques of static analysis. This consistency gives the additional value to the framework having in mind the fact that software systems are every day more complex and heterogeneous, consisting of many components usually developed using the different programming languages.

Language independency is achieved by enriched Concrete Syntax Tree (eCST) [Rakić and Budimac, 2011] that is used as an intermediate representation of the source code representations (section 4.2). It is unique representation of a software code regardless of the language in which it was written. When this representation of input code is available it is possible to use a single implementation of the analysis algorithms. Consequently, for each new input language the whole set of integrated analyses is readily available.

Currently, SSQSA supports some representative input languages: Java, C#, Delphi, Modula-2, Pascal, C, COBOL, Erlang, Scheme, Python, PHP, JavaScript, OWL, WSL, and Tempura in different stages of support. Supported languages are designed to be used in different phases of software development and belong to variety of paradigms. Based on the experience, adding support for a new input language is a straightforward activity (section 4.4)[Kolek et al., 2013]. After adding a new language all available analysis are immediately applicable for that language. Furthermore, in order to prove the concept the framework consists of some fully functional tools. It is also possible to add new analysis. In this case a single implementation of the analysis algorithm is enough for all supported languages, which means that after integration of a new analysis in the framework it is immediately applicable to all supported languages.

Components and tools in SSQSA framework can be divided in three categories and layered in five levels (see Figure 4.1) according to their role in the framework.

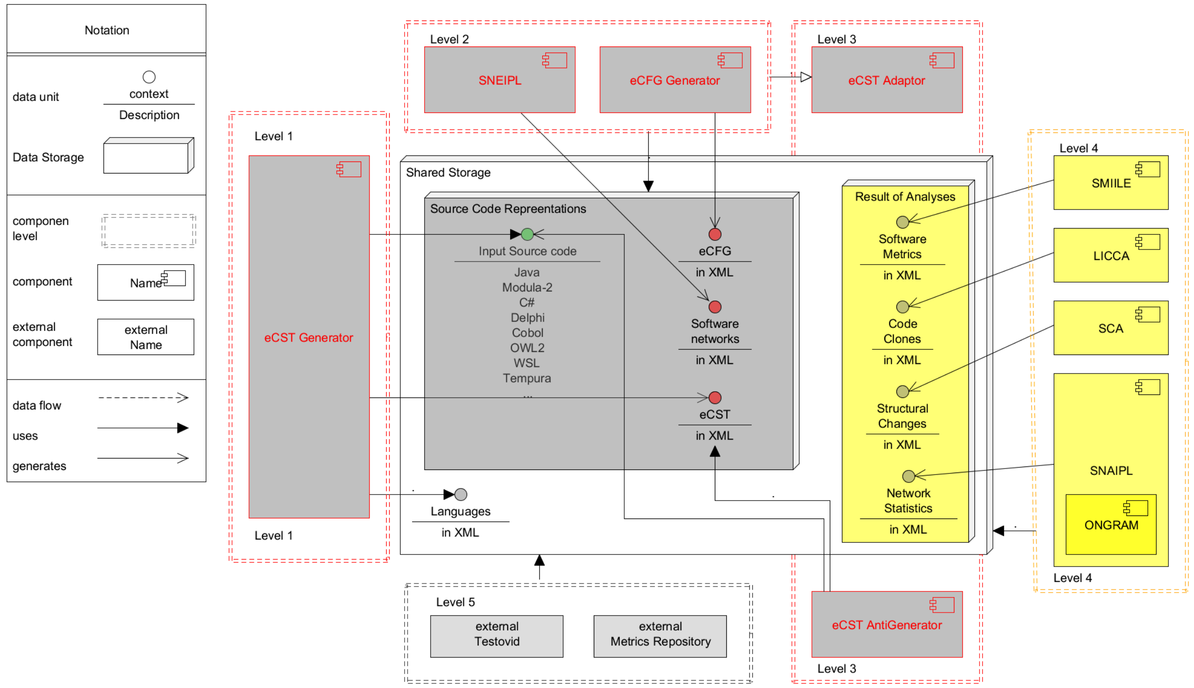


Figure 4.1: Architecture of the SSQSA framework
Slika 4.1: Arhitektura SSQSA okvira

**First category** of SSQSA components consists of first three levels of components with common responsibility: generation and manipulation of internal representations.

**First level.** Central role in the framework plays eCST representation of the source code as a basis for language independency. As a prerequisite for all analyses to be applied independently of input language this internal representation of the source code has to be generated. Therefore, generator of the enriched Concrete Syntax Tree (eCSTGenerator) is placed at the first level and its execution is the first step in using the framework.

**Second level** in the framework consists of components responsible for generation of additional internal representations of the source code to make analyses easier and more comfortable. All these representations are generated from eCST and therefore they retain language independency.

**eCFG Generator:** enriched Control Flow Graph Generator is in charge of generating language independent Control-Flow Graph based on eCST representation of the source code.

**eGDN Generator:** Software Networks Extractor Independent of Programming Language [Savić et al., 2012], [Savić et al., 2014] is a component that provides software network generation to represent relationship between software entities.

**Third level** consists of components responsible for manipulation of eCST (section 4.3.1.3).

**eCST Adaptor** : enriched Concrete Syntax Tree Adaptor is a component that is currently under development. Its responsibility is a manipulation of eCST in order to remove detected imperfections of eCST. It provides all functionalities related to adaptations of eCST to certain algorithm implementations.

**eCST AntiGenerator:** component that re-generates a source code from the eCST representation.

**Second category** of components (static analysers) corresponds to fourth level. On an eCST and its derived representations of the source code, needed analyses algorithms are implemented uniformly. The following fourth-level tools are examples of fully integrated analyzers in the SSQSA framework (section 4.3.2.

**SMIILE:** Software Metrics Independent of Input LanguagE [Rakić and Budimac, 2011];

**LICCA:** Language Independent Code Clone Analysis;

**SSCA:** Software Structure Change Analyzer [Gerlec et al., 2012];

**SNAIPL:** Software Network Analyser Independent of Programming Language [Savić et al., 2012], [Savić et al., 2014];

> **ONGRAM:** Ontology analyzer based on software metrics and network analysis [Savić et al., 2013].

**Third category** of components consisting of external tools integrated in SSQSA, corresponds to fifth level. Some of tools on the fifth level are still in development phase. These tools are using internal representations but also rely on results of basic analysis techniques implemented on a third level. These are:

**Testovid:** a tool for automated assessment of students programming solutions [Pribela et al., 2011], [Pribela et al., 2012];

**Third part repository integration:** SMIILE tool is integrated with existing external software metrics repository so that its results could be imported in it [Rakić et al., 2011].

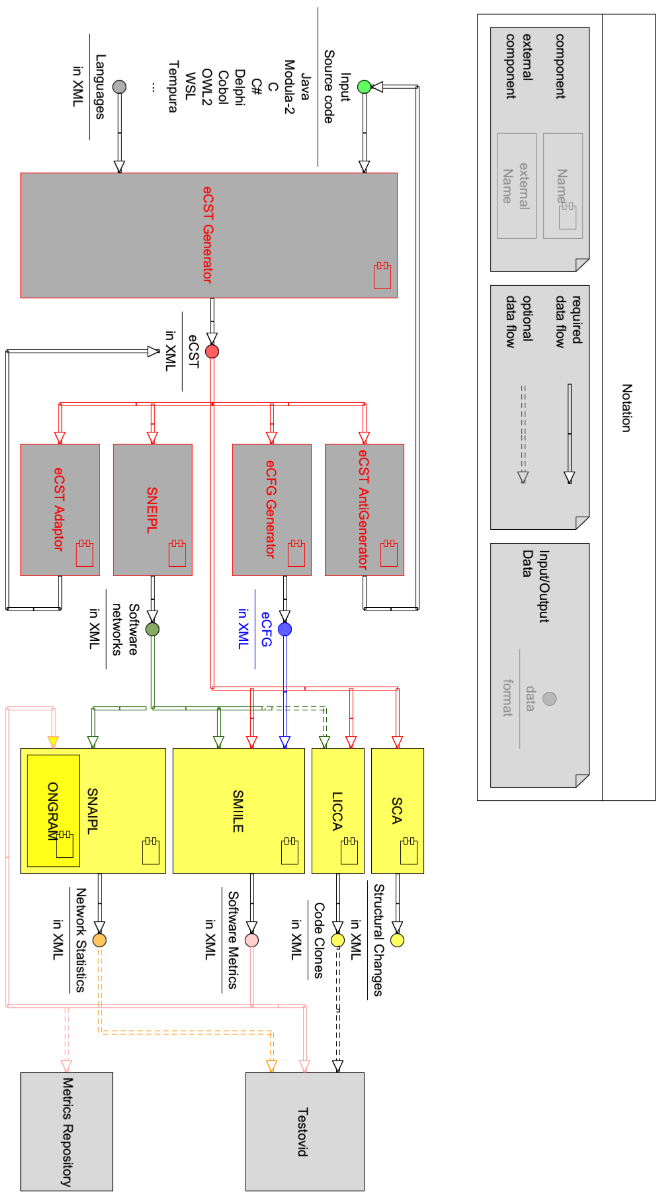


Figure 4.2: *Pipes and filters* architecture of the SSQSA framework
Slika 4.2: *Pipes and filters* architectura SSQSA okvira.

Components do not have direct communication. They communicate by exchanging XML documents using pipes and filters (Figure 4.2). For these purposes, results of all components that may be used by some other component are extracted to XML document and stored to shared data storage.

Described approach supply the SSQSA framework with two crucial features [Kolek et al., 2013]:

**Adaptability** or flexibility in supporting a new language and

**Extendability** or scalability in supporting a new analysis.

In that way the consistency of results and reliable analysis of software product quality, regardless of the input language has been achieved. This is the ultimate contribution of the SSQSA framework.

## 4.2 SSQSA Internal source code representation

Enriched Concrete Syntax Tree (eCST) is a primary internal representation of the SSQSA framework. Each compilation unit is represented as one eCST. It serves as a starting point for generation of derived internal representations representing input program at two levels of abstraction: (1) eCFG enriched Control-Flow Graph generated from eCST as low-level code representation, and (2) as each of these trees are independent, software networks are used to reflect higher level connections between compilation units. Consequently, eCFG and software networks are secondary internal representations.

This section describes mentioned code representations used in SSQSA.

### 4.2.1 eCST: enriched Concrete Syntax Tree

As described in section 2.2.1 there are two tree representations of a source code widely used in static analysis: abstract syntax tree (AST) and concrete syntax tree (CST). We introduce eCST as an innovation in a field of tree representation of the source code. Enriched Concrete Syntax Tree (eCST) represents a union of concepts used in abstract and concrete syntax. It consists of both:

- abstraction of the program structure needed for most of program analyses;
- all concrete program elements, sometimes needed for sophisticated program analyses.

Figure 4.3 represents described similarities and differences between these three types of syntax trees on the common example (figure taken from [Savić et al., 2014]).

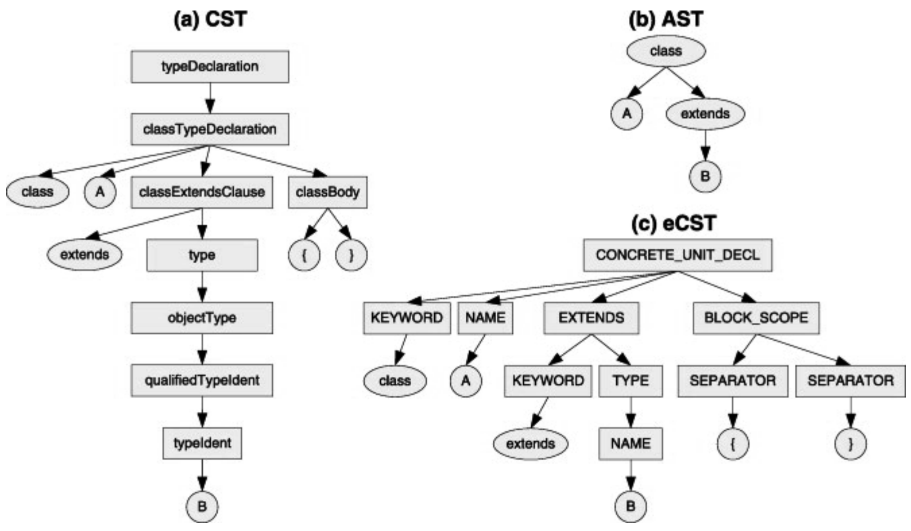


Figure 4.3: Java-like fragment of source code class A extends B represented by (a) CST: Concrete Syntax Tree , (b) AST: Abstract Syntax Tree, and (c)eCST: enriched Concrete Syntax Tree
Slika 4.3: Fragment izvornog koda class A extends B napisan u jeziku poput Jave, predstavljen (a) konkretnim sintaksnim stablom (CST), (b) apstraktnim sintaksnim stablom (AST) i obogaćenim konkretnim sintaksnim stablom (eCST)

Enriched Concrete Syntax Tree (eCST) [Rakić and Budimac, 2011] is the classical concrete syntax tree (or a parse tree) enriched with so-called universal nodes that describe elements of language semantics. Here, the concept of imaginary nodes used in AST building has been involved. Universal nodes can be described as specific imaginary nodes that are carefully chosen (section 4.2.1.2) to be uniformly used for all input languages. This characteristic gives language independency to eCST and correspondingly to all tools using it.

As concept of trees has background in graph theory where tree is connected graph without cycles, we can introduce the following semi-formal definition of eCST:

> Let T = (V, E, i) be a tree where V is set of concrete, universal, and optionally imaginary nodes; and edges $E \subseteq VxV$ connect two nodes $v_i, v_j \in V$ iff between $v_j$ and $v_i$ exists parent-child relationship defined by language construct building rules defined by language grammar. Tree T is called *eCST: enriched Concrete Syntax Tree.* Node $v_i \in V$ which has no incoming edges ($\nexists v \in V : (v, v_i) \in E$) represents the root node. Each node $v_i \in V$ which has no outcoming edges ($\nexists v \in V : (v_i, v) \in E$) represents the terminal nodes (leafs), while each node $v_i \in V$ which has out-coming edges ($\exists v \in V : (v_i, v) \in E$) represents the non-terminal node.

An illustrative example is given in Figure 4.4 where two equivalent segment of the source code for declaration of the concrete unit in two languages, class in Java, and module in Modula-2 are represented by the same sub-tree of the eCST. To simplify the graphical representation some keywords are intentionally omitted from the picture, but their inclusion would not affect the essence of the example.

```
Modula-2:

IMPLEMENTATION MODULE A;
  VAR
     gv: T;

  PROCEDURE p(fp: T): RT;
  BEGIN
   …
  END p;
END A.
```

```
Java:

class A
{
  T gv;

  RT p(T fp)
  {
    …
  }
}
```

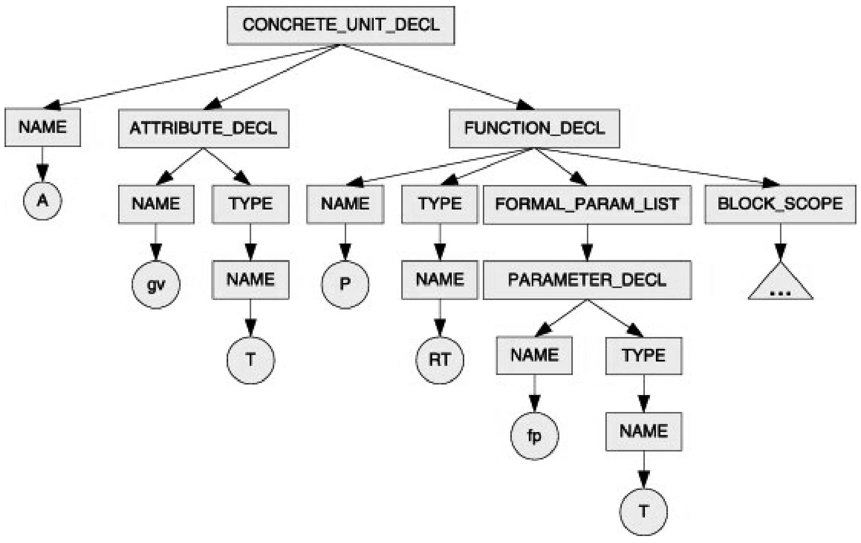


Figure 4.4: Fragment of source code written in Java and Modula-2 represented by common tree. Picture taken from [Savić et al., 2014]
Slika 4.4:Fragment izvornog koda napisan u Javi i Moduli-2 predstavljen zajedničkim stablom. Slika preuzeta iz [Savić et al., 2014]

Figure 4.5 illustrates the reasons for simplification of the tree in the previous example. eCST shown in the figure represents a simple Java class containing only one method - implementation of the BubbleSort algorithm (Listing 4.1).

Listing 4.1: *BubbleSort* implemented in Java
Listing 4.1: *BubbleSort* algoritam implementiran u Javi

```
class bs{
  public static int BubbleSort( int[] num ){
    int j;
    boolean flag;
    flag = true;
    int temp;
    do{
      int count, addElse;
      count = 0;
      addElse = 0;
      flag= false;
      for( j=0;  j < num.length -1;  j++ ){
        if ( num[j] < num[j+1] ){
          temp = num[j];
          num[j] = num[j+1];
          num[j+1] = temp;
          flag = true;
        }
        else{
          count++;
          addElse +=1;
        }
      }
    }while ( flag );
    return j;
  }
}
```

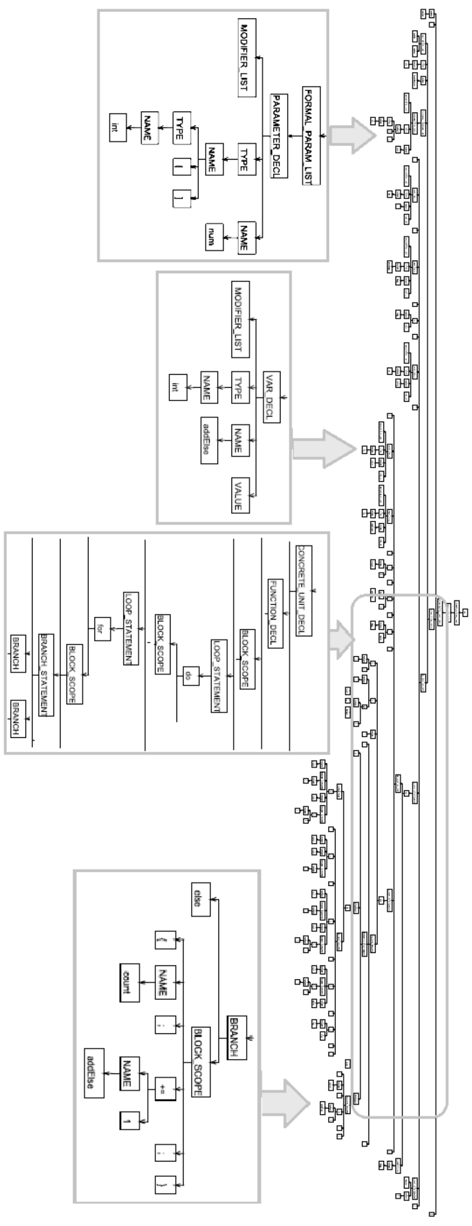


Figure 4.5: eCST representing *BubbleSort* algorithm implemented in Java
Slika 4.5: eCST kojim je prestavljen *BubbleSort* algoritam implementiran u Javi

There are two classes of nodes that always appear in the construction of eCST:

- universal nodes with predefined, language-independent meanings which denote semantic concepts expressed by constructs of the language;
- tokens, elements of the source code, which are leaf nodes of eCSTs. Important characteristic is that every single symbol existing in the source code has to appear in leafs of the corresponding eCST.

Sometimes eCST can contain some imaginary nodes with language-dependent meaning which correspond to a subset of non-terminal symbols in the grammar. Those nodes serve only to retain natural hierarchical structure of language constructs in case that there is no need for universal node corresponding to some non-terminal symbol. These nodes have no meaning for the analysers and could also be eliminated without any consequences.

One universal node can substitute a chain of none or more non-terminal symbols in the CST that is derived through a sequence of unary productions. Therefore an eCST is usually more compact than the corresponding CST, but still more detailed than AST. Furthermore, for our purposes. It is more informative and more useful than both AST and CST: It is complete which means that it contains full source code without skipping any details.

#### 4.2.1.1 Fundamental concept of universal nodes

Imaginary nodes are usually used while building AST to group some logically related units of the source code in its sub-tree. These nodes are imaginary in the sense that they do not represent concrete source code elements and they are not part of the language syntax. They can be introduced freely without following any rule. Usually creator of the grammar or the tree introduce such nodes to satisfy his own needs.

Unification of the imaginary node is a basis on which the concept of eCST is built up. The basic idea is to create the finite set of unified imaginary nodes. These nodes are to be used to annotate the syntax tree in a specific way so that they mark all semantic elements needed for the analyses to be implemented. The same set of nodes is built in the grammar for all languages to be supported. Based on this universality these nodes are called universal nodes. Furthermore, this concept is based on tendency to keep the set of universal nodes as minimal as possible, which additionally differs this approach from other similar ones. (section 4.2.1)

An illustrative example to present important characteristics of the universal node is the one used to mark loops in the source code. In different languages there are: *for*, *do-while*, *while-do*, *repeat-until*, etc. loops with very different syntactical rules. If we use LOOP_STATEMENT universal node to group each of these loops in its sub-tree we will be able to recognize all loops in the source code independently of language and its syntax. Here we can note two levels of the universality. On the first level LOOP_STATEMENT represents loops in all languages, but also, it represents all loops in each of these languages. Figure 4.6 displays eCST representing two loop statements: *do-while* written in Java, and *repeat-until* written in Modula-2 (Figure taken from [Rakić and Budimac, 2011]).

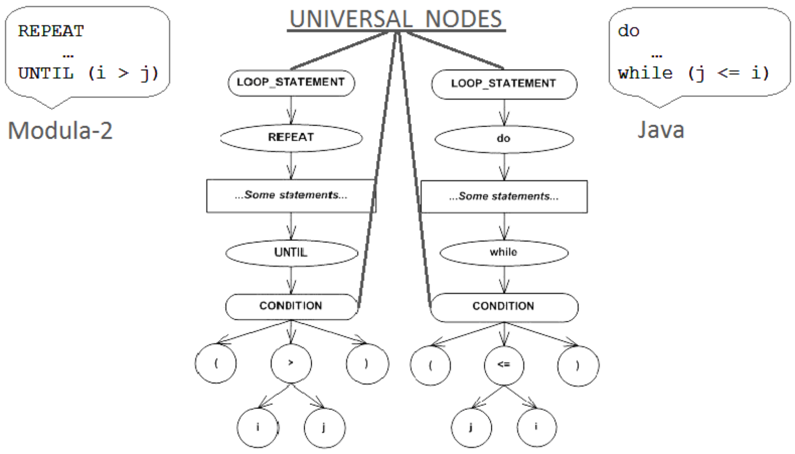


Figure 4.6: Loop statement written in Modula-2 and Java represented by eCST
Slika 4.6: Naredba ponavljanja napisana u Moduli-2 i Javi reprezentovana eCST-om

#### 4.2.1.2 Methodology

Initial set of universal nodes has been derived from nodes used by the first prototype of SMIILE tool [Rakić and Budimac, 2011], [Rakić, 2010]. Therefore, software metrics calculation algorithms were main guidelines for establishing the initial set of universal nodes.

These nodes where defined after the preliminary research and analysis of programming languages and their constructs that are important for metrics calculation. The research included investigation of various computer languages, structure of the code written in these languages, and influence of constructs of these languages on the implementation of different metric algorithms. Initially the set of languages consisted of three representative languages: Modula-2 (represent of procedural and modular languages), Java (represent of object-oriented languages) and COBOL (represent of legacy languages). The first two languages served for creation of the set of nodes, while COBOL was used as a reference for checking the applicability for legacy languages.

First the basic building blocks were identified (packages, units, attributes, functions, blocks, statements, expressions, etc.) and represented by appropriate universal nodes. In the next step statement was refined to loop, branch, jump (also needed for COBOL), and condition. Further refinement of statement categories (e.g. refinement of loop to *for, do-while, while-do, repeat-until, foreach,* etc.) was not necessary and this early decision did not change since then. Note that eCST retains all elements of the source code and the kind of the loop statement is saved in the tree. If needed, it can be used for some subtle or language specific analyses.

At this point the cyclomatic complexity (CC) was implementable for all three languages and the set of universal nodes (containing at the time 14 nodes) proved to be stable during further analyses, testing, and applications. The next step was the refinement of interface-level nodes and introduction of nodes that would enable (a) calculation of design/object-oriented metrics and (b) creation of dependency network between units. For the goal of design metrics a UNIT has been refined to concrete and interface unit (note that the separate node for the abstract unit was not necessary so far). To enable the creation of dependency networks additional nodes to name various entities have been introduced (e.g. types - built-in, declared, or imported).

At this point of time, the set of universal nodes contained 34 nodes and the creation of software networks (as the next important milestone) was possible. The set of universal nodes was fixed, catalogued, documented, and made public for the further usage and inclusion of more languages and more analyses. This represented the end of first phase (creation of the set of nodes), and the emphasis moved to the second (and continual) phase of maintenance. In the process of maintenance some more nodes were needed, as expected. For example, inclusion of Halstead metrics required four new nodes to represent keywords, operators, separators, and directives. On the other hand, structural changes analyzer (SSCA) and code clone analyzer (LICCA) required no additional nodes.

Regarding additional languages, the established set of nodes proved to be stable meaning that no new nodes were necessary.

During the process of creation of (minimal) set of universal nodes, only several ones were dropped in the process. An illustrative example is MAIN_BLOCK_SCOPE. In the beginning, this node was used to mark the main block of the program - a fragment of code that does not belong to any function (procedure, method, etc.) and usually initializes the program for execution. At the same time it existed a BLOCK_SCOPE which denoted any block-scope in the program. After detailed consideration of possible usage of these nodes, the conclusion was made that only one of these nodes is sufficient. Decision was to keep more general one: block scope, having MAIN_BLOCK_SCOPE only as a special case of block-scope. In eCST, when BLOCK_SCOPE does not belong to a sub-tree of some function then block scope node denotes a main block of the program.

The method of inclusion of new nodes in the phase of maintenance nodes follows the prescribed and strict procedure described in sections 4.4 and 4.5. We are aware that new analyses (e.g. data-flow analysis, timing analysis, and program slicing) will probably required introduction of several new nodes, while inclusion of more languages will probably do not enforce inclusion of new nodes.

In other words, regarding the front-end (new computer languages) inclusion of new nodes is not probable. Inclusion of new nodes for the back-end (new analysis algorithms) will be probably needed, but will be kept at the minimal level.

#### 4.2.1.3 Overview of universal nodes

There are two categories of universal nodes: general purpose universal nodes and domain-specific universal nodes. General purpose universal nodes are nodes used by static analysers independently of the domain. The most of the universal nodes belong to this category. Still it is also needed to enable introduction of new universal nodes used for specific purposes which will not be used by all analyzers. These nodes are appearing usually while introducing support for a new, domain specific languages. It is also possible to introduce new universal node which will be used only in the particular domain. Afterwards, the purpose of these new nodes is to be observed from different points of view and finally from the domain-independent point of view. Conclusions of these observations could be very different. Some of the following situations can happen:

- New domain specific universal node has the same purpose in specific domain as already existing general purpose universal node (that are independent of the domain). In that case it can be easily replaced by general purpose universal node.
- New domain specific universal node has the same purpose in the specific domain as the other domain specific universal node in the other domain. In that case these nodes could be unified. Furthermore it can be considered if these nodes could be moved to general usage if their usage is general enough.
- New domain specific node has the application only in the specific domain and remains the domain specific one.

The set of eCST general purpose universal nodes contains 37 nodes and is stable. Still, by extending the framework with new languages and new analysers this set can grow. However, the goal is to keep this set as small as possible. According to their usage in the analyses they can be classified into three categories:

- High-level eCST universal nodes (Table 4.1) denote entities declaration on the architectural level. Interface-level declarations of packages, classes, modules and methods, procedures, functions, etc., as well as explicitly stated high-level relations between them (such as inheritance, instantiation, implementation, etc.)

| Universal Node | Description |
|---|---|
| ATTRIBUTE_DECL | Declaration of class fields, global variables, etc. |
| BLOCK_SCOPE | Root of the sub-tree containing one block scope<br>Block of the source code which does not belong to any function declaration corresponding to main block scope |
| COMPILATION_UNIT | Fragment of a source code recognized by the compiler as a independent unit<br>It usually marks content of an input file |
| CONCRETE_UNIT_DECL | Declaration of a software unit containing implementation (concrete class, implementation module, etc. ) |
| EXTENDS | Inheritance |
| FORMAL_PARAM_LIST | Root of the sub-tree containing one or more formal parameters |
| FUNCTION_DECL | Declaration of a method, procedure, function, etc |
| IMPLEMENTS | Indicates that a concrete unit implements an interface unit |
| IMPORT_DECL | Import of entities declared outside of unit to be used inside it |
| INSTANTIATES | Creating a new instance. |
| INTERFACE_UNIT_DECL | Declaration of a software unit not containing implementation (interface, definition module, etc. ) |
| PACKAGE_DECL | Package/namespace/... |
| PARAMETER_DECL | Declaration of a formal parameter |
| TYPE_DECL | Declaration of a new user/defined type |

Table 4.1: High-level eCST universal nodes
Tabela 4.1: eCST univerzalni čvorovi visokog nivoa

- Middle-level eCST universal nodes (Table 4.2) are those used at the level of entity definition. They appear in the body of the entities and mark individual statements, groups of statements, or parts of statements with appropriate concept expressed by them (jump statement, loop statement, branch statement, condition, import statement, etc.)

| Universal Node | Description |
|---|---|
| ARGUMENT | Argument of the function |
| ARGUMENT_LIST | Root of the sub-tree containing one or more argument of the function |
| BRANCH | Root of the sub-tree containing single branch in a branch statement |
| BRANCH_STATEMENT | Root of the sub-tree containing one or more branches in a branching (branch statement) |
| CONDITION | Root of the sub-tree containing condition for loops/branchings |
| CONST | Declaration of the constant |
| EXPRESSION | Root of the sub-tree containing any expression |
| FUNCTION_CALL | Function call |
| JUMP_STATEMENT | Root of the sub-tree containing jump |
| LOOP_STATEMENT | Root of the sub-tree containing looping (loop statement) |
| STATEMENT | Root of the sub-tree containing any statement |
| VAR_DECL | Declaration of the local variable |

Table 4.2: Middle-level eCST universal nodes
Tabela 4.2: eCST univerzalni čvorovi srednjeg nivoa

- Low-level eCST universal nodes (Table 4.3) are universal nodes that mark individual tokens with appropriate lexical category (keywords, separators, identifiers, comments, etc.).

| Universal Node | Description |
|---|---|
| BUILTIN_TYPE | Type built into the language |
| DIRECTIVE | Directive of a language |
| KEYWORD | Keyword of the language |
| LOGICAL_OPERATOR | Logical operator |
| NAME | Name defined by user |
| OPERATOR | Operator (in general) |
| SEPARATOR | Separator |
| TYPE | Type defined by user |
| COMMENT | Multy-line comment |
| DOC_COMMENT | Documentation comment |
| LINE_COMMENT | Line comment |

Table 4.3: Low-level eCST universal nodes
Tabela 4.3: eCST univerzalni čvorovi niskog nivoa

More details with mapping constructs of characteristic languages to corresponding universal nodes can be found in appendix A.

### 4.2.2 Internal Representations Derived from eCST

Two internal representations of input source code are derived from eCST to be used by analysers in SSQSA framework.

- eCFG: enriched Control-Flow Graph to represent possible program execution paths through the blocks of the program;
- eGDN: enriched General Dependency Network as a union of software networks that reflect dependencies between software entities on different levels of abstraction.

.

#### 4.2.2.1 eCFG: enriched Control-Flow Graph

Enriched Control-Flow Graph (eCFG) is the CFG generated from eCST. The main difference between eCFG and CFG is that eCFG is generated based on detection of universal nodes. Edges in eCFG connect nodes representing universal nodes while sub-trees of universal nodes still keep language-specific constructs. In that way we retain language independency of internal representations. Furthermore, the transformation from eCST to eCFG is language independent.

Figure 4.7 represent the eCFG generated for the BubbleSort method implemented in Java (Listing 4.1). Graph on the figure displays only universal nodes contained in corresponding fragment of eCST and connected during the transformation of eCST to eCFG. Two elliptic nodes (BEGIN_BubbleSort and END_BubbleSort) are representing entry and exit points to the BubbleSort method. These nodes are not universal nodes or part of eCST concept. They are included to play role of real nodes from the set of nodes that can represent entry and exit points of the method or, more generally, function.

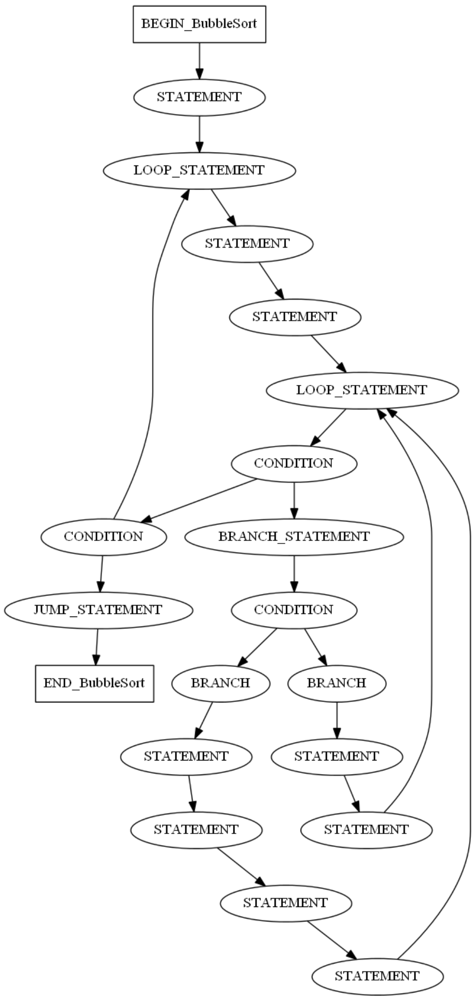


Figure 4.7: eCGF representing *BubbleSort* method
Slika 4.7: eCFG reprezentacija *BubbleSort* metoda

#### 4.2.2.2 eGDN: enriched General Dependency Network

As a part of presented concept software networks are built among the subset of eCST nodes. From a set of eCST representations, each representing one compilation unit, software networks can be extracted independently on input programming language. Extracted networks reflect internal structure of a software system at different levels of abstractions (section 4.3.1.2). In that way generated GDN is called *eGDN: enriched General Dependency Network*.

On figures 4.8 and 4.9 software networks generated for one simple example written in two different languages, Java and C#, are provided. Examples contain equivalent implementation of basic functionality related to a student placed in a class *Student*. Class *Student* extends a class *Person* and implements an interface *IStudent*. Implementations written in both languages consist of 1 package, 5 classes, 1 interface, and 25 methods.

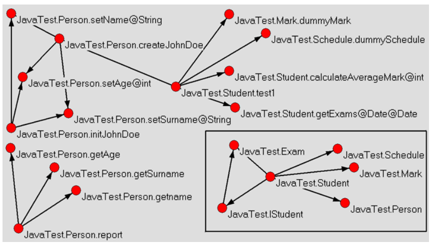

JavaTest.Person.setName@String
JavaTest.Mark.dummyMark
JavaTest.Person.createJohnDoe
JavaTest.Schedule.dummySchedule
JavaTest.Person.setAge@int
JavaTest.Student.calculateAverageMark@int
JavaTest.Student.test1
JavaTest.Student.getExams@Date@Date
JavaTest.Person.setSurname@String
JavaTest.Person.initJohnDoe
JavaTest.Person.getAge
JavaTest.Person.getSurname
JavaTest.Person.getname
JavaTest.Person.report
JavaTest.Exam
JavaTest.Schedule
JavaTest.Mark
JavaTest.Student
JavaTest.IStudent
JavaTest.Person

Figure 4.8: Software network generated for *Student* example written in Java (appendix B, Listing B.1)
Slika 4.8: Softverska mreža generisana za primer *Student* napisan u Javi (appendix B, Listing B.1)

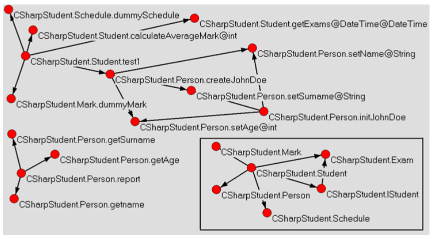

CSharpStudent.Schedule.dummySchedule
CSharpStudent.Student.getExams@DateTime@DateTime
CSharpStudent.Student.calculateAverageMark@int
CSharpStudent.Student.test1
CSharpStudent.Person.setName@String
CSharpStudent.Person.createJohnDoe
CSharpStudent.Person.setSurname@String
CSharpStudent.Mark.dummyMark
CSharpStudent.Person.initJohnDoe
CSharpStudent.Person.setAge@int
CSharpStudent.Person.getSurname
CSharpStudent.Person.getAge
CSharpStudent.Person.report
CSharpStudent.Person.getname
CSharpStudent.Mark
CSharpStudent.Exam
CSharpStudent.Student
CSharpStudent.Person
CSharpStudent.IStudent
CSharpStudent.Schedule

Figure 4.9: Software network generated for *Student* example written in C# (appendix B, Listing B.2)
Slika 4.8: Softverska mreža generisana za primer *Student* napisan u C# (appendix B, Listing B.2)

Source code of the example is provided in appendix B. Listing B.1 contains implementation of *Student* example written in Java, while corresponding eGDN is provided on Figure 4.8. Similarly, Listing B.2 contains source code of *Student* example written in C#, while corresponding eGDN is provided on Figure 4.9. Example and figures taken from [Savić et al., 2012].

Beside keeping semantic equivalence, the aim of the example requires keeping structural equivalence and the same hierarchy among entities. Input languages were carefully selected to satisfy these requirements. As presented on figures 4.8 and 4.9 eGDN Extractor extracted isomorphic software networks.

Software networks for real-life applications are usually very complex. However, simple example of software network representing a real-life project is Static Call Graph of a small Scheme Project (GTK[1] LOC 5.000). This network contains 35 nodes and 18 links which means that 35 functions are defined, and 18 function calls are made. Figure 4.10 taken from [Kolek, 2014].

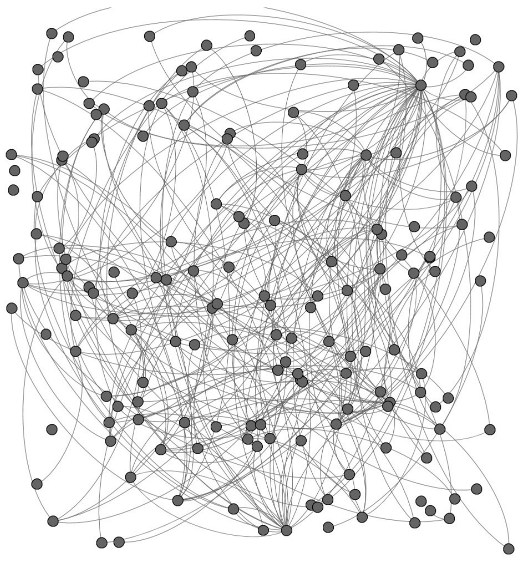


Figure 4.10: SCG for GTK Scheme project
Slika 4.10: Graf statičkih poziva za GTK Scheme projekat

[1]GTK, 2015: http://www.gnu.org/software/guile-gtk/

Another example is Unit Collaboration Network (UCN) of a middle-size in-house Delphi project (LOC 100.000) which consists of 465 nodes (Units) and 816 links between them (Figure 4.11). Example taken from [Rakić et al., 2013b].

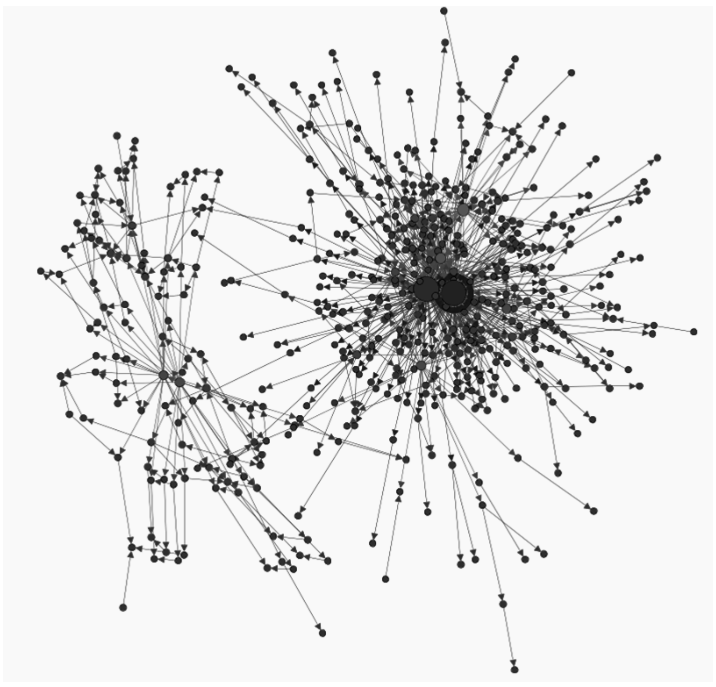


Figure 4.11: UCN of middle-size in-house Delphi project
Slika 4.11: Mreža saradnje jedinica za interni Delphi projekat srednje veličine

More results of application of eGDN Generator and SSQSA framework in general are provided in Chapter 5.

## 4.3 SSQSA components

As described in an introductory section of this chapter, SSQSA components can be categorized according to their role in the framework in three categories. First category of components is responsible for generation of internal source code representations and their manipulation in order to enable language independency of integrated analyses. Integrated static analysers form the second category, while external higher level tools integrated into the framework to use analyses results make third category.

### 4.3.1 Generators and manipulators of SSQSA internal representations

According to described categorisation of the internal representations, generators of these representations can also be divided in two levels. On the first level, eCST Generator is used to produce eCST, while the second level consists of generators of internal representations derived from eCST. Manipulators of internal representations (eCST Adaptor and eCST Anti-Generator) make the third level in the first category of the components.

#### 4.3.1.1 eCST Generator: enriched Concrete Syntax Tree Generator

The most important tool in SSQSA environment is the eCST Generator. The responsibilities of the eCST Generator are to generate the universal representation of the source code independent of input programming language and to export generated eCST in XML format.

Precondition to support any language by SQSSA is to generate appropriate parser for that language. Recommendation is to use ANTLR parser generator [2]. This parser generator provides simple, declarative-like syntax for insertion of imaginary nodes in the syntax tree. This mechanism is used to add universal nodes as special type of imaginary nodes to the syntax tree in order to produce eCST.

[2]http://www.antlr.org

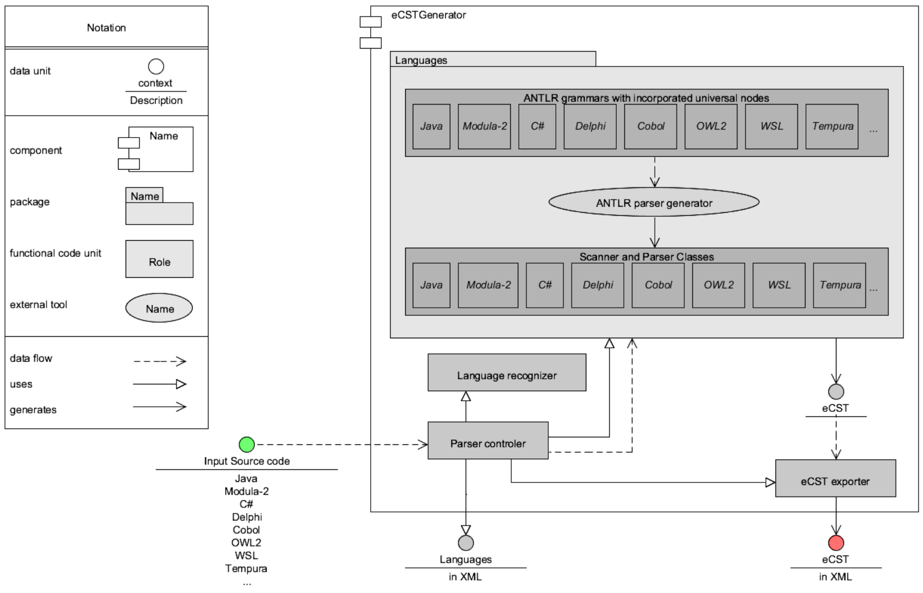


Figure 4.12: Architecture of the eCST Generator component
Slika 4.12: Arhitektura eCST Generatora

When scanner and parsers for producing eCST have been generated, input source code can be processed. After submitting an input file containing the source code written in any of supported languages, an input language will be recognized based on file extension and an appropriate scanner and parser will be called. This is enabled by the storing of all needed information about the input language to shared XML storage. XML scheme for storing information about languages is provided in Figure 4.13. From the shared data storage, eCST Generator reads all needed data about the language and invokes corresponding scanner and parser. As a result, the source code will be translated to an eCST intermediate representation and the result will be saved in the form of XML file. These XML files are inputs for all available tools (Figure 4.12).

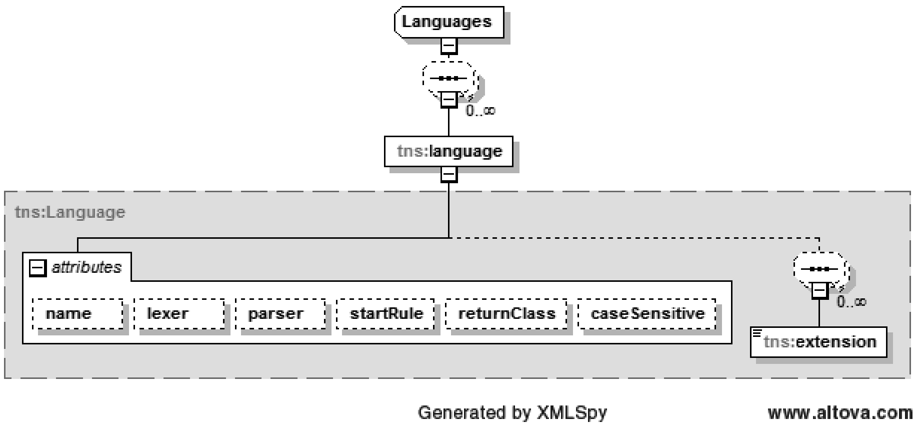


Figure 4.13: XML schema for storing information about supported input languages in XML format
Slika 4.13: XML šema za smeštanje informacija o podržanim jezicima u XML formatu

For storing of eCST, eCST Generator uses XML schema that was created based on standard output for AST used in ANTLR parser generator (Figure 4.14). In this way we keep all information provided by generated scanner and parser that can be useful to the further analysis, but also keep them in widely accepted conventional form.

For each compilation unit one XML file is created. In this form source code representation do not directly reflect different relations between compilation units even if needed information are present in each eCST. For complete source code representation we introduce software networks, as described in section 4.2.2.2, to connect compilation units. Extraction of software networks is done by eGDN Generator (section 4.3.1.2).

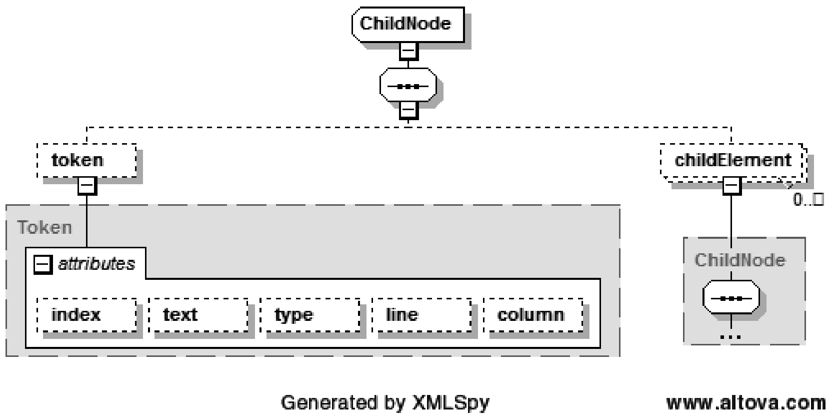


Figure 4.14: XML Schema for storing eCST in XML format
Slika 4.14: XML šema za smeštanje eCST-a u XML formatu

**History** of eCST Generator begins with SMIILE tool (section 4.3.2.1) which was the basis for development of the whole SSQSA framework. Motivation for its development was fulfilment of the gap in the field of systematic application of software metrics and corresponding tools in software industry by overcome of difficulties caused by the weak points of available tools described in section 3.2.

First version of the SMIILE tool consisted of the first prototype of eCST Generator and software metrics calculator . It supported only two characteristic languages: object-oriented Java and procedural Modula-2. Currently, more than 10 different paradigm language such are Java, C#, Python, Delphi, Modula-2, Pascal, C, PHP, JavaScript, COBOL, Scheme, Erlang, Tempura, WSL, OWL, etc. are supported. [Rakić, 2010], [Rakić and Budimac, 2011]

#### 4.3.1.2 Generators of the internal representation derived from eCST

As described in section 4.2, eCST is a starting point for deriving eGDN and eCFG as subordinate internal representations. In this section, generators of these representations are described.

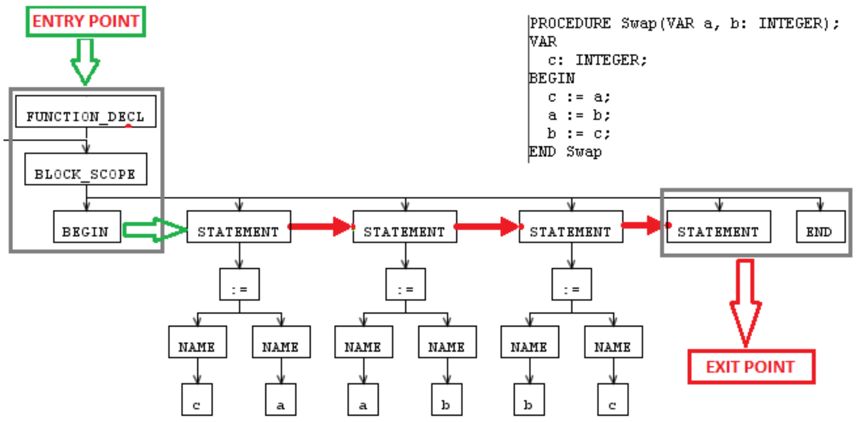
**eCFG Generator: enriched Control-Flow Graph Generator.** The main task of this component is to enrich existing eCST representation with edges that represent control flow between statements or basic blocks contained in source code. It is done based only on universal nodes. More particular, algorithm looks for universal nodes representing general statements (e.g. assignment) and statements taking care of control flow structures (branches, loop, jumps, etc.) as special cases of statements. Here is important not to omit pre-conditions and post-condition as special cases of condition in the sub-tree of these structures. Finally these nodes are to be connected in a directed graph where branches follow possible execution paths of the program.

General approach for eCFG construction is presented in Figure 4.15. Result is stored to XML file according to Schema provided in Figure 4.16. This XML Schema has been adopted from GraphML[3]: XML based file format for graph representation and can be described as sub-schema of official GraphML XML Schema. In SSQSA framework, XML schema shown in Figure 4.16 is used as a common schema for storing all graph-based internal representations.

**eGDN Generator: enriched General Dependency Network Generator.** From a set of eCST trees, each representing one compilation unit, eGDN Generator extracts a set of networks which reflect internal structure of a software system at different levels of abstractions (section 4.2.2.2).

eGDN Generator consists of two sub-components: GDN Extractor and GDN Filter. From the set of eCST trees GDN extractor constructs struc-

[3]GraphML, 2014: http://graphml.graphdrawing.org



```
PROCEDURE Swap(VAR a, b: INTEGER);
VAR
  c: INTEGER;
BEGIN
  c := a;
  a := b;
  b := c;
END Swap
```

Figure 4.15: Construction of eCFG from eCST
Slika 4.15: Konstrukcija eCFG-a polazeći od eCST

ture called General Dependency Network (GDN). GDN is the union of collaboration networks at different levels of abstractions with added CONTAINS links that maintain hierarchy among nodes contained in GDN. After GDN is extracted, GDN Filter will filter the network and extract the network at certain dimension and level (section 4.2.2.2).

GDN is incrementally built from a set of eCST trees in two phases and stored to XML file according to corresponding XML Schema for graph representation used in SSQSA framework (Figure 4.16). Analyzers in both phases traverse each eCST in the set realizing trigger-deduce-action mechanism, where triggers are some of eCST universal nodes. The aim of deducing is to determine the source and destination node for a link that will be created, while actions create one or more GDN nodes and links [Savić et al., 2014].

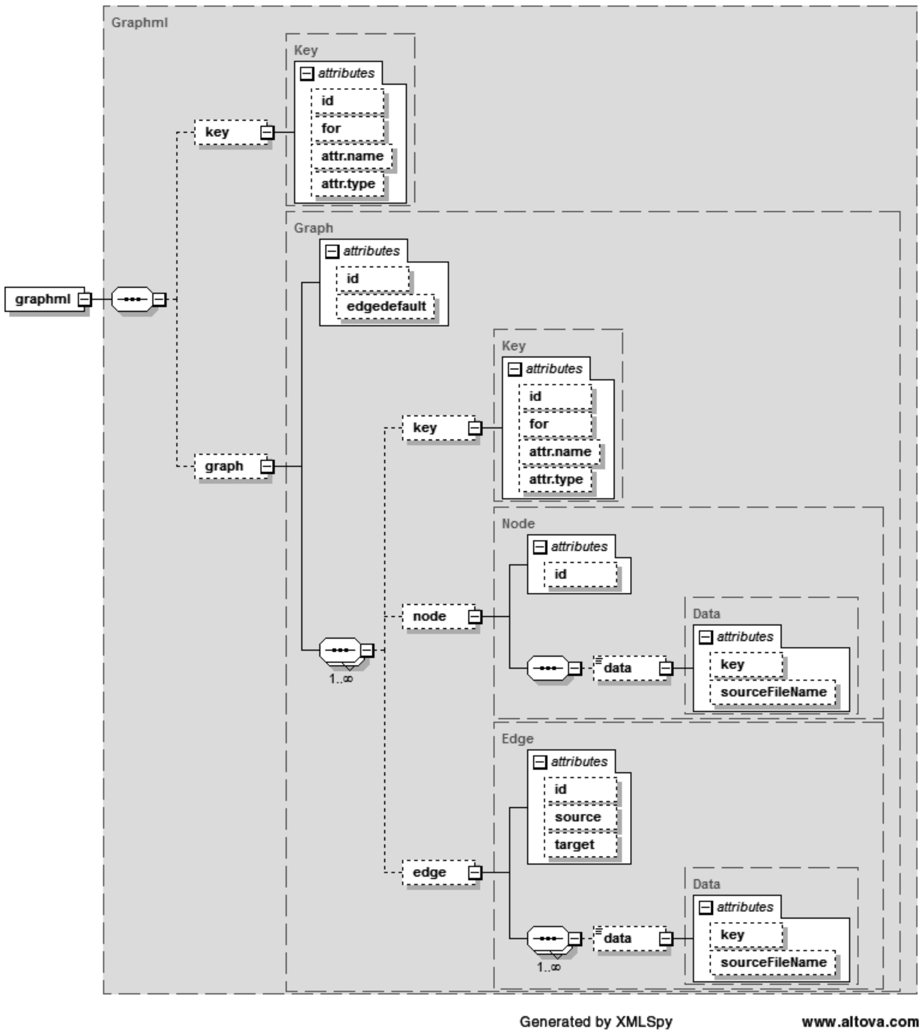


Figure 4.16: XML Schema for storing eGDN in XML format
Slika 4.16: XML šema za smeštanje eGDN-a u XML formatu

**History** of this component begins with similar in-house software network extraction tool named YACCNE [Savić et al., 2011]. It was built to extract and analyse software networks representing programs written in Java programming language. By accepting the eCST as an intermediate representation of the source code and adapting the implementation, this tool started to support all languages that are supported by the framework. It becomes integrated component of SSQSA framework named SNEIPL: Software Networks Extractor Independent of Programming Language [Savić et al., 2014]. This component was still responsible for extraction and analysis of extracted networks. Therefore this tool was only one more analyser and GDN was only its own intermediate representation derived from eCST. After accepting GDN as common internal representation in SSQSA framework, renaming it with eGDN, and separating these responsibilities according to layers in the framework, SNEIPL tool becomes only extractor of the networks as an internal representation (eGDN), and gets the name eGDN Generator. Finally, component in charge for analysis of generated networks, the other part of original responsibility on SNEIPL, is SNAIPL: Software Networks Analyser Independent of Programing Language (section 4.3.2.2).

#### 4.3.1.3 eCST Manipulators

SSQSA components at this level are involved in manipulating eCST. This level includes: specific eCST post-processor called eCST Adaptor and the tool that generates source code from its eCST representation called eCST AntiGenerator.

**eCST Adaptor.** Even if the eCST is quite universal representation of source code, some imperfections caused problems during some analyses. One of the typical examples is dynamic instantiation in Java.

Listing 4.2: Dynamic instantiation in Java
Listing 4.2: Dinamičko instanciranje u Javi

```
String exampleClassName =
getExampleImplClass(); Example example
  = (Example)Class.forName(exampleClassName).newInstance();
```

On the level of eCST it would be recognized as a function (method) call with return type of a super-class or interface.

Another difficulty that often appears is related to languages where types are not statically but dynamically defined.

Listing 4.3: Dynamic types in PHP
Listing 4.3: Dinamički tipovi u PHP-u

```
$variable = 1;
print gettype($variable)."␣variable␣=␣".$variable."<br>";

$variable = "Some␣string";
print gettype($variable)."␣variable␣=␣".$variable."<br>";
```

There are two common characteristics of described imperfections to be solved:

- these imperfections do not affect all the tools and their analysis;
- elimination of some imperfections directly on the generated trees could break the main rule that the eCST is matching the source code.

Therefore, eCST Adaptor which explicitly provides functions to adapt some aspect of the tree has been introduced. Some of these adaptations which would damage the content of the input code (example: adding of determined type of the variable) are done only temporarily by the tool calling the function while the final eCST stored to XML remains identical to source code. However, the modified eCST can be stored to the shared storage if history will be needed.

Each eCST Adaptor function provides stand-alone functionality that is independent on programming language and depends only on content and structure of the eCST. An illustrative example is a unit marked as concrete unit (class, module) in which no function (method, procedure) has its implementation (block scope in the body of the function does not exist or it is empty). This (concrete) unit should become an abstract unit and CONCRETE_UNIT_DECL universal node should be replaced by ABSTRACT_UNIT_DECL) independently of an input language. In the case of dynamic instantiation FUNCTION_CALL will not be replaced by INSTANTIATE, as it is still function call, but this will be simply added so that analysers recognize this as both: function call and instantiation. These functionalities serve to analyzer components to solve some inner problems.

Since the adaptations are language-independent, it is possible that some tools will call the adaptation function for the language in which certain modification does not make sense. In that case the function will not produce any result. For example, if we call adaptor to find and adapt tree for dynamic instantiation for Modula-2 code, function will not find any segment of the tree to modify. Or in the case of conversion of concrete units to abstract ones, in some languages it is done already on the level of grammar because it can be solved on syntax level and it will not be affected by this function. In some other languages syntax of the language does not enable to distinguish concrete and abstract units and it has to be done based on the definition of differences between concrete and abstract units. The important point in these considerations is that the same definition, the most logical one from our point of view, is used for adapting the trees representing all languages. In that way we tend to keep valuable consistency.

**eCST AntiGenerator.** eCST contains complete source code in sub-trees of universal nodes including whitespaces and comments. For each token, line and column are stored as attributes of a XML element representing a eCST node (Figure 4.14). This enables full reconstruction of the source code represented by eCST.

eCST AntiGenerator takes eCST representation of the source code, parses it and produces source code written in original input language. This functionality will get its value after development of automatic improvement of the design and/or implementation such is refactoring. In this case, after required changes on eCST, it will be possible to generate improved source code.

### 4.3.2 SSQSA static analysers

In this section SSQSA static analysers are briefly described.

#### 4.3.2.1 SMIILE: Software Metrics Independent on Input LanguagE

SMIILE is a software metrics tool [Rakić and Budimac, 2011]. It accepts set of XML input files containing eCST and eGDN and calculates values of software metrics that reflect characteristics of corresponding source code and design. Results are stored to XML and placed to shared storage according to XML Schema provided in Figure 4.17.

SMIILE currently supports calculation of several software metrics representing different categories of metrics.

Metrics applied on compilation unit level are calculated on the level of eCST. Metrics reflecting relations between entities are derived from software networks. The list of metrics follows.

**Size and Complexity** of the source code measured on eCST representation of source code:

- **LOC** : Lines of Code calculated only based on lexical structure;
- **CC** : Cyclomatic Complexity;
- **WFCU** : Weighted Function per Concrete Unit which is paradigm independent equivalent of Weighted Method per Class (WMC);
- **H** : Halstead.

**Design metrics** calculated on eCST level even if applicable also on dependency network level:

- **NOP** : Number of Packages;
- **NOCU** : Number of Concrete Units which is paradigm independent equivalent of of object-oriented Number of Classes (NOC);
- **NOF** : Number of Functions which is paradigm independent equivalent of Number of Methods (NOM).

**Design and complexity metrics** reflecting dependencies between entities involved in software design are calculated on the level of dependency networks. Some of these metrics follow:

- **CBI** : Coupling Between Instances which is paradigm independent equivalent of Coupling Between Objects (CBO);
- **NOC** : Number of Children;
- **DIT** : Depth of Inheritance Tree;
- **RFCU** : Response for a Concrete Unit which is paradigm independent equivalent of Response for a Class (RFC);
- **LCOF** : Lack of Cohesion in Functions which is paradigm independent equivalent of Lack of Cohesion in Methods (LCOM);
- **Henry-Kafura complexity** fan-in, fan-out, etc.

All metrics raised to paradigm independency level can be calculated in the domain where not applicable. In this case these metrics will have default value so that their appearance does not lead to confusion in software system analysis and understanding. For example, DIT metric applied to non-object-oriented code will have value 0 because inheritance does not exist in non-object-oriented languages.

SMIILE relies on language independency based on eCST representation of the source code and calculates broad range of metrics with possibility to extend this set if needed. History of the source code and corresponding metrics results are stored to shared storage. Source code is stored in the original format and in its internal representations, while metric results are stored to XML documents according to already mentioned XML Schema (Figure 4.17). Furthermore this shared storage enables the usage of results by other integrated or external tools.

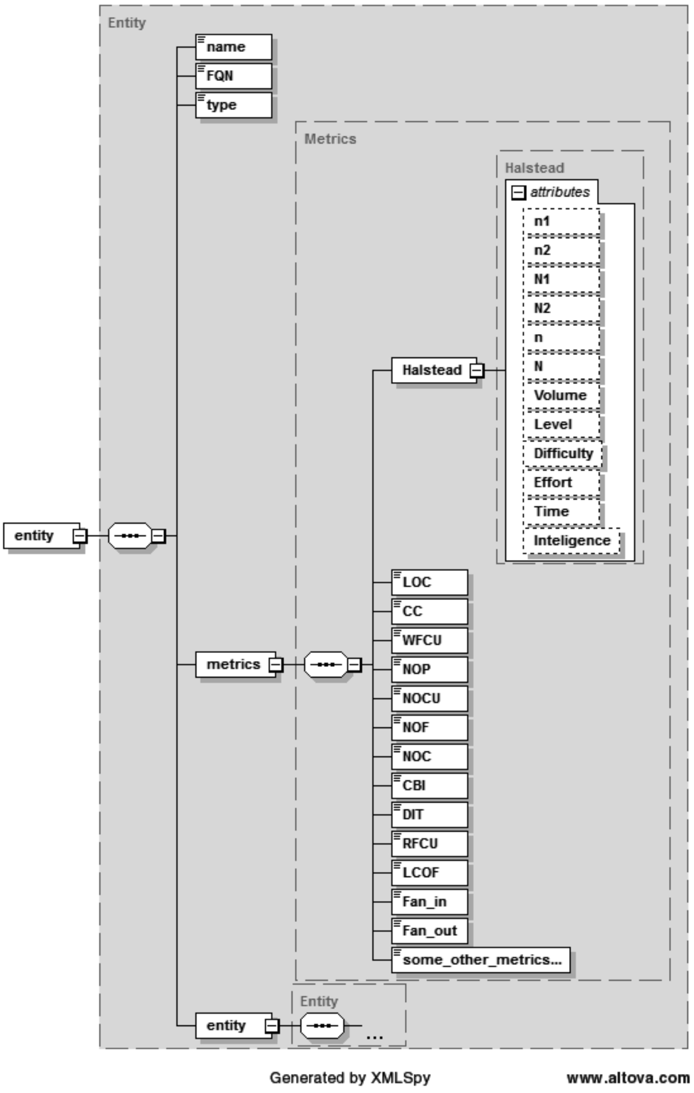


Figure 4.17: XML Schema for storing values of software metrics in XML format
Slika 4.17: XML šema za smeštanje softverskih metrika u XML formatu

**History.** Motivation for development of SMIILE was fulfilment of the gap in the field of software quality monitoring and assurance. First step in this direction is improvement of applicability of software metrics and corresponding tools by introducing a new tool with enhanced features. Thus, the weak points of available tools described in section 3.2 were avoided and the stable foundation for further development was built.

First version of the SMIILE tool [Rakić, 2010, Rakić and Budimac, 2011] consisted of the first prototype of eCST Generator and software metrics calculator with implementation of only two characteristic metric algorithms: language-independent LOC and CC which is very influenced by syntax constructs of observed language. Furthermore, it supported two characteristic languages: object-oriented Java and procedural Modula-2. The prototype proved that described approach can lead to expected results but still some obstacles were to be overcome. The most important one was that each eCST was representing one compilation unit. Even all information needed for extraction of facts about dependencies and relation between units, functions, and other entities were contained in the eCST, support for extraction still was not implemented. This was an obstacle for implementation of design metrics. After integration of SNEIPL tool for extraction of software networks, later reengineered and named eGDN Generator (section 4.3.1.2), this problem was solved.

#### 4.3.2.2 SNAIPL: Software Networks Analyser Independent of Programing Language

As described, in SSQSA framework GDN: General Dependency Networks representing dependencies between software entities figuring in collected set of eCST representation of source code is extracted by eGDN Generator component (section 4.3.1.2). eGDN Generator provides also filtering of GDN and extraction of software networks according to parameter representing perspective (vertical and horizontal) and level of abstraction. Corresponding set of design software metrics are calculated on these networks by SMIILE (section 4.3.2.1).

Based on all supplied facts shared in the XML storage, statistical analyses and visualizations are applicable on software networks (section 2.2.2.2). SNAIPL tool observes generated networks taking into account metrics results, visualise them, applies statistical analyses, and finds different anomalies in software system.

**History** of SNAIPL tool start with integration of Java language specific software networks extractor and analyser (YACCNE) [Savić et al., 2011] into SSQSA framework. This tool was integrated into SSQSA framework as united component called SNEIPL [Savić et al., 2012], [Savić et al., 2014]. First this was only one more analyser with its own intermediate representation of the source code and design in form of a graph (software network later named eGDN) but generated from eCST. After the incorporation and acceptance of eGDN as one of the common intermediate representations into SSQSA framework, SNEIPL was re-engineered so that it fulfils the needs of the framework. SNAIPL remained the analysis component while extractor (eGDN Generator) has been moved one level up to the level of generators of intermediate representation.

**ONGRAM: ONtology GRAphs and Metrics.** Special, domain-specific case of SNAIPL tool is ONGRAM [Savić et al., 2013]. It observes networks in the ontology domain. Currently its results are applied only on eCST representation of OWL code. Even ontology domain is specific in some sense, this analyser is to be generalised and integrated with general SNAIPL to meet domain independency.

#### 4.3.2.3 SSCA: Software Structure Change Analyzer

SSCA is a software tool that is able to track and analyze changes in the structure and design of the software product over the time. These changes occurs during the software evolution as a result of refactoring and reengineering of the software product, but also due to functional changes in the program.

SSCA takes a set of eCSTs as its input and produces the set of changes in the program structure. It uses appropriate meta-model which is to be filled only with data from eCST that are sufficient to apply algorithms detecting software changes. Since it is done based on the information taken from eCST, it is completely independent on the programming language. Detected changes are stored to XML according to a schema given in Figure 4.18. Type of the change can be some of the standard changes such are *move method*, *move fragment*, *move field*, *extract method*, etc.

**History.** Similarly to some other components this tool has history in the language specific domain. Previously used source code representation restricted loading data to meta-model only for C# programming language. After accepting eCST, this tool achieved language independency of the change analysis which was the basic goal of this integration. Additional benefit of inclusion in the framework is possibility to collaborate with other tools. One example of this collaboration is tracking changes and analysis of real effect of these changes to the software quality. This is possible by comparing values of software metrics before and after the changes. The first results and experiences on the SSCA can be found in [Gerlec et al., 2012].

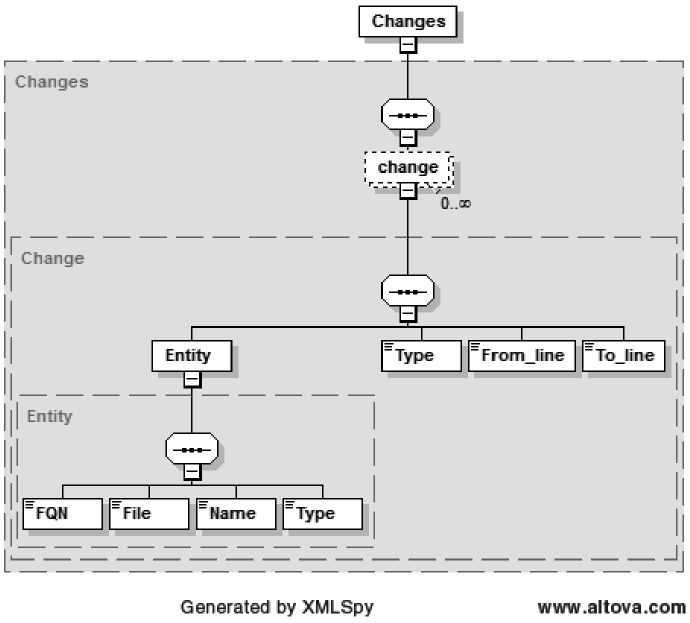


Figure 4.18: XML Schema for storing detected structural changes in XML format
Slika 4.18: XML šema za smeštanje detektovanih strukturnih izmena u XML formatu

#### 4.3.2.4 LICCA: Language Independent Code Clone Anlysis

As described in section 2, clones in source code can be defined as fragments of source code which are real duplicates or similar enough (different up to some level). Clones can be considered as syntactic or semantic duplicates. Consequently, different techniques can be applied for their detection. Some of used techniques are token based, tree based, metrics based, but in practice these techniques are usually combined into different hybrid approaches to clone detection.

Taking into account nature of eCST representation of source code, LICCA combines token based and tree based approach. Combined techniques can be categorized as syntax-based and semantic-based. Analysis relies on universal nodes which contains semantics of marked syntax constructs. Therefore, some of semantically similar code fragments will be recognized. Additional advantage of this tool is that it is capable to detect duplicates in fragments of source code written in different languages. For example, if some code fragment is duplicate to another fragment even if it is placed in the other component written in some other language it will still be detected, if the difference is only in the language specific syntax.

Furthermore, having in mind overall potential of SSQSA framework, next prototype of LICCA tool will include software metric results into clone detection. Thus, applied hybrid approach will be extended with some of metrics based technique. Additionally, it is possible to involve graph-based techniques. Since this tool is based on eCST, it is capable of clone detection in programs written in different programming languages which gives completely new possibilities rarely considered in other tools for clone detection.

Detected clones are exported to XML document and stored to shared storage according to XML Schema provided in Figure 4.19.

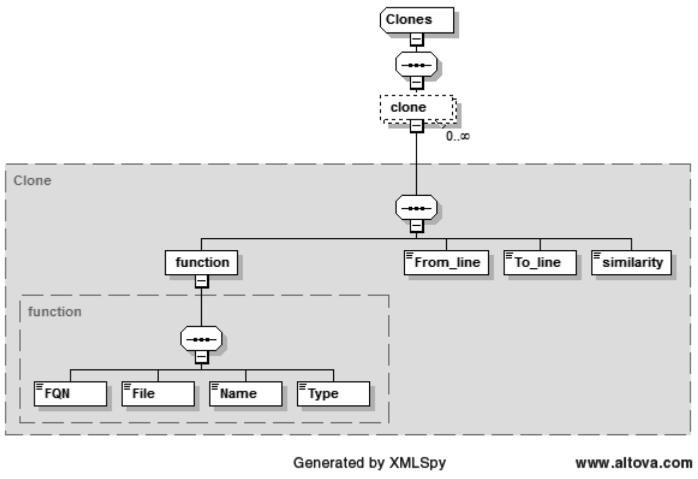


Figure 4.19: XML Schema for storing detected code clones in XML format
Slika 4.19: XML šema za smeštanje detektovanih klonova koda u XML formatu

### 4.3.3 Peripheral tools integrated in SSQSA framework

Some external tools used in the domains where static analysis can be applied, are integrated into SSQSA framework as consumers of generated results. In this section two characteristic and the most prospective components are described.

#### 4.3.3.1 External software metrics repository

SMIILE tool was completely integrated with the software metrics repository that stores metrics data and enables investigation of the Quality Index [Heričko et al., 2007]. Furthermore, repository gives opportunity to user to define new composite metrics based on previously calculated values of basic metrics [Gerlec and Živkovič, 2009], [Gerlec et al., 2011].

This integration provides possibility to import XML metrics history with the intention to make conclusions based on metrics data and to provide meaning of the calculated numbers to the end user.

At this point of evolution this component is integrated into SSQSA framework communicating directly with shared storage without communication with SMIILE tool. It is also possible to collect results of SSCA in this repository.

#### 4.3.3.2 Integration with the Testovid

Application of software metrics can significantly improve quality of education in the field of computer science. Apart from easier and more consistent assessment of students' solutions, a teaching staff acquires additional time for interactive work with students. This is additional benefit of introducing of automatic assessment.

The first results and the small example of usage of software metrics in automatic assessment of the students programming assignment solutions are given in [Pribela et al., 2012]. Even if the import of metric results generated by SMIILE tool into the Testovid [Pribela et al., 2011] environment through XML is fully implemented, there are still many open questions related to this integration. The one of the weak points is that the instructor should use his knowledge and experience to choose appropriate metrics to be used for the assignment in question. There is no automatically generated recommendation which metrics are appropriate to express quality of student solution for some specific problem. In this direction an extensive research has to be done and afterward some intelligent techniques may be needed.

Another open direction to full integration is engagement of code clone analyser and its results in plagiarism detection in students solutions.

## 4.4 Adaptability of SSQSA framework

Based on all facts presented, it can be noted that eCST Generation, and indirectly the whole SSQSA framework can be easily adapted to accept inputs written in new languages. Process for adding support for a new languages is described in [Kolek et al., 2013]. After adding support for new language all analyses integrated into SSQSA framework are immediately applicable to inputs written in this language.

To add support for a new input language we will primarily need the language specification. The best option is to base further steps on formal language specification, hopefully represented by language grammar. Since ANTLR parser generator is the primary tool for adding and maintaining language supports (and its notation is very similar to EBNF: Extended Backup-Naur Form notation), it can be concluded that the most appropriate starting point for adding a new language is its specification in a form of grammar written in EBNF notation. Counterpart example is a language specification in BNF notation, where left recursion is a natural feature and repetitions are expressed with recursion. Grammars that contain a left recursion can not be handled by the recommended version of ANTLR parser generator and hence one who introduce a new language has has an additional task: to eliminate the left recursion during the translation to ANTLR notation. More precisely, eCST Generator relies on parser generated by ANTLR v3[4] which generates standard LL(*) parsers and consequently can-

[4]ANTLR3 2015, http://www.antlr3.org

not handle left recursion in grammar rules [Parr and Fisher, 2011]. The last version of ANTLR (v4)[5] introduces support for left recursion through adaptive LL(*) parsing [Parr et al., 2014]. However, ANTLR v4 is not recommended for generation of parsers for the SSQSA framework since it does not support tree-rewriting: currently ANTLR v3 is the last version of ANTLR supporting tree-rewrite rules. Although, generation of parser by ANTLR v3 is recommended, it is not the only possible choice for adding support for a new language in SSQSA framework (see the following section).

Finding the most ideal language specification, written in EBNF notation and with LL(1) property is very rare case, because most popular languages have ambiguous grammars. Still, any language specification can be used. In that case the process for introducing the language in the framework consists of the following steps (Figure 4.20):

1. write an ANTLR grammar for the language:
    - translate the given language specification to ANTLR notation;
    - add rules for syntax tree generation to the grammar (*rewrite the rules*);
    - add the universal nodes to the syntax tree (*extend the rules*).
2. generate the parser and the scanner;
3. add the information about the new language required for the automatic invocation of scanner and parser to the XML document containing supported languages.

We illustrate step (1) with a simple example - a rule for WHILE statement in a Pascal-like language (Listing 4.4).

Listing 4.4: Grammar rule for the WHILE statement
Listing 4.4: Pravilo gramatike za naredbu WHILE

```
whileStatement : WHILE expression DO statement;
```

[5]ANTLR4 2015, http://www.antlr.org

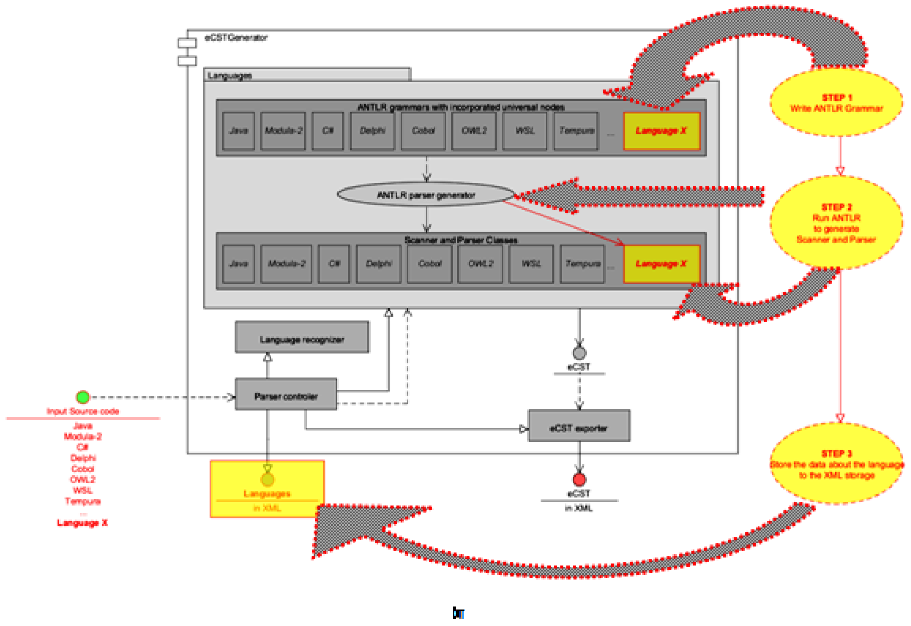


Figure 4.20: Steps for introducing support for a new language in the SSQSA framework
Slika 4.20: Koraci za uvođenje podrške za novi jezik u SSQSA okvir

We rewrite the rule (Listing 4.5) by adding syntax tree generation (Figure 4.21).

Listing 4.5: Rewriting the rule
Listing 4.5: Prepisivanje pravila

```
whileStatement : WHILE expression DO statement
  -> ^( WHILE expression DO statement ) ;
```

By adding universal nodes (Listing 4.6) we convert syntax tree to the eCST: enriched Concrete Syntax Tree (Figure 4.22)

Listing 4.6: Enrichment of the tree
Listing 4.6: Obogaćivanje stabla

```
whileStatement : WHILE expression DO statement
```

```
-> ^( LOOP_STATEMENT
  ^( KEYWORD
    ^( WHILE
      ^( CONDITION expression )
      ^( KEYWORD DO)
        Statement
    )
  )
);
```

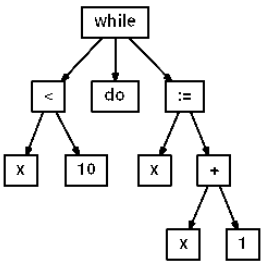


Figure 4.21: Syntax tree before enrichment
Slika 4.21: Sintaksno stablo pre obogaćivanja

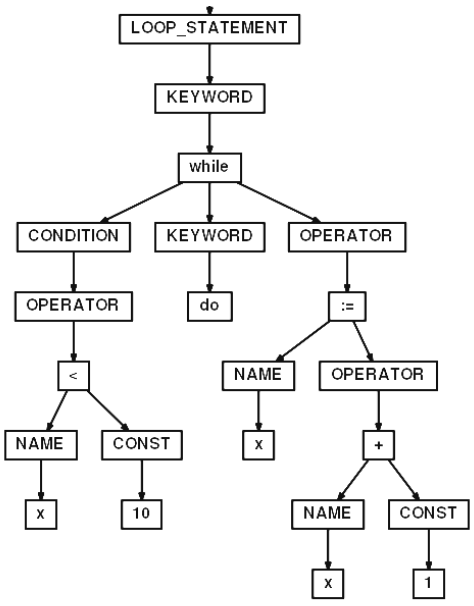


Figure 4.22: Syntax tree after enrichment
Slika 4.22: Sintaksno stablo nakon obogaćivanja

When all rules are defined as described, scanner and parser can be generated and the language can be included in the framework.

### 4.4.1 Role of the parser generator

Syntax trees are usually secondary product of language processing tools such are parser, compiler, etc. These tools could be produced automatically by generators or developed manually implementing the language rules. Automation of parser generation process has its pros and cons. It can be justified by amount of work needed to implement a new parser. In contrast, we have constraints given by functionalities provided by automatic generation tools.

To justify the usage of parser generator we can provide overview of number of lines of code needed to produce a grammar and to implement full scanner and parser. If we consider the example provided in section 4.4 (*while* statement), we can see that the rule without tree generation rules is contained in one line of code, with flat tree generation it occupies two lines; and when we add universal node for tree enrichment it takes 6 lines. The parser method generated for this rule takes 43 lines without eCST generation and 85 including it. These are all lines of code excluding empty lines and comments. In the case of manual implementation this size could eventually be reduced but not importantly.

Parser generators take a language specification usually provided as a language grammar as its input and return parser for that language as an output. This grammar is provided in some default form (e.g. EBNF Extended Backus-Naur form) or in some generator-specific notation. These generators usually have embedded mechanisms to generate syntax trees as internal structures. Additionally, these mechanisms can be extended with mechanism for enrichment of syntax trees with additional information about language or input source code. This opportunity is our key instrument. In this thesis, parser generator is used to generate scanner and parser for each supported language.

To be used by SSQSA framework parser generator has to fulfil certain requirements. The following characteristics of parser generator are required:

**Set of supported target languages.** eCST Generator and the most of SSQSA components are written in Java programming language. Therefore, it is desirable to generate scanner and parser written in Java.

**Rules notation.** Language syntax can be specified using different notations. It is important that notation used by parser generator is easy to learn and similar common or standard notations.

**Syntax tree generation.** Parser generated by parser generator has to be able to extract syntax tree as intermediate source code representation. This is crucial facility of parser generator to be used in SSQSA framework.

**Adaptability of content of syntax tree.** Apart from generation of syntax trees it is important that parser generator supports modifications of syntax trees (rewriting) by affecting their content. It is important to enable:

- inserting imaginary nodes into the trees;
- inclusion of complete source code containing both code and comments.

**Rewriting notation.** Parser generators, if support rewriting of syntax trees, require implementation of rewriting rules in target language. More convenient option is to enable definition of rewriting rules using common notation.

**Structure of generated syntax tree** To be used by SSQSA framework, parser generator should produce syntax trees in a structure which is easily convertible to standard SSQSA eCST output in XML format (Figure 4.14)

**Parsing algorithm.** With respect to parsing algorithms, parsers can be generally categorised as *top-down* and *bottom-up* ones. The most characteristic represent of top-down parsing algorithms is LL(x). It analyses the input from *Left to right*, performing *Leftmost derivation* of it, while *x* denotes that parser uses *x* tokens of look-ahead when parsing a sentence (e.g. LL(1), LL(k) and LL(*)). The most characteristic bottom-ut algorithm is LR. It processes the input from *Left to right* and produces a reversed *Rightmost derivation.* Variations of this algorithm are LARL: Look-Ahead LR and SLR: Simple LR parsing [Aho et al., 2006].

According to fulfilment of these requirements, it is possible to determine possibility of usage of each available parser generator in SSQSA framework. Table 4.4 provides overview of parser generators the most similar to ANTLR considering described characteristics.

Based on described advantages of ANTLR comparing to other parser generators, ANTLR is recommended parser generator for integration of support for a new language in SSQSA framework. The main advantage is its notation for specifying the rules for generation and enrichment of syntax trees by imaginary nodes. There are some parser generators that do not enable generation of the code for producing syntax trees, or they provide building only basic syntax trees without possibility to affect its content or structure. Some other parser generators provide possibility to enrich the syntax tree, even to change the structure of the trees, but changes are possible at the level of implementation in the target programming language (Coco-R, JavaCC, etc). In ANTLR these changes are at the declarative level and require only simple changes in the grammar rule.

In this direction, further automation of the grammar development and parser generation process would be valuable. This is possible by involving additional tools as described in [Porubän et al., 2010]. The aim of this tool is to make easier development of domain-specific languages. It provides graphical interface [Bacikov́a et al., 2013], and generation of the grammar directly from the graphical representation of the language. Further steps involve selected (third-party) parser generator to generate parser of the language. There are two precondition for involving this tool to improve SSQSA adaptability: (1) to enable development of general purpose languages, and (2) to involve ANTLR (or alternative parser generator with required characteristics) in generation of the parsers. With two additional characteristics, this tool would become candidate to be recommended as default tool for introducing support for new language in the SSQSA framework.

Finally, recommendation to use ANTLR parser generation does not mean limitation to use only this mechanism to produce the parser. If some-one want to add support for some language and has a mechanism to generate syntax tree with possibility to enrich the tree by imaginary nodes, in that case we do not need to start writing grammar from the scratch and passing through presented steps. In this case it is more painless to modify existing mechanism and to adapt the generated intermediate representation to become convertible to eCST. One such example is integration of support for Erlang into SSQSA environment [Páter-Részeg, 2013], [Tóth et al., 2015] by transforming semantic graph produced by RefactorErl[6] tool.

[6]RefactorErl, 2014 http://plc.inf.elte.hu/erlang/

| Parser generator | Java target language | Rule notation | ST: Syntax Trees generation | Adapt. of ST content | Rewriting | Structure of ST | Parsing algorithm |
|---|---|---|---|---|---|---|---|
| ANTLR [a] | + | EBNF | + | + | Declarative | + | LL(*) |
| Coco-R [b] | + | EBNF | + | requires implementation | Imperative | Adaptable | LL(1) |
| JavaCC [c] | + | EBNF | + | + | declarative /imperative | + | LL(k) |
| Lisa[d] | + | BNF | + | + | declarative | adaptable | LL, SLR, LALR, LR |

Table 4.4: Overview of parser generators considering requirements of SSQSA framework
Tabela 4.4: Pregled generatora parsera uzimajući u obzir zahteve SSQSA okvira

[a]ANTLR v3, online 2015, http://www.antlr3.org/
[b]Coco-R, online 2015 http://www.ssw.uni-linz.ac.at/Coco/
[c]JavaCC, online 2015 https://javacc.java.net/
[d]Lisa, online 2015, http://labraj.feri.um.si/lisa/

### 4.4.2 Demonstration od Adaptability

The aim of this section is to demonstrate the adaptability of SSQSA framework to a new language. Hence, characteristic issues during integration of characteristic languages representing corresponding paradigm will be described and illustrated by appropriate examples. The basic idea is to mark all constructs with common semantic by the same universal node. Following this idea all units (modules, classes, etc.) are marked by corresponding unit node, all functions (procedures, functions, methods, etc.) are marked by function node, all loop statements are marked by universal node for loops, similar holds for all branch statements, etc. General approach applied for mapping language constructs to universal nodes is described in section 4.2.1.1, while more details about mapping certain constructs of supported languages to universal nodes is provided in appendix A.

#### 4.4.2.1 Procedural languages

As described in section 2.3, the main concept of procedural languages is splitting program functionality into smaller functional units (procedures). It is not a rare case that procedural languages support modularisation as a mechanism to group logically related functional units into modules. Represents of pure procedural languages supported in SSQSA framework are Modula-2, Pascal, and C. These languages are at the moment in different stages of support. While Modula-2 is fully integrated in the framework, Pascal is still in testing phase. C is in the last phase of introducing of universal nodes in the grammar.

Modula-2 was one of the first languages fully-supported by eCST Generator. Mapping of language constructs has been made according to semantics and natural flow of thoughts. Modula-2 is strongly typed and modularised language with strict syntax rules which enabled setting of baselines and fundamental guidelines for further mappings.

There are some noticeable characteristics of Modula-2:

- Packaging and other constructs close to object-oriented programing are not supported.

- Even there is no object-oriented constructs in Modula-2, appearance of definition and implementation modules enabled distinguishing between interface and concrete units.

- Main blocks are code fragments that do not belong to functions and procedures. They usually serve as the initialisation fragments of modules. These fragments are to be distinguished from functions and procedures but, according to their roles in program operation, in some analyses they has to be observed as main functions. Therefore, appropriate marking of these code fragments is very important. Example of main block is given in Listing 4.7.

Listing 4.7: Simple example written in Modula-2
Listing 4.7: Jednostavan primer napisan u Moduli-2

```
MODULE Example;

FROM Terminal2 IMPORT WriteString, WriteLn;

PROCEDURE DoNothing();
BEGIN
   WriteString("This procedure is doing nothing,...)
   WriteString("except of writing this message.");
END DoNothing;

BEGIN    (* Main block *)
   DoNothing();
END Example.
```

- Observation of records as constructs close enough to object-oriented concepts of classes and objects has been left as an open topic for further discussion. Usage of records can be demonstrated by example given in Listing 4.8

Listing 4.8: Records in Modula-2
Listing 4.8: Slogovi u Moduli-2

```
TYPE
  FullName =  RECORD
          FirstName : ARRAY[0..12] OF CHAR;
              Initial   : CHAR;
              LastName  : ARRAY[0..15] OF CHAR;
          END;
  Date =  RECORD
              Day, Month, Year : CARDINAL;
            END;

  Person =  RECORD
          Name      : FullName;
                     City      : ARRAY[0..15] OF CHAR;
                     State     : ARRAY[0..3] OF CHAR;
                     BirthDay : Date;
        END;

VAR
  Self,Mother,Father : Person;
```

Important notice related to mapping of Modula-2 constructs to eCST universal nodes is that all mappings are manageable already on grammar level without any eCST Adaptor actions.

Fundamental mapping established in Modula-2 is used in further mappings, not only for other procedural languages, but for all other languages integrated in SSQSA framework.

#### 4.4.2.2 Object-oriented languages

Object-oriented approach brings many new language constructs. New universal nodes are to be involved to mark inheritance, instantiation and other object related syntax elements. Still, there are much more similarities between procedural and object-oriented languages than differences. As a represent of object-oriented languages Java is fully integrated in the framework, while C# is still in a testing phase.

When Java programming language is observed, many language construct are to be mapped very similarly as it was done in case of Modula-2. Some characteristic mappings are the following:

- There are packages and appropriate universal node is introduced.
- Classes and interfaces are mapped to the same universal nodes as implementation and definition modules in Modula-2, respectively. Still, Java introduces one more kind of units: abstract unit.

- Methods are marked by the same node as functions and procedures. Main block is placed to main method which means that characteristic main block code fragment does not exist in Java, but if it appears in other object-oriented languages the same approach as used in Modula-2 can be applied.

- Method calls are marked by the same node as function and procedure calls. The special case of method call can be recognized as dynamic instantiation. This is an issue which is not resolvable on level of language grammar, but by the eCST Adaptor (section 4.3.1.3). Example code is provided in Listing 4.9.

Listing 4.9: Dynamic instantiation in Java
Listing 4.9: Dinamićko instanciranje u Javi

```
String exampleClassName = getExampleImplClass();
Example example
  = (Example)Class.forName(exampleClassName).newInstance();
```

#### 4.4.2.3 Functional languages

As explained in section 2.3, functional languages can be described as syntactic interfaces to lambda calculus with less or more improvements with additional constructs. SSQSA fully supports Scheme as one of dialects of Lisp. Even pretty simple functional language, Scheme covers all characteristics of lambda calculus and therefore it is demonstrative enough to show supportability of functional constructs by SSQSA framework. To demonstrate deeper support of functional paradigm with all accompanying difficulties, Erlang is in process of integration in SSQSA framework. Integration of Erlang has introduced alternative approach for adding new language in SSQSA framework [Páter-Részeg, 2013], [Tóth et al., 2015].

Some characteristics of Scheme detected during its inclusion into SSQSA framework follow.

- Scheme is dynamic language where types are not explicitly defined in the code. Types are resolved in eCST Adaptor.

- Packaging is enabled by manipulation of libraries (Listing 4.10)

  Listing 4.10: Packaging in Scheme
  Listing 4.10: Paketi u Scheme-u

```
(library (example-lib) (export example-function)
(import example-lib-import)
    (define example-function 0))
```

- Units (concrete, abstract, and interface ones) are not defined. Fuctions are contained in package (i.e. library)

- There is not a certain point where the running of the program starts. Any of the functions can be called and executed. Therefore existence of a main block or main function is not a subject of discussion.

- Tail-recursive calls are used a mechanism to simulate loops, but these recursive calls were not marked as loop statements but as function calls to keep consistency with other languages. After analysis of the example given in Listing 4.11, it is clear why is this construct marked as a function call.

  Listing 4.11: Recursive calls in Scheme
  Listing 4.11:Rekurzivni poziv u SCheme-u

```
(let los ((i n) (res '()))
(if (< i 0)
  res
  (los (- i 1) (cons (* i i) res)))))
```

  This construct is not marked as a loop statement because in other languages which support recursive call, these calls are not marked as loops. This was subject of consideration but this issue was postponed to the analysers and generators of specific representations. These algorithms detect loops in call graphs and other code representations appropriate for observing these cases. This deeper consideration of the recursion over the languages resulted with first ideas for specific and new complexity metrics [Rakić et al., 2013a].

#### 4.4.2.4 Mixed paradigm languages

Many languages, even designed to be dedicated to one paradigm, during the evolution introduced elements of other paradigms. Thus, recently lambda calculus has been introduced in Java. On the other hand some languages are designed to be mixed paradigm languages. SSQSA fully supports Delphi as an example of mixed-paradigm programming language. Another mixed paradigm language, Python, is currently in the testing phase.

Delphi programming language mixes procedural and object-oriented programming. This means that syntax of Delphi enables writing programs in both styles (e.g. creating modules, but also classes and interfaces). After previous mapping for one procedural (Modula-2) and one object-oriented language (Java) has been introduced, mapping of Delphi constructs was quite a simple task. Still, some characteristics of the language were noticed.

- Delphi is not case-sensitive.
- Similarly to Modula-2, declarations are separated from the implementation, but in Delphi it is not easy to recognize (at the grammar level) which implementation belongs to which declaration. This required additional effort to be resolved. Code example is given in Listing 4.12.

Listing 4.12: Example of unit written in Delphi
Listing 4.12: Primer jedinice napisane u Delphiu

```
unit ExampleUnit; interface
  uses
    Windows, Messages, SysUtils, Variants, Classes,
    Graphics, Controls, Forms, Dialogs, StdCtrls;

  type
      exampleTForm = class(TForm);

      exampleLabel: TLabel;
      exampleButton: TButton;

      procedure exampleButtonClick(Sender: TObject);

      private
```

```
        { private declarations }
      public
        { public declarations }
      end;
  var
    exampleForm: exampleTForm;

implementation
  {$R *.dfm}

  procedure exampleTForm.exampleButtonClick(Sender: TObject);
  begin
      exampleLabel.Caption := 'Hello␣World';
    end;

 end.
```

#### 4.4.2.5 Script languages

As described in section 2.3, script languages are designed to be used for writing application extensions executable in particular environment. Everyday examples are extensions of web pages written in some script language to extend the functionality of a web page and to be executed in a browser or on the web server. However, it is not a rare case that script languages mix different paradigms.

Representatives of script languages (in testing phase of the integration) in the SSQSA framework are PHP and JavaScript. JavaScript is designed for development of client-side web functionality. PHP is programming language which can be used as general-purpose language, but it is designed for development of server-side of web applications. It combines several programming paradigms such are imperative, procedural, functional, object-oriented, etc. There are several characteristics of PHP those are important for mapping of code written in this language to eCST.

- Packaging and modularisation are not explicitly enabled, but developers often simulate this by:

    1. Placing all elements that logically belong to the same module to one file simulating modularisation;

2. Placing all files containing units logically belonging to one package to the same folder simulating packaging.

- Main block is expressed in a form of sequence of statement at top-level scope. It is not explicitly delimited by separators, but all code placed out of all available functions (e.a. methods) belongs to main block of corresponding unit. In the example in Listing 4.13 function call is the only statement in the main block.

  Listing 4.13: Example written in PHP
  Listing 4.13: Primer napisan u PHP-u

  ```
  <?php
    function writeMsg() {
      echo "Hello␣world!";
    }

    writeMsg();
  ?>
  ```

- Types are dynamically assigned and this issue is like in case of Scheme resolved in eCST Adaptor. It is important that implementation of this algorithm is language independent and that there exists only one implementation for marking types for all supported languages. Regardless the language (e.g. Scheme or PHP) types are assigned universally.

#### 4.4.2.6 Legacy languages

Legacy languages are very specific. COBOL is one of the most present legacy language and as such it is represent of legacy languages integrated in SSQSA framework. COBOL is primarily an imperative language, but during the decades it was extended by language constructs to support structural and object oriented programming. However, conventional language constructs are the most interesting in this research because modern constructs are already covered. Hence, support for COBOL was not fully introduced, but only its core. However, the most characteristic aspects of language are covered and tested on small examples.

Some of characteristic facts about basic versions (e.g. COBOL-68) of COBOL are the following ones:

- The most difficult step in integration of COBOL in SSQSA framework was writing a grammar because of the specific nature of the language. One of the reasons is very rich vocabulary of the language (set of reserved words, keywords, operators, etc.) which made a problem with generation of scanner because of limitations set by ANTLR parser generator.

- COBOL program is divided into divisions. These divisions semantically correspond to certain constructs in modern languages. For example, procedure division corresponds to concrete unit.

- In the rich vocabulary of the language, semantics of some constructs is not very obvious. For example, branching and loop statements have unusual and sometime not so obvious syntax. For example, to make branching in COBOL some easily recognizable constructs such are WHEN, IF, ON, etc. are used, but also END-OF-PAGE, INVALID KEY, etc. Example of usage of INVALID KEY is given in Listing 4.14

  Listing 4.14: Branch statement written in COBOL
  Listing 4.14: Naredba grananja napisana u COBOL-u

```
READ FILENAME NOT INVALID KEY
  perform good-read INVALID KEY
    perform bad-read
END-READ
```

- COBOL supports explicit jumps such is GO-TO.

#### 4.4.2.7 Specification languages

Specification languages can be described as formal languages used in early phases of software development process to formally specify product properties. Representative specification language fully integrated in the SSQSA framework is Tempura. It is formal specification language and can be described as syntactic interface to ITL: Interval Temporal Logic. ITL represents First Order Language extended by temporal expressions. It is designed for specification of systems by formal description of time intervals which represent sequence of states. States are mapping variables to values.

Tempura is a logical language, but also procedural because it is possible to define functions representing ITL formulae. Code fragment given in Listing 4.15 represents one function written in Tempura. This function defines an interval consisting of one state. Consequently the length of the interval is equal to 0 (len=0). Formula is true on that interval if in the initial state (the only one) X has value of 1 (X=1).

Listing 4.15: Simple example written in Tempura
Listing 4.15: Jesdnostavan primer napisan u Tempuri

```
define example() = {X = 1 and len(0)}.
```

Furthermore, Tempura has many other characteristics similar to characteristics of dynamically typed procedural language. For example, state and static variables characteristic for temporal formulae can be observed as variables and constants. Other example is operator *chopstar* which can be described as repeated self-composition of an interval and which is observed as loop statement. Example that illustrate usage of *chopstar* operator is GCD:Greatest Common Divisor (Listing 4.16).

Listing 4.16: GCD example in Tempura
Listing 4.16: GCD primer napisan u Tempuri

```
define gcd(M, N, R) = {
  exists Mt, Nt: {
    Mt = M and
    Nt = N and
    fin(R = Nt) and
    halt(Mt = 0) and
    chopstar{
      skip and
      (next Mt) = Nt mod Mt and
      (next Nt) = Mt
    } and
    always( format("M=%d, N=%d\n",Mt, Nt) )
  }
}.
```

Therefore, translation of Tempura code to eCST was highly influenced by the experience gained during integration of procedural languages.

#### 4.4.2.8 Intermediate languages

Intermediate languages are used to represent input code in an intermediate form in process of program translation, transformation, migration, or modernisation. For example, WSL: Wide Spectrum Language is designed to be intermediate language used for program translations and transformations in reverse engineering and software modernisation. It contains low-level and high-level constructs and code written in both, low and high level languages, can be expressed in WSL. WSL is integrated in the SSQSA framework as a representative of intermediate languages and its testing is in progress.

From the perspective of program structure and simplicity of syntax it is similar to scripting languages. An illustration is given in Listing 4.17 which contains GCD example in WSL. Program is sequence of statements and has mainly linear structure. It is dynamic, weakly typed language which does not support object-oriented concepts and modularisation. Main difficulties met during translation of WSL code to eCST are related to packaging and modularisation, and to types which are not explicitly assigned. Both problems can be resolved in eCST Adaptor, and solution and its implementation is unique for all languages with the same characteristics.

Listing 4.17: Example in WSL
Listing 4.17: Primer napisan u WSL-u

```
WHILE x <> y DO
  IF x >= y THEN
    x := x - y
  ELSE
    y := y - x
  FI
OD
```

#### 4.4.2.9 Domain-Specific languages

Domain-specific languages are designed to satisfy needs related to certain domain. One of the important domains nowadays is knowledge-description.

OWL2: Web Ontology Language is a declarative, domain-specific knowledge description language. It has a functional-style syntax and formal semantics. These characteristics distinguish OWL2 from languages already supported by SSQSA framework. Additionally, OWL2 axioms represent explicitly stated relations among ontological entities, but there are no existing universal node for marking relations. Therefore, new universal nodes had to be introduced. These nodes are initially related closely to ontology domain.

Following procedure for introduction of new nodes described in section 4.2.1.1 three domain-specific universal nodes that denote different categories of explicitly stated relations in general are defined:

- BINARY_RELATION (BR) marks binary relations;
- SYMETRIC_RELATION (SR) marks symmetric n-ary relations;
- PARTIALLY_KNOWN_BINARY_RELATION (PNBR) marks binary relations in which one of the arguments is not known at the moment.

An illustrative example for binary relation is *SubClassOf* relation which denotes that one class is a subclass of another one (Listing 4.18).

Listing 4.18: Binary relation in OWL2
Liting 4.18: Binarna relacija u OWL2

```
  SubClassOf ( :C :CPP) )
```

In OWL2 BINARY_RELATION marks all OWL2 relations that denote subsumptions and assertions. The SYMETRIC_RELATION universal node is associated with relations indicating the equivalent and disjoints classes, same and different individuals, and equivalent and disjoint object properties. The PARTIALLY_KNOWN_BINARY_RELATION universal node marks object property domain and object property range relations.

The newly introduced universal nodes are currently used only as domain-specific universal nodes for the eCST representation of ontological descriptions. However, they can be used to mark explicitly stated binary and symmetric relations in other descriptive languages as well. Explicitly stated relations among entities in already supported imperative programming languages are marked with specific, more concrete universal nodes, such as EXTENDS and IMPLEMENTS. Those universal nodes can be viewed as sub-concepts of the BINARY_RELATION universal node. Therefore, usage of these nodes has to be observed with special attention in the future in order to keep the set of universal nodes as minimal as possible having in mind procedure described in section 4.2.1.1

## 4.5 Extendability of SSQSA framework

When eCST and derived representations are generated all available analyses are applicable for all supported languages. Furthermore, set of analysers is extensible by integration of new analyses. Adding support for new analysis means implementing corresponding algorithm over an internal code representation. If this algorithm is to be implemented on some of internal representation derived from eCST than it is enough to implement it. If implementation of the algorithm requires traversing of eCST than, to add support for this analysis in SSQSA framework the following steps are to be followed [Kolek et al., 2013]:

1. Analyse the set of available universal nodes from the catalog of nodes and determine the role of each one in the algorithm to be implemented. In this step it can be concluded that some nodes has to be included in the set of universal nodes. As defined by the procedure (section 4.2.1.1) nodes are primarily included in the catalog as domain-specific (or analysis-specific) nodes.

   In the procedure of introducing of new universal node, the following steps can be determined:

- Consider all supported languages and include new nodes to catalog of nodes;
- Extend existing grammar rules (for all supported languages) as it is described in section 4.4;
- Generate new scanner and parser classes (for all changed grammars).

2. Traverse the eCST to accomplish the analysis according to parameters determined in step (1). It is possible that this step will require generation of some additional analysis-specific internal representation. For example, techniques relying on control-flow analysis required generation of eCFG because algorithms are implemented on this representation.

Important facts are:

- Newly integrated analysis is applicable to all supported languages;
- Integration of new analysis and potential introduction of new universal nodes must not affect exiting functionality.

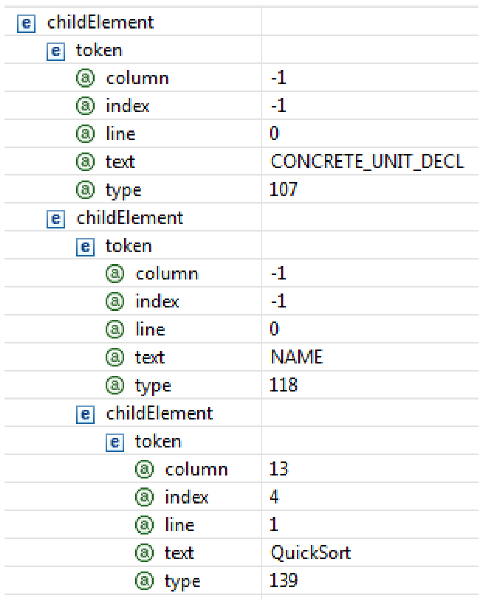
### 4.5.1 Demonstration of extendability

To demonstrate extendability, this section will provide description of characteristic examples of analyses and procedures for their inclusion in SSQSA framework. There are three characteristic cases of possible new analysis: syntax-independent, syntax-based, and dependency-based.

#### 4.5.1.1 Syntax-independent analyses

Analyses which do not depend on syntax of input language keep this characteristic when implemented directly on source code or any internal representation. Example of such analyses are LOC, SLOC, CLOC and BLOC metrics (section 2.2.2.1). When calculating these metrics, the only information needed is the usage of certain lines in the input file. Calculating of these metrics is based on counting all lines in the file, lines containing code, and lines containing comments. This can be done by parsing input source code or any representation of it that contains these information. Calculation of other metrics from this family (PLOC: Physical Lines of Code and LLOC: Logical Lines of Code) requires additional information related to language syntax, programming style, etc.

In the SSQSA framework, all analyses are implemented on some of internal representations of source code. The most appropriate is to implement LOC, SLOC, CLOC and BLOC metrics on eCST as it contains all required information:

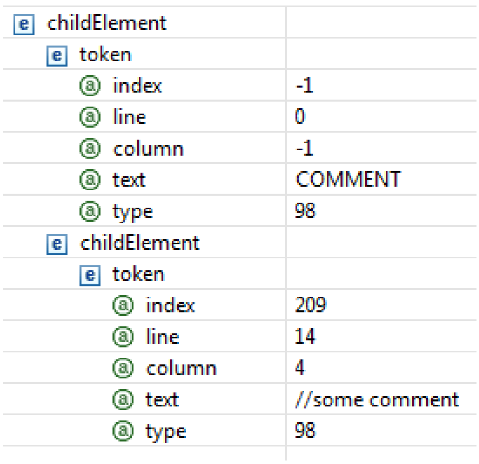


| | |
|---|---|
| childElement | |
| token | |
| column | -1 |
| index | -1 |
| line | 0 |
| text | CONCRETE_UNIT_DECL |
| type | 107 |
| childElement | |
| token | |
| column | -1 |
| index | -1 |
| line | 0 |
| text | NAME |
| type | 118 |
| childElement | |
| token | |
| column | 13 |
| index | 4 |
| line | 1 |
| text | QuickSort |
| type | 139 |

Figure 4.23: Fragment of eCST representing part of class declaration (class name)
Slika 4.23: Fragment u eCST koji predstavlja deo deklaracije klase (ime klase)

- each node in the tree contains information about its position in the source code as line and column attributes (Figure 4.14). Example of fragment of eCST consisting of two universal and one node representing concrete element of the source code (class name) is shown in Figure 4.23. If attributes line and column has value 1 this means that this node is an universal node. Otherwise this token appears in the input source code.

- all concrete syntax elements (tokens) are contained in the tree (section 4.2.1) which means that nothing is omitted (separators, comments, etc.). Each comment is stored to the eCST as one string as a child node of corresponding universal node: COMMENT, LINE_COMMENT and DOC_COMMENT (Figure 4.24).



| | |
|---|---|
| childElement | |
| token | |
| index | -1 |
| line | 0 |
| column | -1 |
| text | COMMENT |
| type | 98 |
| childElement | |
| token | |
| index | 209 |
| line | 14 |
| column | 4 |
| text | //some comment |
| type | 98 |

Figure 4.24: Fragment of eCST representing comments
Slika 4.24: Fragment u eCST koji predstavlja komentar

In these conditions, an implementation of the algorithm of these metrics consists of traversing the tree, taking care of content of each line (each line can contain code, comment, both, or none of it), and accumulating these values.

For example, Listing 4.19 contains example written in C#. For this example these metrics have following values: LOC=32, SLOC=26, CLOC=1 and BLOC=5. Chapter 5 contains more about application of SSQSA framework on real-life examples.

Listing 4.19: Simple example written in C#
Listing 4.19: Jednostavan primer napisan u C#

```
namespace CSharpStudent {
    using System;

    public class Student : Person, IStudent
    {
    private Mark _mark;
        public int StudentNumber { get; set; }

        public Student()
        {
        }

    private decimal CalculateAverageMark(int level)
        {
            /*Business logic*/
      return 0;
        }
    }

  public interface IStudent
  {
    int StudentNumber { get; set; }
  }

  public class Person
  {
    public int Age { get; set; }
    public string Name { get; set; }
    public string Surname { get; set; }
  }
}
```

### 4.5.1.2 Syntax-based analyses

The characteristic of syntax-based analyses is that they rely on syntax of input language. For example, calculation of all complexity metrics need information about concrete syntax elements independently of the fact if it will be done directly on the source code or based on some of internal representations.

An illustrative example of syntax-based analyses is Halstead metrics. It expresses program size and complexity (section 2.2.2.1). Calculation of Halstead metrics are based on number of occurrences of the operators and the operands.

In SSQSA environment the calculation of Halstead metrics is implemented on eCST. While traversing the tree, the implemented algorithm must count the total and distinct number of occurrences of the operators (keywords, operators, and separators) and the operands (variables, constants, types, and directives).

To recognize keywords, operators and separators the algorithm for computing Halstead metrics uses KEYWORD, OPERATOR and SEPARATOR universal nodes, respectively. TYPE, BUILTIN_TYPE, DIRECTIVE and CONST universal nodes are used to identify operands. Initial test cases were selected in that way that generated values can be manually verified. One of such examples (implemented in Delphi) is given in Listing 4.20. Values of Halstead metrics generated for this example are given in Table 4.5.

Listing 4.20: Simple example written in Delphi
Listing 4.20: Jednostavan primer napisan u Delphiu

```
unit Primer;
  interface
    uses Windows, Messages;

  implementation
    uses Azbucnik, Stanje, Saradnici, SarFunkcije;
    {$R *.DFM}

    const a = b + 11;
    var x, y, z: integer;

    procedure TfBioIstorija.FormCreate(Sender: TObject);
    begin
        x := y + z;
    end;

begin
  PrimerProcedure();
end.
```

| Metric | Value |
|---|---:|
| Distinct Operators (n1) | 19 |
| Distinct Operands (n2) | 16 |
| Total Operators (N1) | 41 |
| Total Operands (N2) | 20 |
| Program Vocabulary (n) | 35 |
| Program Length (N) | 61 |
| Program Volume (V) | 312.88626000 |
| Program Level (L) | 0.08421053 |
| Program Difficulty (D) | 11.87499900 |
| Programming Effort (E) | 3715.52420000 |
| Programming Time (T) | 206.41801000 |
| Intelligent Content (I) | 26.34831800 |

Table 4.5: Halstead metrics generated for example in Listing 4.20
Tabela 4.5: Halstead metrike generisane za primer u Listingu 4.20

#### 4.5.1.3 Dependency-based analyses

Dependency-based analyses are selected as analyses whose algorithms implicitly depend on syntax but require generation of some additional information. More specific example is set of design software metrics reflecting relationships between software entities. Calculation of these metrics requires existence of code representation reflecting collaboration between the entities in the code. Implementation of these metrics in SSQSA framework relies on eGDN representation of the source code (section 4.2.2.2) which is derived from eCST based on universal nodes (section 4.3.1.2).

For example, DIT metric (section 2.2.2.1) calculates depth of the inheritance tree. For these purposes eGDN Generator can filter inheritance tree as one of possible middle-level horizontal dimension software network (section 2.2.1). This graph is actually a tree. When inheritance tree is generated then implementation of DIT metrics consists of traversing the tree and counting levels in it.

This example is appropriate to illustrate language and paradigm independence of the implementation. This implementation is immediately applicable to all supported languages independently if they support inheritance or not. If inheritance is not supported, all the nodes of the corresponding graph will have in and out degrees equal to 0 and the graph will not contain edges. Therefore, DIT metric will have the value 0. Furthermore, in some specific cases when declarations are separated from implementations (e.g. in definition and implementation modules in Modula-2) this metric can have a value different from default one (i.e. 1). Even in mentioned cases inheritance in not supported this value reflects the real situation in the source code.

Sub-graph of unit collaboration network (UCN) representing SSQSA libraries for manipulation of eCST and eGDN, and eGDNGenerator is shown in Figure 4.25. This graph is actually inheritance tree of these components. Maximal depth of inheritance tree (DIT) is 3.

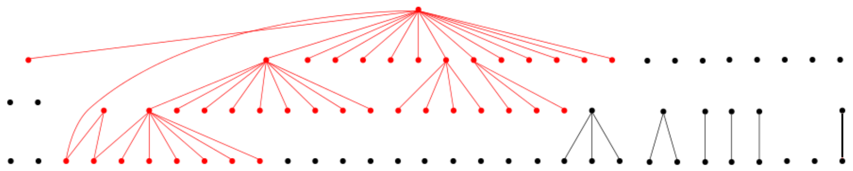


Figure 4.25: Inheritance tree representing SSQSA libraries and eGDGenerator
Slika 4.25: Stablo nasleđivanja koje predstavlja SSQSA biblioteke i eGDN Generator

## 4.6 Summary

SSQSA: Set of Software Quality Static Analyzers is the set of static analysers that are integrated in the framework with the common goal: to achieve consistent software quality analysis. The main characteristic of all integrated analysers is the independency of the input computer language. Language independency is achieved by introducing a new intermediate representation of source code eCST: enriched Concrete Syntax Tree. eCST is designed based on concepts of universal nodes. Furthermore, analysers in the framework rely on alternative intermediate representations of source code derived from eCST. In this section, SSQSA framework and its architecture are described. Furthermore, fundamentals of eCST and description of concept of universal nodes are provided and integrated components and tools are described. Finally, two crucial characteristics of the described framework are described and demonstrated on examples: adaptability and extendability.

# Chapter 5

# Validation and results

In this chapter results of SSQSA framework will be described in two levels. At the first level validation of the results of the SSQSA framework will be demonstrated by application of its components and tools is some representative projects. At the second level, potential benefits of the SSQSA framework for the industry and education will be discussed.

## 5.1 Inspection of SSQSA results

Validation of the results of application of SSQSA framework is a very complex task. Described inconsistency between the results generated by alternative available tools additionally complicates this activity. Therefore, validity of results generated by SSQSA components is observed i two steps:

**Internal consistency check.** This validation step is to be done in order to compare the results generated for the same examples (where possible) written in different languages as demonstrated in [Rakić, 2010] and [Savić et al., 2012]. It is clear that this activity is possible only on small examples as it is usually impossible to find or create equivalent large systems written in different languages.

**External correctness check.** In this step the goal is to find language specific analysers for each supported analysis and compare their results with SSQSA generated ones. This is to be done for each language and each analysis as described in section 5.1.1.3. According to conclusions of available tools review (section 3) two important factors affect this process: (1) inconsistency of the results between two or more external tools and therefore unreliability of the external results, and (2) weak support for majority of languages by the tools. Hence, the main difficulty in this process is related to finding appropriate external tool to compare results. Therefore, most of analysers were externally checked selectively, for languages supported by alternative tools. However, an conclusion about external validity of SSQSA results for all supported languages can be made based on the consistency of SSQSA results among languages which was proved by internal consistency check.

### 5.1.1 Internal consistency check

Internal validation of results generated by SSQSA components will be demonstrated following a basic execution scenario as described in section 4.1 and illustrated in the Figure 4.2. *Pipes and filters* architectural style [Shaw and Garlan, 1996] is selected to describe the information flow and collaboration between the components. Following the execution flow, testing process and results (pipes) of the the components (filters) will be described.

#### 5.1.1.1 Internal validation of eCST Generator

Precondition for application of any analyser integrated in the SSQSA framework is a generated eCST representation of the source code. Correctness of the eCST representation is very important for all the analysers. Therefore, special attention is paid to correctness checking of the eCST generation. To ensure correctness of the generated eCST, eCSTGenerator has to be under constant multiple crosscheck. The reason for this are continual language adaptations and their integrations into the generator.

For each newly introduced language first testing phase is already in the step (1) - writing a language grammar (section 4.4). During the process of writing grammar for a new language, a set of small examples is created so that inspection of each construct defined by language specification, each grammar rule, and each alternative is ensured. Thus, validation of the grammar against the language specification is done. These test cases are used again in testing rules for tree generation and again in the phase of enrichment of the tree by universal nodes. Test cases in this basic set are small enough so that visual inspection is possible (see example in section 4.4). This is the only possible way to validate if the grammar correspond to the language specification. In this way, testing is started early enough and quality of results is importantly improved.

Another level of crosscheck is done by usage of the preliminary defined small examples (e.g. sorting algorithms). These examples are suitable because they are small enough so that they can be re-written in each new language. Important characteristic of these test cases is the tendency to keep equivalency between the implementations of the same algorithm in all languages (as much as possible) so that generated trees stay comparable. By generating, comparing, and correcting the eCST representation for the same set of the small examples, consistency of the results is increased (if not ensured).

Comparisons are done partially automatically by involving SMIILE (section 4.3.2.1) and LICCA components (section 4.3.2.4). Here is important to change the depth of the comparisons during the clone analyses to make sure that possible differences in trees are representing differences in the implementations and that they are not result of the failure of the eCST generator. If the *QuickSort* example and its Java (Listing 5.1) and Modula-2 (Listing 2.1) implementations are observed, results of the duplicates analysis will differ depending on the depth of the comparisons. If it is observed only based on the universal nodes and to the interface or control-flow level, similarity will be close to 99%. The only difference is in the main block which is in Java placed to the function declaration (method), while in Modula-2 this is not the case (Listing 5.2). If the analysis go deeper to the expression level, these values will be lower as expected.

Listing 5.1: *QuickSort* algorithm implemented in Java
Listing 5.1: *QuickSort* algoritam implementiran u Javi

```
public class QuickSort {

    public static unos(String[] args, int[] niz){
      int n = Integer.parseInt(args[0]);
      niz = new int[n];
      for (int i = 0; i < n; i++) {
        do{
          niz[i] = Integer.parseInt(args[i+1]);
        }while (((Object)niz[i]).equals(null));
      }
    }

    public static void ispis(int[] niz){
      int n = niz.length;
      String strN = "";
      for (int i = 0; i < n; i++) {
            strN += niz[i] + ", ";
        }
      System.out.println(strN);
    }

    public static void sort(int[] niz, int levi, int desni){
      int temp;
      int i = levi;
        int j = desni;
        int sredina = niz[(i + j) / 2];
        do{
          while (niz[i] < sredina)
            i++;

          while (niz[j] > sredina)
            j--;

          if (i <= j) {
            temp = niz[i];
            niz[i] = niz[j];
            niz[j] = temp;
            i++;
            j--;
          }
        }while (i <= j);

        if (levi < j)
          sort(niz, levi, j);

        if (i < desni)
```

```
            sort(niz, i, desni);
    }

    public static void qSort(int[] niz)
    {
        sort(niz, 0, niz.length - 1);
    }

    public static void main(String[] args) {
      int[] niz = unos(args);
      System.out.println("Uneti niz je:");
      ispis(niz);
      qSort(niz);
      System.out.println("sortirani niz je:");
      ispis(niz);
    }
}
```

Listing 5.2: *QuickSort* eCST: comparative view (Modula-2 and Java)
Listing 5.2: *QuickSort* eCST: uporedni pregled (Modula-2 i Java)

```
Modula-2                            |Java
==============================      |==============================
CONCRETE_UNIT_DECL                  |CONCRETE_UNIT_DECL
    FUNCTION_DECL                   |    FUNCTION_DECL
        LOOP_STATEMENT              |        LOOP_STATEMENT
            LOOP_STATEMENT          |            LOOP_STATEMENT
    FUNCTION_DECL                   |    FUNCTION_DECL
        LOOP_STATEMENT              |        LOOP_STATEMENT
    FUNCTION_DECL                   |    FUNCTION_DECL
        LOOP_STATEMENT              |        LOOP_STATEMENT
            LOOP_STATEMENT          |            LOOP_STATEMENT
            LOOP_STATEMENT          |            LOOP_STATEMENT
            BRANCH_STATEMENT        |            BRANCH_STATEMENT
                BRANCH              |                BRANCH
        BRANCH_STATEMENT            |        BRANCH_STATEMENT
            BRANCH                  |            BRANCH
        BRANCH_STATEMENT            |        BRANCH_STATEMENT
            BRANCH                  |            BRANCH
    FUNCTION_DECL                   |    FUNCTION_DECL
    BLOCK_SCOPE                     |    FUNCTION_DECL
```

These examples are very important for later testing of other components by comparing the results trying to confirm their consistency. Furthermore, processing them by SMIILE and LICCA components provides additional test for these analysers.

Furthermore, after generating an eCST, eCSTGenerator will check the structure of the generated tree: hierarchy of the entities, existence of the name and type of each entity, etc. The set of the rules that have to be satisfied so that other generators and analysers can be successfully executed is precisely defined. Successful execution completes with message which looks like the one shown in Listing 5.3.

Listing 5.3: Successful eCST Generator execution message
Listing 5.3: Poruka nakon uspešnog izvršavanja eCST Generatora

```
eCSTGenerator started (Mon Feb 23 23:21:19 CET 2015)
InputOutputManipulator warning, outDir ..\test\toy_ecst already exists
Cleaning ..\test\toy_ecst
Creating InputOutputManipulator (Mon Feb 23 23:21:20 CET 2015)
tree: (COMPILATION_UNIT ...)
tree: (COMPILATION_UNIT ...)
tree: (COMPILATION_UNIT ...)
[eCSTGenerator status: OK] - 3 compilation units  successfully parsed
LOC: 251,
LOCwoEmpty: 230,
ecstNodes: 2064
Suspicious node checker [OK]
Empty node checker [OK]
eGDNGenerator compatibility checker [OK]
Metrics compatibility checker [OK]
eCSTGenerator finished (Mon Feb 23 23:21:27 CET 2015)
```

Checks can generate errors or warnings. In case that errors occurred, generated tree does not contain some critical property and component related to that property can not be successfully executed. For example, function declaration does not satisfy the requirement that it is contained in the unit (abstract or concrete). This requires immediate reparation of the tree (i.e. its generation) because eGDNGenerator can not process this tree. If some warnings are listed this means that some unexpected constructions appeared, but all required rules are followed and components can be executed. However the quality of the results can be lowered.

Recommendation is to process some large enough test cases through all phases of development to support a new language, but when language is finally integrated in the eCSTGenerator, real-life examples can be processed. Table 5.1 contains overview of some of the largest real-life test cases written in representative languages successfully processed by eCSTGenerator.

| Software system | Language | Number of | | |
|---|---|---|---|---|
| | | LOC[1] | eCSTs | eCST nodes |
| Tomcat | Java | 329924 | 1083 | 1641488 |
| MAS | Modula-2 | 100546 | 329 | 824043 |
| DelphiProp | Delphi | 104438 | 491 | 1099961 |
| Kernel Compiler | Scheme | 10986 | 1 | 232534 |
| Supermarket | Tempura | 663 | 1 | 9443 |

Table 5.1: Examples of the application of eCSTGenerator
Table 5.1: Primeri primene eCST Generatora

#### 5.1.1.2 Internal validation of eGDN Generator

After the set of XML files containing eCST representation of input source code is generated and stored to the shared storage, the next task is to generate derived internal representations (eCFG and eGDN). At this point analyses applicable to eCST (syntax independent and syntax based ones) can be executed, but as tools partially rely on eGDN, results would not be complete. For example, if SMIILE tool would be applied only on the eCST, it would report that certain set of dependency metrics can not be calculated and will offer to generate it. If user rejects it, SMIILE will calculate only LOC, Halstead, CC and other metrics applicable on the level of eCST. Therefore, next logical step is generation of eGDN (and eCFG). As described in section 4.3.1.2, eCFG is still at a prototype level of development. Therefore, focus of this section is on eGDN Generator.

eGDNGenerator takes a set of XML files containing eCST representation, extracts eGDN, and stores it in XML format. This component takes input whose acceptable level of correctness is already ensured (previous section). Therefore, eGDN generation process is in the main focus of observation. Testing of this component is also divided in steps.

The first step is monitoring of the generation process. This is done by usage of software metrics applied on eCST and afterward calculating the same metrics on the level of eGDN. These are the design metrics applicable on both representations: eCST and eGDN representation (number of packages, number of units, number of attributes, number of functions, number of children, number of function calls, etc.). By comparing these values calculated on two levels, correctness of generated representation is checked. This testing process has double value: apart from testing eGDN generation process, two implementations of the applied metrics are tested. If some differences are found all involved aspects of SSQSA framework are checked in order to locate and solve the issue.

As an illustrative test example, eGDN Generator is selected as an internal and familiar component. Table 5.2 contains values of design metrics applied on the two code representations: eCST and eGDN.

| Metric | Value calculated on | |
|---|---|---|
| | eCST | eGDN |
| NoCU: Number of Compilation Units | 14 | 14 |
| NoP: Number of Packages | 2 | 2 |
| NoU: Number of Units (concrete and interface) | 15 | 15 |
| NoF: Number of Functions | 124 | 124 |
| NoFC: Number of Function Calls | 1419 | 1419 |

Table 5.2: Values of Design metrics calculated on eCST and eGDN representation
Tablela 5.2: Vrednosti metrika dizajna izračnatih na osnovu eCST i eGDN reprezentacijom

The next step in testing eGDN Generator is consistency check. This is done based on already used set of test cases used in testing of consistency of eCST Generation process among the languages. Therefore these examples are enriched by dependencies. Tendency to make equivalency of the realisations between the languages remains which is not always an easy task. Therefore, this process has some limitations. These limitations are partially eliminated by involving software metrics as described. An additional limitation is that these examples are always very simple (Figures 4.8 and 4.9) and do not reflect real characteristics of the software networks so that they can not enable reliable test for network analyses (SNAIPL tool).

#### 5.1.1.3 Internal validation of static analysers

After internal representations are generated and stored to shared data storage, different static analysers can be applied uniformly. This means that each analyser has only one implementation of analysis algorithm independently of input language because it is executed on the unique, language independent representation of the source code.

All results produced by static analysers are also collected to shared data storage. All these data together are used by all tools involved in SSQSA framework. This enables collaboration of the analysers in the service of achieving better quality analysis results. Example of such collaboration is usage of metric results provided by SMIILE in network analysis implemented by SNAIPL tool. Apart of static analysers stored data are used by peripheral tools integrated in SSQSA framework as consumer of static analysis results.

Collaborative application of SSQSA components give an importance to the execution order. Still components are not always fully dependent on collaborative results. For example, if SNAIPL does not receive on input metric result it still can execute statistical analyses of the networks based on eGDN structure if eGDN is generated. Furthermore it can exploit needed components to complement the set of information in order to complete the analyses.

Each analyser has to be tested separately and correctness of each implemented algorithm has to be ensured. Afterward, additional testing effort is invested in integration testing. As collaboration with other components is in charge of actual component, its integration into the framework has to be tested. As software metrics are basics of other static analysis techniques, description of testing process will be illustrated by the example of SMIILE tool.

Correctness of the implemented analyses is verified on the two levels: internally and externally. Internal verification is done based on comparative analysis of the results generated by the implementations of the multiple versions of the analyses algorithms. For example, CC: Cyclomatic Complexity metric can be implemented based only on the source code or its representation by eCST by counting control-flow predicates in the code. Another algorithm is based on CFG (in SSQSA terminology eCFG) representation of the source code. Implementing these two algorithms and comparing the results quality of the CC implementation is monitored. This example is used for double check in monitoring eCFG building process because differences in the CC values can signalize faults in one of these two domains.

For example, *BubbleSort* algorithm implemented in Java (Listing 4.1) is represented by eCST (Figure 4.5) and by eCFG (Figure 4.7). CC value in both cases is 5.

Similar example is used in previous section as an illustration of the correctness verification of eGDN generation process (Table 5.2). In this case, basic set of design metrics is used as a double test: for eGDN extraction and design metric calculation.

Another dimension of internal verification is a consistency check between the languages. Already mentioned equivalent implementations of small examples (section 5.1.1.1) are processed by analysers (in this case by SMIILE tool) and metric results are compared among the languages. An illustrative example is *QuickSort* algorithm implemented in different languages. Listing 5.2 provides the comparative overview of the relevant constructs (universal nodes) in Java and Modula-2 implementations for CC calculation. Table 5.3 contains CC metric results for *QuickSort* implementations in various representative languages supported by SSQSA. It can be mentioned that Tempura and Erlang code have pretty different values. This is caused by simulating loops by recursive calls which is characteristic of declarative style of programming. As other languages also support recursion, these simulations are not marked as loop statement to keep consistency. However, this issue did not stay unprocessed (section 6.1).

| Language | Listing | CC |
|---|---|---|
| Java | 5.1 | 7 |
| Modula-2 | 2.1 | 7 |
| COBOL | 2.9 | 7 |
| PHP | 2.8 | 7 |
| Erlang | 2.3 | 4 |
| Tempura | 2.6 | 2 |

Table 5.3: Values of CC metric calculated for *QuickSort* implementation in various languages
Tabela 5.3: Vrednosti CC metrike za *QuickSort* implementacije u raznim jezicima

### 5.1.2 External correctness check

External validation consists of application of SSQSA components to real-life software systems. Afterwards, gained results are compared with results gained after application of external tools on the same examples. The main difficulty in this process is related to finding appropriate external tool for comparison of the results. Two factors affect this: (1) inconsistency of the results between two or more external tools and therefore unreliability of the external results, and (2) weak support for majority of languages by the tools (section 3). The most appropriate validation is possible for software systems written in Java. However, validity of SSQSA results for other languages can be derived from the consistency of SSQSA results among all supported languages which was proved by internal validation. External validation will be demonstrated by examples of application of eGDN Generator and SMIILE tool.

#### 5.1.2.1 External validation of eGDN Generator

Testing of the eGDN extraction process on the large (real-life) examples is done by comparing extraction results with results gained from other software networks generation tool. As support for language-independent network extraction is pretty weak, results of eGDN Generator are compared to results of the language dependent tools. It is sometimes hard to find the tool supporting specific language. For example, currently there is no available dependency extraction tool able to process Modula-2 source code, while the support for Delphi is available only by commercial and closed-source dependency extractors. The best support is available for Java programming language. Therefore, statement on correctness of the eGDN Generator is mainly based on comparing result of network extraction from source code written in Java. Still, available real-life examples written in supported languages are processed and results are analysed. An overview of some of the largest real-life test cases written in representative languages successfully processed by eGDN Generator is given in the table 5.4.

| Software system | Language | Number of | |
|---|---|---|---|
| | | GDN nodes | GDN links |
| Tomcat | Java | 23118 | 64545 |
| MAS | Modula-2 | 6857 | 31193 |
| DelphiProp | Delphi | 13721 | 18153 |
| Kernel Compiler | Scheme | 1334 | 4743 |
| Supermarket | Tempura | 51 | 191 |

Table 5.4: Examples of the application of eGDN Generator
Tabela 5.4: Primeri primene eGDN Generatora

In order to investigate the correctness and completeness of the eGDN extraction process, real-world open-source software systems mostly written in Java are observed. Results of application of eGDN Generator (eGDNG) were compared with the class collaboration networks extracted by a Java language dependent tool Dependency Finder[2] (DF) and Doxygen [3](DX) a language-independent tool currently supporting C, C++, C#, Objective-C, Java, JavaScript, D, PHP and IDL.

The characteristics set of 10 real-life examples written in Java, with the extraction results and comparisons is given in [Savić et al., 2014]. Overview of the results is given in Table 5.5 The article describes how the networks are compared and measured with final conclusion that extracted networks are correct. Even all observed products are written in Java, identified correctness is an important contribution to the correctness verification. Based on these explorations and considering previous testing of all the steps in generation of the eGDN it can be concluded that generated eGDN is reliable representation of the source code.

[2]Dependency Finder, online 2015, http://depfind.sourceforge.net/
[3]Doxygen, online 2015, http://www.stack.nl/ dimitri/doxygen/

| Software system | LOC | Number of GDN nodes | | | Number of GDN links | | |
|---|---|---|---|---|---|---|---|
| | | eGDNG. | DF | DX | eGDNG | DF | DX |
| CommonsIO | 25663 | 108 | 108 | 100 | 174 | 174 | 71 |
| Forrest | 4683 | 35 | 35 | 33 | 56 | 52 | 21 |
| PBeans | 8502 | 58 | 58 | 36 | 143 | 144 | 19 |
| Colt | 84592 | 299 | 299 | 228 | 1272 | 1280 | 263 |
| Lucene | 111763 | 789 | 789 | 637 | 3544 | 3606 | 925 |
| Log4j | 43898 | 251 | 251 | 230 | 853 | 853 | 246 |
| Tomcat | 329924 | 1494 | 1487 | 1310 | 6839 | 6832 | 1707 |
| Xerces | 216902 | 876 | 876 | 813 | 4775 | 4677 | 1494 |
| Ant | 219094 | 1175 | 1175 | 1055 | 5521 | 5517 | 1406 |
| JFreeChart | 226623 | 624 | 624 | 597 | 3218 | 3249 | 792 |

Table 5.5: Results of the application of eGDN Generator (eGDNG) in comparison with results gained from Dependency Finder (DF) and Doxygen (DX)
Tabela 5.5: Rezultati primene eGDN Generatora u poređenju sa rezultatima dobijenim od Dependency Finder (DF) and Doxygen (DX)

Therefore, during the evolution of SSQSA framework each new supported language was tested on a broad set of small examples such that representation of most constructs can be observed and analysed [Kolek, 2014], [Savić et al., 2012]. Furthermore, generation of internal representations were tested on large and medium software systems [Rakić et al., 2013b].

#### 5.1.2.2 External validation of SMIILE

External verification is based on comparison of the results with other tools with same functionality. Selected tools are both: language specific, or ones applicable to more than one language. Furthermore, as unstable metric values among the tools were recognised as one of the problems in the area [Lincke et al., 2008], [Novak and Rakić, 2010] preferred option is to compare results with various tools (if available). In this testing phase the accent is on real-life examples, but if some important divergence is detected, small examples are used to locate and eliminate the problem.

An illustrative test example is again eGDN Generator. Values of basic size metrics (LOC: Lines of Code and number of units) are calculated by usage of two external tools (Source Monitor (SM)[4] and Understand (U)[5]) and SMIILE tool integrated in SSQSA framework.Results are equal after application of all three tools (table 5.6).

| | LOC | Units |
|---|---|---|
| Compilation unit | SM/U/SSQSA | S/U/SSQSA |
| Filters.java | 176 | 1 |
| HierarchyTreeAnalyzer | 132 | 1 |
| NetworkQDAnalyzer | 191 | 1 |
| FuncCallResolver | 785 | 1 |
| HardToMatchFunction | 118 | 1 |
| ImportList | 346 | 2 |
| NameResolver | 443 | 1 |
| Phase1 | 254 | 1 |
| Phase2 | 80 | 1 |
| Phase3 | 143 | 1 |
| RuneGDNGenerator | 84 | 1 |
| eGDNGenerator.java | 331 | 1 |
| SymbolTableSearch | 411 | 1 |
| TypeResolver | 637 | 1 |
| Total | 4131 | 15 |

Table 5.6: Value of size metrics for the test example
Tabela 5.6: Vrednosti metrika veličine testnog primera

In SSQSA framework, complexity of the source code is observed by two characteristic metric sets: Halstead and CC: Cyclometic Complexity based metrics. Correctness check for Halstead metrics is very difficult task because of two main obstacles: (1) it is hard to find available tools calculating Halstead metrics, and (2) there are many open questions concerning rules for calculating these metrics.

The first and the main difficulty faced during external checking of Halstead metrics gained from SMIILE tool is unavailability of alternative tools to which the results would be compared. Available tools compute Halstead metrics only for one language, usually for Java. The other obstacle

[4]SourceMonitor, Campwood Software, 2015, http://www.campwoodsw.com/sourcemonitor.html

[5]Understand, SciTools, online 2015, https://scitools.com/

are inconsistent results among the tools. Generated results are usually completely different which can be explained by different approaches or by different interpretation of the rules for the calculation of the metrics, but sometimes results are not logical and obviously wrong.

Example of the cause for differences in the results are separators of the language (e.g. {, }, etc.). Consequences of such subtle differences in the approaches can be illustrated by the following example:

Listing 5.4: Inconsistency of Halstead operators - example
Listing 5.4: Nekonzistentnost Halstead operatora - primer

```
if  (l.getSrc().getType() == GDNNodeType.PACKAGE &&
   l.getDst().getType() == GDNNodeType.PACKAGE)
```

In this example the dot ('.') can be observed as an operator or not, depending on the approach. Consequently, total number of operands in this example may vary from 6 to 17. Hence, differences between the results gained from the different tools for the large programs will be larger. However, an adjustment of the sets of operators and operands between the tools would result with adjustment of the results, too. Furthermore, in the case of SSQSA frameworks mentioned changes would have effects on all supported languages because consistency has to be retained. Afterwards, changing the approach would result with new differences comparing to other external tools that calculate these metrics applying similar approach as SMIILE tool.

This observation of such specific metric increase benefits of the SSQSA framework. Once, when an approach and interpretation of the considered algorithm is defined, it is applied consistently to all languages. Described justifiable differences in the results among the tools are shown in the tables 5.7, 5.8 and 5.9 where SMIILE tool is compared with CodePro AnalitiX[6]

[6]CodePro Analytix, online 2015, https://developers.google.com/java-dev-tools/codepro/doc/

| Compilation unit | n1: Dist. operators | | n2: Dist. operands | | N1: Total operators | | N2: Total operand | |
|---|---|---|---|---|---|---|---|---|
| | SSQSA | CPA[a] | SSQSA | CPA | SSQSA | CPA | SSQSA | CPA |
| filter/Filters | 29 | 17 | 117 | 109 | 785 | 227 | 1455 | 378 |
| filter/HierarchyTreeAnalyzer | 30 | 15 | 159 | 78 | 862 | 158 | 2579 | 270 |
| filter/NetworkQDAnalyzer | 27 | 17 | 227 | 106 | 1439 | 243 | 4217 | 463 |
| FuncCallResolver | 31 | 22 | 411 | 235 | 3163 | 732 | 9329 | 1389 |
| HardToMatchFunction | 31 | 19 | 421 | 74 | 2345 | 106 | 9999 | 200 |
| ImportList | 32 | 21 | 475 | 119 | 3336 | 341 | 12421 | 656 |
| NameResolver | 31 | 21 | 529 | 147 | 4037 | 458 | 15763 | 844 |
| Phase1 | 32 | 21 | 566 | 112 | 4163 | 220 | 17562 | 466 |
| Phase2 | 27 | 14 | 560 | 44 | 3934 | 49 | 17991 | 110 |
| Phase3 | 26 | 17 | 579 | 76 | 4234 | 106 | 18911 | 223 |
| RuneGDNGenerator | 23 | 11 | 655 | 91 | 4185 | 43 | 19466 | 139 |
| eGDNGenerator | 36 | 22 | 727 | 226 | 5208 | 304 | 21699 | 631 |
| SymbolTableSearch | 26 | 15 | 675 | 54 | 5000 | 124 | 22541 | 229 |
| TypeResolver | 32 | 20 | 765 | 217 | 6550 | 547 | 26111 | 1049 |

Table 5.7: Comarative overview of the values of basic Halstead metrics calculated for the eGDN Generator example by usage of different tools
Tabela 5.7: Uporedni pregled vrednosti osnovnih Halstead metrika za eGDN Generator primer upotrebom različitih oruđa

[a]CPA: CodePro AnalytiX, online 2015, https://developers.google.com/java-dev-tools/codepro/doc/

| Compilation unit | n: Program Vocabulary | | N: Program Length | | $\hat{N}$: Est. program length | |
|---|---|---|---|---|---|---|
| | SSQSA | CPA[a] | SSQSA | CPA | SSQSA | CPA |
| filter/Filters | 146 | 126 | 902 | 487 | 944,71 | 907,22 |
| filter/HierarchyTreeAnalyzer | 189 | 93 | 1021 | 348 | 1309,96 | 548,86 |
| filter/NetworkQDAnalyzer | 254 | 123 | 1666 | 569 | 1905,01 | 782,65 |
| FuncCallResolver | 442 | 257 | 3574 | 1624 | 3722,29 | 1949,09 |
| HardToMatchFunction | 452 | 93 | 2766 | 274 | 3823,72 | 540,21 |
| ImportList | 507 | 140 | 3811 | 775 | 4383,60 | 912,72 |
| NameResolver | 560 | 168 | 4566 | 991 | 4939,51 | 1150,59 |
| Phase1 | 598 | 133 | 4729 | 578 | 5335,88 | 854,66 |
| Phase2 | 587 | 58 | 4494 | 154 | 5240,78 | 293,52 |
| Phase3 | 605 | 93 | 4813 | 299 | 5435,94 | 544,33 |
| RuneGDNGenerator | 678 | 102 | 4840 | 230 | 6231,80 | 630,26 |
| eGDNGenerator | 763 | 248 | 5935 | 857 | 7096,84 | 1865,47 |
| SymbolTableSearch | 701 | 69 | 5675 | 283 | 6466,36 | 369,37 |
| TypeResolver | 797 | 237 | 7315 | 1266 | 7488,18 | 1770,70 |

Table 5.8: Comarative overview of the derived values of Halstead metrics calculated for the eGDN Generator example by usage of different tools (part 1)
Tabela 5.8: Uporedni pregled vrednosti izvedenih Halstead metrika za eGDN Generator primer upotrebom različitih oruđa (1. deo)

[a]CodePro AnalytiX, online 2015, https://developers.google.com/java-dev-tools/codepro/doc/

| Compilation unit | V: Program volume | | D: Difficulty | | E: Programming Effort | |
|---|---|---|---|---|---|---|
| | SSQSA | CPA[a] | SSQSA | CPA | SSQSA | CPA |
| filter/Filters | 6485,22 | 3397,94 | 180,32 | 29,48 | 1169418,51 | 100161,16 |
| filter/HierarchyTreeAnalyzer | 7721,05 | 2275,63 | 243,30 | 25,96 | 1878545,92 | 59078,78 |
| filter/NetworkQDAnalyzer | 13309,15 | 3950,29 | 250,79 | 37,13 | 3337811,37 | 146663,86 |
| FuncCallResolver | 31407,96 | 13001,13 | 351,82 | 65,02 | 11050062,90 | 845295,02 |
| HardToMatchFunction | 24396,62 | 1791,73 | 368,13 | 25,68 | 8981228,45 | 46003,87 |
| ImportList | 34245,04 | 5525,19 | 418,39 | 57,88 | 14327837,87 | 319811,25 |
| NameResolver | 41684,31 | 7325,79 | 461,86 | 60,29 | 19252515,41 | 441640,28 |
| Phase1 | 43620,30 | 4077,95 | 496,45 | 43,69 | 21655400,07 | 178155,58 |
| Phase2 | 41332,29 | 902,13 | 433,71 | 17,50 | 17926294,71 | 15787,26 |
| Phase3 | 44475,93 | 1955,21 | 424,60 | 24,94 | 18884448,59 | 48764,44 |
| RuneGDNGenerator | 45520,88 | 1534,66 | 341,77 | 8,40 | 15557648,43 | 12892,81 |
| eGDNGenerator | 56830,83 | 6816,75 | 537,25 | 30,71 | 30532458,68 | 209358,56 |
| SymbolTableSearch | 53647,31 | 1728,71 | 434,12 | 31,81 | 23289529,54 | 54982,66 |
| TypeResolver | 70505,16 | 9987,15 | 546,11 | 48,34 | 38503742,73 | 482788,91 |

Table 5.9: Comarative overview of the derived values of Halstead metrics calculated for the eGDN Generator example by usage of different tools (part 2)
Tabela 5.9: Uporedni pregled vrednosti izvedenih Halstead metrika za eGDN Generator primer upotrebom različitih oruđa (2. deo)

[a]CodePro AnalytiX, online 2015, http:

Results of Cyclomatic Complexity of this component calculated by usage of two external tools (Source Monitor[7] and Understand[8]) are given in table 5.10. There are certain differences in the values observing among the tools. This can be explained by some specific differences in the calculations implemented by the tools. For example, while SMIILE tool takes into account explicitly defined constructors when counting functions and all constructor calls when counting function calls (call of the constructor is also an instantiation), other tools have exclude these values when calculating metrics related to functions (methods in their case).

Characteristic difference in the results of CC values is caused by always open topic on adaptations of basic CC algorithm to different conditions in programing practice. For example, jump statements (break, return, etc. in Java) has to affect CC in some extent. Open question is in which cases different jumps increase the complexity. SMIILE follows recommendations from the practice that jump statement should affect complexity only in the case when it causes creation of an extra path in the program execution. Jump statements are currently under testing and deeper consideration in te SSQSA framework as a part of control-flow analyses, and the results of the investigation can cause some changes in current approach to CC calculation. Currently, jump statements is affecting CC only if it is not the last statements and there are alternative path to be taken otherwise.

Another characteristic difference in the CC values is caused by different approaches when observed conditions for loops and branching. SMIILE increases complexity whenever finds some logical operation in the condition.

[7]SourceMonitor, Campwood Software, 2015, http://www.campwoodsw.com/sourcemonitor.html

[8]Understand, SciTools, online 2015, https://scitools.com/

| Compilation unit | Functions | | | Max CC | | | Total CC | | | Average CC | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | SM[a] | U[b] | SSQSA | SM | U | SSQSA | SM | U | SSQSA | SM | U | SSQSA |
| Filters.java | 6 | 6 | 6 | 18 | 16 | 21 | 37 | 34 | 45 | 6,17 | 6 | 7,50 |
| HierarchyTreeAnalyzer | 4 | 4 | 4 | 8 | 8 | 9 | 17 | 17 | 21 | 4,25 | 4 | 5,25 |
| NetworkQDAnalyzer | 8 | 8 | 8 | 5 | 5 | 5 | 20 | 20 | 20 | 2,50 | 3 | 2,50 |
| FuncCallResolver | 17 | 18 | 18 | 31 | 28 | 31 | 138 | 140 | 138 | 8,12 | 8 | 7,67 |
| HardToMatchFunction | 5 | 5 | 5 | 11 | 11 | 9 | 20 | 20 | 17 | 4,00 | 4 | 3,40 |
| ImportList | 16 | 16 | 16 | 20 | 20 | 18 | 70 | 67 | 66 | 4,38 | 4 | 4,13 |
| NameResolver | 11 | 11 | 11 | 40 | 38 | 42 | 102 | 98 | 96 | 9,27 | 9 | 8,73 |
| Phase1 | 10 | 10 | 10 | 16 | 12 | 15 | 51 | 41 | 55 | 5,10 | 4 | 5,50 |
| Phase2 | 4 | 4 | 4 | 8 | 7 | 9 | 11 | 10 | 12 | 2,75 | 3 | 3,00 |
| Phase3 | 7 | 7 | 7 | 13 | 12 | 18 | 24 | 22 | 28 | 3,43 | 3 | 4,00 |
| RuneGDNGenerator | 2 | 2 | 2 | 3 | 3 | 3 | 5 | 5 | 5 | 2,50 | 3 | 2,50 |
| eGDNGenerator.java | 10 | 10 | 10 | 9 | 7 | 8 | 41 | 36 | 40 | 4,10 | 4 | 4,00 |
| SymbolTableSearch | 5 | 5 | 5 | 19 | 18 | 20 | 23 | 22 | 24 | 4,60 | 4 | 4,80 |
| TypeResolver | 18 | 18 | 18 | 25 | 25 | 25 | 131 | 121 | 116 | 7,28 | 7 | 6,44 |
| Total | 123 | 124 | 124 | 226 | 210 | 233 | 690 | 653 | 683 | 5,61 | 5 | 5,51 |

Table 5.10: Comarative overview of the values of CC metrics calculated for the eGDN Generator example by usage of different tools
Tabela 5.10: Uporedni pregled vrednosti CC metrike za eGDN Generator primer upotrebom različitih oruđa

[a]SourceMonitor, Campwood Software, online 2015, http://www.campwoodsw.com/sourcemonitor.html
[b]Understand, SciTools, online 2015, https://scitools.com/

## 5.2 SSQSA potentials

Due to described characteristics, but primarily its two-dimensional extensibility, SSQSA framework can be easily adapted to the needs of management of any software company or software department. Such tool can be used to manage resources and processes during the development of software products as well as to monitor and control the quality of the product. The main advantage of application of SSQSA framework in these processes is consistency of all taken observation and analysis. Applicability of the SSQSA framework in quality monitoring in real-life projects is partially demonstrated on the example of the self-assessment in development and testing of the components and their integration.

Furthermore, present framework can be used in education at different levels.

- SSQSA framework can be applied in automated assessment of student programming solutions. The benefit of this application of the framework is double:

  **In progress benefit.** Automating the assessment process enables greater consistency and eliminates the possibility of a subjective evaluation of student solutions. In addition, teachers have more time to work with students emphasizing specific problems.

  **After progress benefit.** Using the tools suitable for the application in the industry allows students an early introduction to the technologies used in a real working environment and getting used to the tools for evaluating the quality of their work and the impact of the project. This contributes to the completeness of the competencies of students.

- The presented framework is a suitable basis for the inclusion of students in the development of certain functionalities. It is suitable for individual supervised work, with a wide range of topics available: from compiler construction and computational linguistics to advanced topics in the field of software engineering with elements of XML technology and a variety of other areas. It is possible to include students from all three levels of study.

**Undergraduate students** are planned as developers of technical solutions. Based on exactly specified requirements they need to develop specific functionality.

**Master students** are expected to invest a little research effort. Therefore, based on documentation and description of the framework, they are required to explore and choose the most appropriate approach to the presented problem and to implement the required functionality following selected approach.

**Doctoral students** must demonstrate the ability to identify the topic that they will deal with in the framework, and based on their own research to lead a small team of students of lower levels of study through the research and development process for the selected functionality.

Described application in education is already implemented through the various courses and achieved results are witnesses of this. One successful examples of such course is *ATSE: Advanced Topics in Software Engineering*[9] at master and doctoral level of studies.

### 5.2.1 ATSE and project-oriented teaching

Education in the field of computer science and software engineering requires engagement of the students in practice. Practical activities of students can be organized in companies, but in this case there exists a risk that students will become tightly related to the company and technologies used in it. This technology-oriented development of the personnel is for the company shorter way to the future employees. For the students it is a shorter way to the future job. Still, this is pretty risky option for the students. They

[9]ATSE: Advanced Topics in Software Engineering 2015, http://www2.informatik.hu-berlin.de/swt/intkoop/daad/bans2013/index.html#atse

risk to stay with a limited knowledge and experience. Changing between the companies during the studies in order to expand horizons is often very difficult task for the students. Finally, some universities still have a weak connection with the industry and can not offer very rich experience in the practice through the courses.

In the described conditions, project-oriented courses can importantly increase the quality of the practical education in the field. During these courses students are included in real-life projects and participate in it through different levels of the studies. Early inclusion into the project and growth through the years of studies, observing the problem from the different viewpoints, passing through the different roles in the project organisation, and cooperation with other students and lecturers provide students with very rich experience. To keep the continuity and broadness of the topics, approaches, and technologies covered by the project oriented-courses, without switching between projects when moving to the each next course, appropriate project is necessary.

SSQSA framework has all the important characteristics of appropriate project for project-oriented teaching. The main course organised around the SSQA framework is *ATSE: Advanced Topics in Software Engineering.* This course is delivered at fourth year of studies when students have the main prerequisites to successfully participate in the project. At this point of studies, students have passed all elementary programming courses and have basic knowledge about software engineering.

After introductory presentations about architecture of the SSQSA framework and related topics, student can chose the aspect of SSQSA they are interested in. The offered aspects are:

- front-end development which means adapting the framework to new languages and domains. This aspect requires basic knowledge in compiler construction and computer languages.

- back-end development which means extending the framework with new analyses. This aspect requires basic knowledge about static analysis techniques.

- external development which means integration of the framework with external tool. This aspect requires the knowledge in the field of application. Possible application field are project management, information storage and manipulation, visualisation, educational tools, etc.

- maintenance of the framework which means integration of the newly developed functionalities and continual work on the improvement of the framework quality. This is very important activity because of variety of developers with different levels of knowledge, different experience, different (sometimes unexpected) ideas, etc. This becomes easier task with the component-based organisation of the framework based on pipes and filters architectural style.

The course on ATSE is usually the first effort invested by students in the SSQSA framework. After this course students are familiar with the framework and become curious. Furthermore, during the semester they got some new knowledge and become interested in other aspects of the SSQSA.

Additionally, another fourth-year course *ADP: Architecture, Design and Patterns* is also organised around SSQSA. At the begining of the course, students work on an architecture of a (at that moment unknown) framework for static analyses based on its description. After that they receive the real architecture and source code of the early prototypes of some components. Their task is to restructure it according to the architecture and to refactor it. The last task in this course is to add some analyses with obligated usage of appropriate design patterns.

Some other courses where SSQSA framework can be used are:

- compiler construction advanced level, where emphasis is on parser generators and languages;

- software project and student practice with emphasis on practical work and development of SSQSA components;

- seminar work and research methods with accent on research related to characteristics of SSQSA framework;

- different courses at doctoral level (Advanced topic in Software Engineering, Software Quality, Research Methods, etc.) where students are required to be leaders of the groups or teams of younger students. They are expected to guide the teams through their research and development process. Finally, following the principles learned during the studies, students write research papers to describe the results.

. Obviously, activities on SSQSA framework are related to many other study subjects such are communication skills, databases, XML technologies, parallel programming, etc.

SSQSA framework and teamwork organized around it, make good environment for realisation of student practice. In this case students would be guided through the real-life project and all processes, but with possibility to follow their work personally by examiner. Additional advantage in this case is that all students participating in SSQSA team projects have the same working environment, equivalent conditions, and therefore differences in their results are less affected by different attributes of working conditions in various companies. Thus the higher fairness of the assessment of results will be gained.

## 5.3 Summary

Chapter described the process of inspection of results gained by application of SSQSA framework. This is done on two levels: (1) internally by checking consistency of the generated results, and (2) externally by checking correctness of the results. Correctness is observed by comparing results with available external tools from the same domain. External validation is demonstrated on large examples written only in Java because of described limitations of available alternative tools. However, this proves the correctness for all supported languages because of demonstrated consistency. Furthermore this chapter describes triple potential benefits from SSQSA framework: for industry, for education and implicitly for the science.

# Chapter 6

# Conclusion

In this thesis SSQSA: Set of Software Quality Static Analysers framework has been described. The whole SSQSA system and its particular parts are in-house products aimed for academic research in the field of software quality. The final goal of the SSQSA framework is a consistent software quality monitoring and control on code level. This goal is gained by involving new intermediate source code representation called eCST: enriched Concrete Syntax Tree. Therefore the thesis emphasizes on description of eCST and intermediate representations derived from it (section 4.2). The main contribution of the thesis is an infrastructure established to enable further development of fully functional platform for software quality monitoring and control. The infrastructure is based on concepts which are proved by integration of characteristic functionalities, while certain extensions are still to be integrated. However, procedures to be followed through these integrations are precisely stated and described in the thesis. Finally, when consistent results of static analysis techniques are obtained, SSQSA framework can be strengthen by involving intelligent techniques and following the quality standards in order to gain useful feedback instead of only numerical values.

The main characteristic of eCST is language independence. This characteristic gives to SSQSA framework two-dimensional flexibility: by adding support for new language, and by adding support for new analysis. All integrated static analyses tools take as an input eCST or code and design representations derived from it. Consequently all integrated tools are applicable to all programming languages that are supported by the SSQSA system and any combination of these languages. This ensures consistency of all implemented analyses. Furthermore, it promotes efficiency in development of tools even in analyses that will be done only on one input language. It is enough to include a new language in the framework, and set of already implemented tools are immediately available. Similarly, after the new tool is implemented, it is immediately applicable to a whole range of included languages.

Currently, SSQSA framework supports some characteristic input languages such are Java, C#, Delphi, COBOL, C, Pascal, Modula-2, Scheme, Erlang, Python, PHP, JavaScript, OWL, WSL, and Tempura. Validation of correctness and consistency are at different levels among the tools and languages. The set of supported languages is not finite because new languages are continuously integrating, but the languages which are not explicitly enumerated in the above list are still in the early phases of the integration. However, at least on language from each category described in section 2.3 i integrated in the framework.

Using eCST as an input for static analysis tools proves its usefulness in theory and in practice. However, its potential in industrial and educational applications is huge (section 5.2). Software metrics calculation, code clone detection, structural changes analysis and software network analyses are static analysis techniques developed to prove the concepts. Some of these tools are still on level of prototype and will be improved as a part of future work (section 6.1).

Correctness cross-check of the intermediate representation generation process and analysers has been done on simple examples covering main language constructs building cases and afterward on large and medium size software systems (section 5).

## 6.1 Future work

The main and continual activity related to the front-end development is languages integration and consolidation. Variety of supported languages opened some questions. One example is mentioned simulation of loops by recursion. This topic is currently under an investigation. The main point is to keep the consistency, but not to skip this information in measuring complexity. Therefore, a new software metric is to be involved.

The first next activity at the back-end side of SSQSA development will be upgrading the tools which are still on a prototype level such is code clone detector. Additional testing is to be done also for all the analysers integrated in the framework.

Even if absolute validation is not really possible two options to ensure the correct and consistent results are available: internal consistency validation and external correctness check. Both validation steps were partially done during the development of the tools. Application of SSQSA to large and complex industrial software systems is very important in further testing.

Extension of eCST to enable some domain specific analysis could be very valuable. Some of possible directions are static timing analysis and agent activity analysis on the code level.

Furthermore, we can notice that eCST possibly could be used for automatic source code translation between programming languages, but also full source code representation including networks could be used in model transformations.

Another complex task is to develop (semi)automatic refactoring tool which would use as much of SSQSA results as possible to find anomalies in code or design. Possibly, this tool would involve intelligent techniques to decide which improvement approach to chose. As this task is complex enough it can be distributed in steps. First steps are already done by development of appropriate analysers, but also by development of eCST AntiGenerator tool which can generate source code from eCST, originally generated or changed after refinements. Refinement could also be done semi-automatically. In this case at least some recommendations for code and design improvements could be done based on results of the analysers, while employment of intelligent techniques can be replaced by human reasoning in the early stages of prototyping development.

Adding support for more computer languages including declarative, even modelling languages will continue. Although complicated at some times, this should be fairly straightforward activity.

Still, there is more space to investigate additional possibilities in this direction. We can consider other parser generators and their mechanisms to enrich syntax trees. Furthermore, development of translator through different grammar notations would make adding support for new language much easier. Usually we can find grammar written in some notation, but it has to be translated to ANTLR notation. And finally, development of visual tools to support front-end and back-end development would be valuable. For example, visualisation of adding nodes in the tree would make easier to any user to add new language or new analyser by adding nodes for these purposes.

Further processing of the obtained results and visualisation of the feedback is final goal of the SSQSA development as described in the introduction to this chapter. The first prototype of eCST Visualisator tool is described in [Škatarić, 2012].

## 6.2 Summary

The last chapter of the thesis summarises the results and the contribution of SSQSA framework confirms achievement of the stated goals. Finally, focus is moved to the open topics for the future work.

# Appendix A

# Catalog of universal nodes

This appendix provides description of universal nodes and current state of mapping of the certain nodes to language specific constructs. As consolidation of the languages is continual activity, set of nodes can grow and the mapping can change.

| | COMPILATION_UNIT |
|---|---|
| Description | The root node for whole content of a single input file. More precisely it marks each fragment of a source code recognized by the compiler/translator as a self-sufficient unit. |
| Java | |
| C# | |
| Delphi | |
| Modula-2 | |
| Pascal | |
| C | |
| COBOL | |
| WSL | |
| Tempura | |
| Scheme | |
| Erlang | |
| Python | |
| PHP | |
| JavaScript | |
| OWL | |

Table A.1: COMPILATION_UNITuniversal node
Tabela A.1: COMPILATION_UNITuniverzalni čvor

| | PACKAGE_DECL |
|---|---|
| Description | Package/namespace/... |
| Remark | If the package is not defined or the name of the package does not exist in the sub-tree, the package is considered as a default package. If a language does not support packaging, it can be simulated by structuring the files and folders or by introducing default package. |
| Java | package declaration |
| C# | namespace declaration |
| Delphi | unit declaration |
| Modula-2 | - |
| Pascal | - |
| C | preprocessed compilation unit containing header declaration unit and implementation unit |
| COBOL | - |
| WSL | - |
| Tempura | - |
| Scheme | artificially created default package corresponding to all compilation units available in the project |
| Erlang | artificially created default package corresponding to all compilation units available in the project |
| Python | compilation unit |
| PHP | artificially created default package corresponding to all compilation units available in the project |
| JavaScript | artificially created default package corresponding to all compilation units available in the project |
| OWL | ontology declaration |

Table A.2: PACKAGE_DECL universal node
Tabela A.2: PACKAGE_DECL univerzalni čvor

Package, interface and concrete unit declarations are basic entities for analyses. All following universal nodes have to be contained in sub-tree of these modes.

| | INTERFACE_UNIT_DECL | CONCRETE_UNIT_DECL |
|---|---|---|
| Description | Declaration of a software unit not containing implementation | Declaration of a software unit containing implementation |
| Remark | Direct or indirect parents of each of these nodes should be an PACKAGE_DECL nodes, while in sub-tree UNIT has to contain a NAME | |
| Java | interface | class |
| C# | interface | class |
| Delphi | interface in object-oriented concept and interface of the unit | class and implementation of the unit |
| Modula-2 | definition module | implementation module |
| Pascal | - | program |
| C | header declaration unit corresponding to content of a .h file | implementation unit corresponding to content of .c file |
| COBOL | - | procedure division |
| WSL | - | compilation unit |
| Tempura | - | compilation unit |
| Scheme | - | compilation unit |
| Erlang | interface of the module specified by export | module |
| Python | - | class, module |
| PHP | - | compilation unit |
| JavaScript | - | compilation unit |
| OWL | - | - |

Table A.3: INTERFACE_UNIT_DECL and CONCRETE_UNIT_DECL universal nodes
Tabela A.3: INTERFACE_UNIT_DECL and CONCRETE_UNIT_DECL univerzalni čvorovi

| | IMPLEMENTS | EXTENDS |
|---|---|---|
| Description | Indicates that an unit (concrete one usually) implements an interface unit | Indicates that an unit extend another one |
| Remark | These nodes are contained in the sub-tree of an unit declaration. Sub-tree of each of these nodes has to contain the name of the type which is extended/implemented. | |
| Java | IMPLEMENTS | EXTENDS |
| C# | : | : |
| Delphi | In procedural aspect of the language implementation part of the unit by default implements an interface part of the unit. In object-oriented aspect of the language inheritance is inbuilt in the syntax. Syntactical constructs for both cases of inheritance is the same. *type unitToBeDeclared = class(baseUnit)* this is resolved in eCSTAdaptor. *IF baseUnit IS interface THEN IMPLEMENTS ELSE EXTENDS* | |
| Modula-2 | an implementation module by default implements an definition module | - |
| Pascal | - | - |
| C | .c file by default implements entities declared in .h file | - |
| COBOL | - | - |
| WSL | - | - |
| Tempura | - | - |
| Scheme | - | - |
| Erlang | module implementation by default implements the interface specified in export | - |
| Python | Similar as in Delphi *class DerivedClassName(BaseClassName)* | |
| PHP | - | - |
| JavaScript | - | - |
| OWL | - | - |

Table A.4: IMPLEMENTS and EXTENDS universal nodes
Tabela A.4: IMPLEMENTS and EXTENDS univerzalni čvorovi

| | IMPORT_DECL |
|---|---|
| Description | Import of other entities (packages, units, functions, types, etc.) |
| Java | import other entities |
| C# | using other entities |
| Delphi | using other entities |
| Modula-2 | import other entities |
| Pascal | using other enities |
| C | import other entities |
| COBOL | - |
| WSL | - |
| Tempura | load other specification |
| Scheme | import other entities |
| Erlang | import other entities |
| Python | import other entities |
| PHP | - |
| JavaScript | - |
| OWL | import ontology |

Table A.5: IMPORT_DECL universal node
Tabela A.5: IMPORT_DECL univerzalni čvor

| | TYPE_DECL |
|---|---|
| DEscription | Declaration/definition of an user specified type. |
| Remark | — |
| Java | All types defined by users not encompassed by concrete or interface unit declarations. Type is defined by its name. Therefore in the sub-tree this node has to contain a name of the type. |
| C# | |
| Delphi | |
| Modula-2 | |
| Pascal | |
| C | |
| COBOL | |
| WSL | |
| Tempura | |
| Scheme | |
| Erlang | |
| Python | |
| PHP | |
| JavaScript | |
| OWL | |

Table A.6: TYPE_DECL universal node
Tabela A.6: TYPE_DECL univerzalni čvor

| | ATTRIBUTE_DECL (AND CONST) |
|---|---|
| Description | Declaration of global parameters (class fields, global variables, etc.) |
| Remark | If attribute is constant CONST universal node in the sub-tree will mark it. |
| Java | field declaration/definition |
| C# | field declaration/definition |
| Delphi | field declaration/definition, global variable declaration/definition, constant definition |
| Modula-2 | global variable declaration/definition, constant definition |
| Pascal | global variable declaration/definition, constant definition |
| C | global variable declaration/definition, constant definition |
| COBOL | global variable declaration/definition, constant definition |
| WSL | global variable declaration/definition, constant definition |
| Tempura | static variable |
| Scheme | - |
| Erlang | - |
| Python | field declaration/definition, global variable declaration/definition, constant definition |
| PHP | global variable definition |
| JavaScript | global variable definition |
| OWL | property declarations |

Table A.7: ATTRIBUTE_DECL universal node
Tabela A.7: ATTRIBUTE_DECL univerzalni čvor

| | BLOCK_SCOPE |
|---|---|
| Description | Root of the sub-tree containing one block scope |
| Remark | Block of the source code which does not belong to any function declaration corresponds to main block scope |
| Java | code between { and } |
| C# | code between { and } |
| Delphi | code between BEGIN and END |
| Modula-2 | code between BEGIN, DO, THEN, etc. and END |
| Pascal | code between BEGIN and END |
| C | |
| COBOL | code between indent and dedent |
| WSL | code between BEGIN-WHERE and END |
| Tempura | code between { and } |
| Scheme | code between ( and ) in a function definition |
| Erlang | code between ( and ) in a function definition |
| Python | code between indent and dedent |
| PHP | code between { and } or block of the code on the top level (between ¡?php and ?¿) which is not function definition (out of function) |
| JavaScript | code between { and } or block of the code on the top level which is not function definition (out of function) |
| OWL | - |

Table A.8: BLOCK_SCOPE universal node
Tabela A.8: BLOCK_SCOPE univerzalni čvor

| | FUNCTION_DECL |
|---|---|
| Description | Declaration/definition of the method, function, procedure, etc. |
| Remark | This node is contained in the sub-tree of package and/or unit. It has to contain its name in the sub-tree, while type does not have to be specified. |
| Java | declaration/definition of a method |
| C# | declaration/definition of a method |
| Delphi | declaration/definition of a method and/or function |
| Modula-2 | declaration/definition of a procedure/function |
| Pascal | declaration/definition of a procedure/function |
| C | declaration/definition of a procedure |
| COBOL | declaration/definition of a procedure |
| WSL | declaration/definition of a procedure/function |
| Tempura | definition of a procedure/function |
| Scheme | definition of a function (standard, annonymous, let, etc) |
| Erlang | definition of a function |
| Python | definition of a function and/or method |
| PHP | definition of a function |
| JavaScript | definition of a function and/or method |
| OWL | - |

Table A.9: FUNCTION_DECL universal node
Tabela A.9: FUNCTION_DECL univerzalni čvor

| | FORMAL_PARAM_LIST | PARAMETER_DECL |
|---|---|---|
| Description | Root node of a sub-tree containing parameter declarations | One element in the list of declared formal parameters of the function |
| Remark | All formal parameters declared in a header of the function have to be contained in a sub-tree of this node. This node is placed the a sub-tree of the function declaration tree and consists of branches. Each branch contains one formal parameter declaration. | A parameter declaration has to be placed in the sub-tree of the root of all parameter declarations (formal parameter list) Each of these nodes in its sub-tree has to contain the name and the type of the parameter. |

Table A.10: FORMAL_PARAM_LIST and PARAMETER_DECL universal node
Tabela A.10: FORMAL_PARAM_LIST i PARAMETER_DECL univerzalni čvorovi

| | FUNCTION_CALL |
|---|---|
| Description | Call of the function |
| Remark | Function call has to contain name of the function in the sub-tree. Sub-tree of this node has to contain argument list which can be empty. |
| Java | method call |
| C# | method call |
| Delphi | procedure/function/method call |
| Modula-2 | procedure/function call |
| Pascal | procedure/function call |
| C | procedure/function call |
| COBOL | procedure call (explicit bay usage of *CALL* keyword) |
| WSL | procedure/function call |
| Tempura | function call |
| Scheme | function call (basic one or explicit by usage of *apply* keyword |
| Erlang | function call |
| Python | procedure/function/method call |
| PHP | method call |
| JavaScript | method call |
| OWL | - |

Table A.11: FUNCTION_CALL universal node
Tabela A.11: FUNCTION_CALL univerzalni čvor

| | ARGUMENT_LIST | ARGUMENT |
|---|---|---|
| Description | Root node of of sub-tree containing arguments of the function | One element in the list of arguments of the function |
| Remark | All arguments (actual parameters) passed to the function in the function call have to be contained in a sub-tree of this node. This node is placed in the a subtree of the function call sub-tree and consists of branches. Each branch contains one argument. | An argument has to be placed in the sub-tree of the root of all arguments passed to the function (argument list). Each of these nodes in its subtree can to contain the name node and in that case the argument is attribute, variable, constant, etc. |

Table A.12: ARGUMENT_LIST and ARGUMENT universal nodes
Tabela A.12: ARGUMENT_LIST i ARGUMENT univerzalni čvorovi

| | VAR_DECL |
|---|---|
| Description | The root node of the sub-tree containing declaration/definition of one or more local variable. |
| Remark | Local variable is any variable whose visibility is limited to specific block scope (function/main block/...)<br>This node is always contained in the sub-tree of the block scope in which the variable will be visible.<br>In the sub-tree of var declaration/definition the name and the type of the variable have to be specified. |
| Java | |
| C# | |
| Delphi | |
| Modula-2 | |
| Pascal | |
| C | |
| COBOL | |
| WSL | |
| Tempura | |
| Scheme | |
| Erlang | |
| Python | |
| PHP | |
| JavaScript | |
| OWL | |

Table A.13: VAR_DECL universal node
Tabela A.13: VAR_DECL univerzalni čvor

| | STATEMENT |
|---|---|
| Description | The root node of the sub-tree containing any statement. |
| Remark | Statement node is general mark for any statement. In special cases of the statements figuring in analyses, separate statement node will appear in the tree (branch, loop, jump, etc.)<br>This node is always contained in the sub-tree of the block scope. |
| Java | |
| C# | |
| Delphi | |
| Modula-2 | |
| Pascal | |
| C | |
| COBOL | |
| WSL | |
| Tempura | |
| Scheme | |
| Erlang | |
| Python | |
| PHP | |
| JavaScript | |
| OWL | |

Table A.14: STATEMENT universal node
Tabela A.14: STATEMENT univerzalni čvor

| | INSTANTIATES |
|---|---|
| Description | Special kind of statement related primarily to object-oriented. Marks creation of new object which is instance of concrete unit (class). |
| Remark | |
| Java | Instantiation is basically done by call of constructor method often with keyword *new* in front.<br>Some languages support dynamic instantiation which has to be resolved in eCSTAdaptor |
| C# | |
| Delphi | |
| Modula-2 | |
| Pascal | |
| C | |
| COBOL | |
| WSL | |
| Tempura | |
| Scheme | |
| Erlang | |
| Python | |
| PHP | |
| JavaScript | |
| OWL | |

Table A.15: INSTANTIATES universal node
Tabela A.15: INSTANTIATES univerzalni čvor

| | LOOP_STATEMENT |
|---|---|
| Description | The root node of the sub-tree containing any loop statement. |
| Remark | Specialisation of the statement node. This node is always contained in the sub-tree of the block scope. Sub-tree of the loop statement usually contains a condition for continuation or completion of the repetition. |
| Java | for, while-do, repeat-until, and do-while |
| C# | for, foreach, while-do, repeat-until, and do-while |
| Delphi | for, while-do, and repeat-until |
| Modula-2 | for, while-do, repeat-until, and loop |
| Pascal | for, while-do, and repeat-until |
| C | |
| COBOL | *VARYING, TIMES*, and *PERFORM-UNTIL* |
| WSL | *WHILE-DO-OD, DO-OD, FOR-DO-OD, D_DO-OD* |
| Tempura | for, while-do, repeat-until, and chopstar |
| Scheme | do |
| Erlang | - (simulated by recursion) |
| Python | for and while-do |
| PHP | for, foreach, while-do, and do-while |
| JavaScript | for, and do-while |
| OWL | - |

Table A.16: LOOP_STATEMENT universal node
Tabela A.16: LOOP_STATEMENT univerzalni čvor

| | BRANCH_STATEMENT | BRANCH |
|---|---|---|
| Description | The root node of the sub-tree containing any branch statement. | The root node of the sub-tree containing separate branch in a branch statement. |
| Remark | One branch statement can consist of one or more branches. Each branch is marked by separate branch node. Each branch can contain a condition for entering the branch. | |
| Java | if-else, ? - :, switch-case-default, try-catch | |
| C# | if-elseif-else, ? - :, switch-case-default, try-catch, finally | |
| Delphi | if-then-else, case-else, try-except, on | |
| Modula-2 | IF(THEN)-ELSIF(THEN)-ELSE, CASE | |
| Pascal | if-then-else, case-otherwise | |
| C | if-else,switch-case-default | |
| COBOL | WHEN, END-OF-PAGE,(ON) EXCEPTION, INVALID KEY, (ON) OVERFLOW, (ON) SIZE ERROR, IF | |
| WSL | IF-ELSIF-ELSE, D_IF, [] | |
| Tempura | if-then-else | |
| Scheme | case, cond, else, if, unless, when | |
| Erlang | if, case, try-catch, pattern matching | |
| Python | if-elsif-else, try, finally, except | |
| PHP | if-elsif-else, switch | |
| JavaScript | if-else, case-default, try-catch, finally | |
| OWL | - | |

Table A.17: BRANCH_STATEMENT and BRANCH universal nodes
Tabela A.17: BRANCH_STATEMENT i BRANCH univerzalni čvorovi

| | JUMP_STATEMENT |
|---|---|
| Description | The root node of the sub-tree containing any jump statement. |
| Remark | Specialisation of the statement node. This node is always contained in the sub-tree of the block scope. Jump statements usually take part in conditional statements where condition for jumping has to be satisfied. |
| Java | break, return, continue, throw |
| C# | break, return, continue, throw |
| Delphi | abort, break, continue, exit, goto, halt, raise, runerror |
| Modula-2 | EXIT, RETURN |
| Pascal | goto |
| C | |
| COBOL | STOP, EXIT, GO-TO, GO-BACK |
| WSL | EXIT |
| Tempura | - |
| Scheme | continuations |
| Erlang | |
| Python | break, continue, return, pass, raise |
| PHP | break, continue, return |
| JavaScript | break, return, continue, throw |
| OWL | - |

Table A.18: JUMP_STATEMENT universal node
Tabela A.18: JUMP_STATEMENT univerzalni čvor

| | CONDITION |
|---|---|
| Description | Condition which has to be satisfied for execution of some statement(s) - usually related to loops or branchings. |
| Remark | |
| Java | Condition consists of boolean expression.<br>Condition can be precondition or postcondition depending on the moment of evaluation of the expression.<br>Condition can be positive or negative depending on which logical value is expected. For example, for entering the branch condition has to be true, while repeat loop is executed until condition is false.<br>These parameters are related to control-flow and data-flow analysis which is still in the development. |
| C# | |
| Delphi | |
| Modula-2 | |
| Pascal | |
| C | |
| COBOL | |
| WSL | |
| Tempura | |
| Scheme | |
| Erlang | |
| Python | |
| PHP | |
| JavaScript | |
| OWL | |

Table A.19: CONDITION universal node
Tabela A.19: CONDITION univerzalni čvor

| | EXPRESSION |
|---|---|
| Description | Root of the sub-tree containing any expression. |
| Remark | - |

Table A.20: EXPRESSION universal node
Tabela A.20: EXPRESSION univerzalni čvor

| | KEYWORD | OPERATOR | SEPARATOR | BUILTIN_TYPE |
|---|---|---|---|---|
| Description | Elements of the language | | | |
| Remark | Special case of the operators is logical operator (marked by LOGICAL_OPERATOR universal node) figuring in some analyses such is CC:Cyclomatic Complexity. Directives (universal node DIRECTIVE), as part of some languages, are usually eliminated by pre-processing and/or compiling the code after which a native code is being parsed and translated to eCST. | | | |

Table A.21: KEYWORD, OPERATOR, SEPARATOR, and BUILTIN_TYPE universal nodes
Tabela A.21: KEYWORD, OPERATOR, SEPARATOR, i BUILTIN_TYPE univerzalni čvorovi

| | NAME | TYPE |
|---|---|---|
| Description | Name defined by user | Type of the defined name |
| Remark | In dynamically typed languages type can initially have value EMPTY. This can be resolved in eCST Adaptor if needed by assigning the appropriate type. | |

Table A.22: NAME and TYPE universal nodes
Tabela A.22: NAME i TYPE univerzalni čvorovi

| | COMMENT | LINE_COMMENT | DOC_COMMENT |
|---|---|---|---|
| Description | Multi-line comment | Single-line comment | documentation comment |
| Remark | Comments can be marked in eCST Adaptor. | | |

Table A.23: Comments universal nodes
Tabela A.23: Comments univerzalni čvorovi

# Appendix B

# Student example

Listing B.1: *Student* example implemented in Java
Listing B.1: Klasa *Student* implementirana u Javi

```
package JavaTest;

import java.util.Date;
import java.util.Iterator;
import java.util.List;

public class Student extends Person implements IStudent {

  private Schedule _schedule;
  private Mark _mark;

  private int studentNumber;
  private String programName;

  public Student(){

  }

  public int getStudentNumber() {
    return studentNumber;
  }

  public void setStudentNumber(int studentNumber) {
     this.studentNumber = studentNumber;
  }

  public String getProgramName() {
    return programName;
  }

  public void setProgramName(String programName) {
     this.programName = programName;
  }

  public List<Exam> getExams(Date from, Date to)  {

    /*Business logic*/
    return null;
  }

  private double calculateAverageMark(int level)  {

    /*Business logic*/
    return 0;
  }

  public void test1() {
    double t = calculateAverageMark(1);
    List<Exam> e = getExams(new Date(), new Date());
    Iterator<Exam> it = e.iterator();
    while (it.hasNext()) {
      it.next().dummyExam();
    }
    Person per = Person.createJohnDoe();
    _schedule.dummySchedule();
    _mark.dummyMark();
  }
}

interface IStudent{

  int getStudentNumber();
  void setStudentNumber(int studentNumber);

  String getProgramName();
  void setProgramName(String programName);

  List<Exam> getExams(Date from, Date to);
}

class Person {

  private int age;
  private String name;
  private String surname;

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
     this.age = age;
  }

  public String getname() {
    return name;
  }

  public void setName(String name) {
     this.name = name;
  }

  public String getSurname() {
    return surname;
  }

  public void setSurname(String surname) {
     this.surname = surname;
  }

    public static Person createJohnDoe() {
       Person jd = new Person();
       jd.setName("John");
       jd.setSurname("Doe");
       jd.setAge(33);
       return jd;
    }

    public void initJohnDoe() {
        setName("John");
        setSurname("Doe");
        setAge(33);
     }

    public String report() {
      StringBuilder sb = new StringBuilder();
      sb.append(getname()).append(getSurname()).append(getAge());
      return sb.toString();
    }
}
```

```
class Exam {
  public void dummyExam() {}
}
class Schedule {
  public void dummySchedule() {}
}
class Mark {
  public void dummyMark() {}
}
```

Listing B.2: *Student* example implemented in C#
Listing ??: Klasa *Student* implementirana u C#-u

```
namespace CSharpStudent
{
  using System;
  using System.Collections.Generic;

  public class Student : Person, IStudent
  {
    private Schedule _schedule;
    private Mark _mark;
    private int studentNumber;
    private String programName;

    public Student() {
    }

    public int getStudentNumber() {
      return studentNumber;
    }

    public void setStudentNumber(int studentNumber) {
      this.studentNumber = studentNumber;
    }

    public String getProgramName() {
      return programName;
    }

    public void setProgramName(String programName) {
      this.programName = programName;
    }

    public List<Exam> getExams(DateTime from, DateTime to) {
      return null;
    }
  private double calculateAverageMark(int level)  {
    return 0;
  }

  public void test1() {
    double t = calculateAverageMark(1);
    List<Exam> e = getExams(new DateTime(), new DateTime());

    /*Iterator<Exam> it = e.iterator();
    while (it.hasNext()) {
      it.next().dummyExam();
    }*/
    foreach (Exam ex in e) {
      ex.dummyExam();
    }
    Person per = Person.createJohnDoe();
    _schedule.dummySchedule();
    _mark.dummyMark();
  }
}

interface IStudent {
  int getStudentNumber();
  void setStudentNumber(int studentNumber);
  String getProgramName();
  void setProgramName(String programName);
  List<Exam> getExams(DateTime from, DateTime to);
}

public class Person {
  private int age;
  private String name;
  private String surname;

  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  public String getname() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
    }

    public String getSurname() {
      return surname;
    }

    public void setSurname(String surname) {
      this.surname = surname;
    }

    public static Person createJohnDoe() {
       Person jd = new Person();
       jd.setName("John");
       jd.setSurname("Doe");
       jd.setAge(33);
       return jd;
    }

    public void initJohnDoe() {
        setName("John");
        setSurname("Doe");
        setAge(33);
     }

    public String report() {
      /*StringBuilder sb = new StringBuilder();
      sb.append(getname()).append(getSurname()).append(getAge());
      return sb.toString();*/

      string s = getname() + getSurname() + getAge();
      return s;
    }
  }

  public class Exam {
  public void dummyExam() {}
  }

  class Schedule {
  public void dummySchedule() {}
  }

  class Mark {
  public void dummyMark() {}
  }
}
```

# Prošireni sažetak

Predmet istraživanja disertacije obuhvata metode za kreiranje prilagodljivog i proširivog okvira za statičku analizu softvera pod nazivom *Skup statičkih analizatora za kvalitet softvera (eng. SSQSA: Set of Software Quality Static Analyzers).* Okvir je nezavisan od ulaznog (kompjuterskog) jezika. Ova karakteristika zasnovana je na univerzalnoj među-reprezentaciji izvornog koda nazvanoj *obogaćeno konkretno sintaksno stablo (eng. eCST: enriched Concrete Syntax Tree).*

SSQSA okvir je realizovan na osnovu precizno definisane softverske arhitekture, koja omogućava laku prilagodljivost novim ulaznim jezicima i proširenje novim algoritmima za statičku analizu, tako da su: (1) sve raspoložive analize odmah primenljive na novi jezik i (2)) svaka nova analiza je odmah primenljiva na sve podržane jezike.

## Uvod

U modernim pristupima razvoju softvera veliki značaj pridaje se kontroli kvaliteta softvera u ranim fazama razvoja. Zbog toga, statička analiza postaje sve značajnija. Takođe, softverski proizvodi postaju heterogeni i sve kompleksniji. Tačnije, danas se razvijaju i koriste softverski proizvodi pisani u više kompjuterskih jezika. Ova heterogenost otežava analizu i kontrolu kvaliteta u toku životnog ciklusa proizvoda. Posebno je ugrožena konzistentnost dobijenih rezultata analize. Takođe, još uvek su prisutne komponente starije i od ovih analiza, razvijene u (u to vreme) aktuelnim jezicima, tako da one neretko ostaju nepokrivene analizom kvaliteta tokom održavanja, dok se o konzistentnosti ne može ni govoriti.

Predmet istraživanja disertacije je ostvarivanje konzistentnosti analize kvaliteta softverskih proizvoda pri primeni na izvorni kod pisan u različitim kompjuterskim jezicima. Istraživanje je usmereno na kreiranje okvira za statičku analizu softverskog proizvoda. Dve su osnovne karakteristike okvira: (1) prilagodljivost različitim ulaznim jezicima i (2) proširivost skupa dostupnih analiza novim algoritmima po potrebi. Ove karakteristike zasnovane su na internoj reprezentaciji izvornog koda koja je nezavisna od ulaznog jezika čime se postiže jedinstvena implementacija analizatora za sve ulazne jezike. Na ovaj način postiže se konzistentnost dobijenih rezultata i pouzdanija analiza kvaliteta softverskog proizvoda, nezavisno od jezika u kom je proizvod razvijan.

U ovoj disertaciji opisan je okvir SSQSA : skup statičkih analizatora za kontrolu kvaliteta softvera (eng. Set of Software Quality Static Analyzers). Namena SSQSA okvira je konzistentna statička analiza. Cilj se postiže uvođenjem nove međureprezentacije izvornog koda nazvane eCST: obogaćeno konkretno sintaksno stablo (eng. enriched Concrete Syntax Tree). Naglasak disertacije je primarno na eCST reprezenataciji koda, reprezentacijama izvedenim iz eCST i procesu njihovog generisanja, sa opisom oruđa angažovanih u ovim procesima.

Osnovna i najbitnija karakteristika eCST reprezenatacije je nezavisnost od jezika u kom je izvorni kod pisan, što SSQSA okviru daje proširivost na dva nivoa: kroz podršku za nove jezike i kroz podršku za nove analize. Ovo dovodi do efikasnog uvođenja funkcionalnosti na oba navedena nivoa, kao i do kozistentnosti uvedenih funkcionalnosti.

Kao dokaz ispravnosti koncepta, podrška za više od 10 ulaznih jezika je uvedena. Takođe, implementirane su karakteristične tehnike statičke analize (izračunavanje oftverskih metrika, otkrivanje duplikata u kodu, itd.) i integrisane u SSQSA okvir.

Na opisani način, postavljanjem SSQSA okvira, obezbeđena je infrastruktura za dalji razvoj kompletne platforme za doslednu kontrolu kvaliteta softvera.

U uvodnom poglavlju disertacije dat je sažet prikaz ciljeva disertacije kao i naznaka konačnih rezultata. Naznačeno je na koji način su ciljevi ispunjeni i okolnosti u kojima je to učinjeno.

U narednom poglavlju izloženi su osnovni pojmovi iz oblasti kvaliteta softvera, statičke analize i kompjuterskih jezika. Pored opšte podele i taksonomije (tehnika, analizatora i jezika), naročita je pažnja posvećena onim tehnikama, jezicima i jezičkim paradigmama koji su realizovani u okviru za statičku analizu.

Sledi poglavlje koje sadrži motivaciju i obrazloženje za odabir opisanog pristupa razvoju SSQSA okvira, sa akcentom na probleme u oblasti analize kvaliteta softverskog proizvoda, alternativne interne među-reprezentacije i dostupna alternativna softverska rešenja razvijena sa sličnim ciljem ili bazirana na sličnim principima i pristupima.

Centralno poglavlje opisuje SSQSA okvir. Prvo je opisana arhitektura okvira. Zatim se akcenat prebacuje na interne međureprezentacije i njihovo generisanje. Sledi opis komponenti po slojevima kako je to opisano arhitekturom. Na samom kraju poglavlja dato je detaljno obrazloženje kako se u pisanom ookviru obezbeđuju dve ključne karakteristike: proširivost i prilagodljivost.

Sledi poglavlje koje opisuje način na koji se vrš validacija istih i pruža uvid mogućnosti primene okvira u privredi i edukaciji, dok su mogućnosti primene u nauci očigledne.

U zaključku se sumiraju rezultati i opisuju mogući pravci za dalja istrazivanja.

Dodaci pružaju dodatne informacije kao što je katalog univerzalnih čvorova sa opisima i načinima mapiranja za pojedine jezike i izvorni kodovi za veće primere korišćene u tekstu.

## Osnove

Kvalitet svakog proizvoda, a samim tim i *kvalitet softverskog proizvoda* može se opisati kao nivo do kog dati proizvod zadovoljava potrebe i zahteve korisnika. Model kvaliteta softvera definisan standardom ISO 9126-1 razlikuje šest atributa kvaliteta softvera. To su: funkcionalnost, upotrebljivost, pouzdanost, efikasnost, prenosivost i lakoća održavanja. ISO 25010 uvodi osam kvalitativnih karakteristika: funkcionalna primerenost, pouzdanost, efikasnost performansi, upotrebljivost, sigurnost, kompatibilnost, lakoća održavanja i prenosivost[1].

Navedeni atributi kvaliteta softvera mogu se pratiti, ocenjivati i kontrolisati od ranih faza razvoja softvera na nivou izvornog koda i drugih statičkih artefakata ili u fazi izvršavanja i testiranja. Ocenjivanje atributa kvaliteta softvera koja se vrši nad izvornim kodom ili nekoj njegovoj internoj reprezentaciji bez izvršavanja programa naziva se statička analiza, dok se u vreme izvršavanja programa vrši dinamička analiza. U savremenim pristupima razvoja softvera veliki značaj se pridaje praćenju i kontroli kvaliteta od ranih faza razvoja čime statička analiza dobija na značaju.

Kako nije moguće upravljati onim što nije moguće izmeriti, numeričko izražavanje činjenica o pojavama i objektima u procesu razvoja softvera, koristi se kao elementarni postupak u procesu praćenja i kontrole kvaliteta. Za merenje atributa kvaliteta softvera koriste se softverske metrike kao osnovna tehnika statičke analize. *Softverska metrika* može biti definisana kao mera koja odražava neko svojstvo softverskog proizvoda ili njegove specifikacije. Vrednost softverske metrike može odražavati svojstvo celog proizvoda ili neke njegove sastavne jedinice.

*Statička analiza* obuhvata i niz tehnika koje se neretko manje ili više oslanjaju na softverske metrike. Svaka od ovih tehnika podrazumeva implementaciju algoritama nad statičkom reprezentacijom programskog koda. Primer ovih analiza su lociranje duplikata u kodu, otkrivanje manjkavosti u dizajnu i implementaciji, praćenje izmena tokom evolucije, priprema podataka za fazu testiranja, detekcija potencijalnih grešaka i sl.

Pojam kompjuterskih jezika označava sve veštačke jezike koji se obrađuju od strane računara. Kompjuterski jezici se kategorišu na osnovu pojavnog oblika, paradigme, namene, faze u toku procesa razvoja softvera u kojoj se koristi, nivoa apstrakcije, itd. Svaka od navedenih kategorizacija je opisana u ovoj sekciji. Polazi se od globalne podele jezika po njihovoj nameni na jezike opšte namene i jezike namenjene specifičnom domenu. Dalje se ova

[1]ISO, 2014 https://www.iso.org

kategorizacija profinjava na kategorije po paradigmi i baziČnom stilu programiranja, sa osvrtom na jezike koji koriste više paradigmi. Kod jezika namenjenih specifičnim domenima pažnja je posvećena karakterističnim predstavnicima ovih domena (specifikacija i modeliranje).

*Programski jezici* čine podskup skupa kompjuterskih jezika. Programski jezici su dizajnirani za pisanje implementacije izvrsnog rešenja posmatranog problema. Programski jezici se kategorišu na isti način i po istim kriterijumima kao kompjuterski jezici.

Pojam *izvorni kod* označava kod pisan u bilo kom kompjuterskom jeziku bez obzira na notaciju, namenu i paradigmu.

## Obrazloženje SSQSA koncepta

U ovom poglavlju se opisuje motivacija za postavljene ciljeve disertacije, istraživanja koja opravdavaju tu motivaciju, kao i poređenje sa drugim pristupima i softverskim rešenjima.

Motivacija ima svoje korene u nesavršenosti oruđa i tehnika za softverske metrike: često nekonzistetnim rezultatima između različitih softverskih oruđa, nedovoljnoj podržanosti za objektno-orijentisane metrike, nemogućnosti da se softverske metrike na konzistentan način primene na više-jezične projekte koji neretko sadrži kod pisan u starim programskim jezicima poput COBOL-a i FORTRAN-a.

Obavljeno je preliminarno istraživanje dostupnih oruđa za softverske metrike. Dostupna oruđa su posmatrana kroz dve grupe karakteristika: (1) podrška za razne jezike i metrike i (2) mogućnosti čuvanja i korišćenja različitih istorijskih činjenica u toku raspoloživih analiza (istorija koda, međurezultata i rezultata, itd.)

Na osnovu ovih istraživanja i uočenih nedostaka, nastao je analizator za softverske metrike SMIILE (eng. Software Metrics Independent on Input LanguagE), nezavisan od kompjuterskog jezika [Rakić and Budimac, 2011], [Rakić, 2010]. Pristup primenjen pri razvoju SMIILE-a će kasnije postati osnova za kreiranje proširivog i prilagodljivog okvira za statičku analizu. Ove karakteristike okvira su posledica upotrebe univerzalne međureprezentacije izvornog koda nazvane obogaćeno konkretno sintaksno stablo (eng. eCST: enriched Concrete Syntax Tree) [Rakić and Budimac, 2011].

Poređenje sa drugim softverskim rešenjima, pristupima i radovima sastoji se iz: (1) poređenja eCST i izvedenih među-reprezentacija sa sa alternativnim reprezentacijama koda i (2) poređenja SSQSA okvira sa dostupnim alternativnim softverskim rešenjima.

Alternativne među-reprezentacije koda pojavljuju se u vidu među-jezika, alternativnih sintaknih stabala, grafovskih reprezentacija ili meta-modela. Najsličniji pristup zabeležen je u predlogu za upotrebu standardnog meta-modela na dva nivoa apstrakcije koji bi bili generisani upotrebom automatski generisanog parsera. Ovi meta-modeli definisani su od strane kod OMG (eng. Object Management Group)[2] koji daje samo opis standardnog meta-modela dok realizacija ideje u vidu konkretne implementacije još uvek nedostaje. Ipak, ovo je potvrda da ideja na kojoj je zasnovana disertacija ima izglede da bude uspešno realizovana.

Alternativna softverska rešenja, sa sličnim ciljem ili pristupem, postoje, ali svako od ovih rešenja ima neki od ograničavajućih faktora kao sto su selektivna mogućnost podrške kada su kompjuterski jezici u pitanju, nekonzistentnost rezultata usled različitih implementacija za različite jezike, itd. Primer jednog ovakvog oruđa je MOOSE platforma [3] za analizu softvera koja se oslanja na FAMIX metamodel [Tichelaar et al., 2000b] nezavisnih od jezika. Ipak ostaje zavisnost od ulaznog jezika u procesu popunjavanja podataka iz izvornog koda u metamodel. Ovo učitavanje se vrši korišćenjem namenskog oruđa za svaki podržani jezik, što dovodi do opravdane sumnje u konzistentnost uvezenih podataka. Dodatno ograničenje je i orijentisanost platforme ka analizi samo objektno-orijentisanog dizajna i koda.

## SSQSA okvir

Okvir SSQSA i njegove karakteristike baziraju se na specifičnoj među-reprezentaciji izvornog koda. Sve komponente okvira sadrže tačno jednu

[2]OMG: Object Management Group, 2014 http://www.omg.org/

[3]The MOOSE book, 2014 http://www.themoosebook.org/book

implementaciju za svaki jezik. Na ovaj način obezbeđuje se konzistentnost analiza kao i proširljivost i prilagodljivost okvira.

eCST, interna među-reprezentaciju koda, omogućava nezavisnost od kompjuterskog jezika. Polazeći od eCST generišu se dve izvedene među-reprezentacije: eGDN (eng. enriched General Dependency Network)[Savić et al., 2014] i eCFG (eng. enriched Control Flow Graph).

*Univerzalni čvorovi* su osnova univerzalnosti eCST reprezentacije, a dizajnirani su tako da jedinstveno za sve jezike obeleže semantiku sintaksnih konstrukcija u izvornom kodu. Dodatno, vodi se računa o minimalnosti skupa univerzalnih čvorova. U tekstu je dat opis koncepta univerzalnih čvorova i opis svih čvorova, kao i metod koji je korišćen da bi se odabrao minimalan skup čvorova stabla eCST tako da omogućava nezavisnost od ulaznog jezika.

Sve komponente predloženog okvira podeljene su po ulozi u okviru na pet nivoa. Prva tri nivoa komponenti namenjena su radu sa internim reprezentacijama koda (generisanje, konverzija i adaptacija). Ove komponente čine prvu kategoriju komponenti. Druga kategorija komponenti čine oruđa za statičku analizu, dok treću kategoriju čine eksterna oruđa integrisana u okvir.

Opisano je kako se u prikazanoj arhitekturi (Slika **??**) obezbeđuje prilagodljivost u odnosu na uključenje novih kompjuterskih jezika, pri čemu su prikazani i karakteristični problemi prilikom uključenja karakterističnih primera jezika za različite klase jezika uzimajući u obzir namenu, paradigmu i stil programiranja.

Peti odeljak prikazuje kako se u predloženom okviru postiže proširivost okvira u odnosu na nove i postojeće statičke analizatore. Ova osobina je demonstrirana na karakterističnim primerima tri različite kategorije analiza: analize bazirane na leksičkim konstrukcijama u kodu, analize bazirane na sintaksnim konstrukcijama u izvornom kodu i analize bazirane na zavisnostima izmedju softverskih entiteta sadržanih u kodu.

## Validacija i rezultati

Ovo poglavlje disertacije prikazuje postupak testiranja i validacije okvira pri dodavanju novog kompjuterskog jezika i novog analizatora. Takođe prikazuje i neke karakteristične rezultate pri primeni okvira na velike programe (reda veličine oko 100,000 redova koda) pisanim u raznim programskim jezicima. Konzistentnost i korektnost rezultata su proveravani: (1) Interno: u odnosu na rezultate nastale primenom SSQSA okvira na programe pisane u različitim jezicima. Zbog prirode problema, ova analiza se može izvršiti samo na manjim i analognim primerima. (2) Eksterno: u odnosu na rezultate drugih analizatora. Ove analize su vršene na srednjim i velikim programima , a upoređivane su vrednosti metrika dizajna, Halstedova metrika i metrika ciklomatske složenosti.

Na osnovu dve vrste opisanih analiza rezultata, pokazuje se da su rezultati dobijeni primenom okvira SSQSA tačni i konzistentni, između ostalog i zato što svi ulazni programi imaju istu unutrašnju reprezentaciju zasnovanu na eCST. Na primer, da bi se pokazala validnost dobijenih vrednosti primene analize X na Modula-2 programu, dovoljno je pokazati validnost dobijenih vrednosti analize Y u (npr. Javi) poređenjem sa nekoliko drugih analizatora. Validnost analize X tada sledi automatski na osnovu činjenice da su programi pisani u Javi i Moduli-2 predstavljeni istom međureprezentacijom.

Opisani okvir očigledno ima primenu u naučnim istraživanjima, ali i mogućnosi dalje primene u privredi i obrazovanju.

## Zaključak

SSQSA okvir za statičku analizu poseduje dve bitne karakteristike: prilagodljivost novom kompjuterskom jeziku i (2) laka proširivost za nove analize. Osobine SSQSA okvira koje su preduslov za obezbeđivanje ovih karakteristika osnov su i za doslednost podržanih analiza.

Navedene karakteristike su prikazane na reprezentativnom skupu podržanih jezika i analiza. Od komjuterskih jezika u okvir su uključeni: Java, C#, Delphi, Modula-2, Pascal, C, COBOL, Erlang, Scheme, Python, PHP, JavaScript, OWL, WSL, i Tempura. Od statičkih analizatora uključeni su analizatori za softverske metrike, softverske mreže [Savić et al., 2014], detekciju klonova i praćenje strukturnih izmena u softveru [Gerlec et al., 2012].

Ovakav okvir predstavlja dobru infrastrukturu za doslednu statičku analizu, ali i dobru polaznu osnovu za budući rad koji se primarno sastoji iz ukljčivanja novih analiza i unapređenja postojećih. Takođe u planu je unapređenje upotrebljivosti okvira različitim interpretacijama dobijenih rezultata analiza. Ove interpretacije mogu biti u vidu vizuelizacije ali i u vidu pružanja saveta korisnicima za unapređenje kvaliteta proizvoda što bi bilo zanovano na upošljavanju inteligentnih tehnika za zaključivanje na osnovu generisanih rezultata.

# Bibliography

[Aho et al., 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Arbuckle, 2011] Arbuckle, T. (2011). Measuring multi-language software evolution: a case study. In *Proc. of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, IWPSE–EVOL '11, pages 91–95, Szeged, Hungary. ACM.

[Baciková et al., 2013] Baciková, M., Porubän, J., and Lakatos, D. (2013). Defining Domain Language of Graphical User Interfaces. In Leal, J. P., Rocha, R., and Simões, A., editors, *Proc. of the 2nd Symposium on Languages, Applications and Technologies*, volume 29 of *OpenAccess Series in Informatics (OASIcs)*, pages 187–202, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[Bär and Ducasse, 1999] Bär, H. and Ducasse, S. (1999). *The FAMOOS Object Oriented Reengineering Handbook.* Karlsruhe, Forschungszentrum Informatik an der Univerzitet.

[Baxter et al., 1998] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 368–377, Bethesda, Maryland.

[Ben-Menachem and Marliss, 1997] Ben-Menachem, M. and Marliss, G. (1997). *Software Quality: Producing Practical, Consistent Software, Slaying the Software Dragon Series.* Boston, International Thomson Computer Press.

[Bergel et al., 2009] Bergel, A., Denier, S., Ducasse, S., Laval, J., Bellingard, F., Vaillergues, P., Balmas, F., and Mordal-Manet, K. (2009). Squale–software quality enhancement. In *Proc. of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*, pages 285–288, Kaiserslautern, Germany. IEEE.

[Bhatt et al., 2012] Bhatt, K., Tarey, V., Patel, P., Mits, K. B., and Ujjain, D. (2012). Analysis of source lines of code (SLOC) metric. *International Journal of Emerging Technology and Advanced Engineering*, 2(5):150–154.

[Boccaletti et al., 2006] Boccaletti, S., Latora, V., Moreno, Y., Chavez, M., and Hwang, D.-U. (2006). Complex networks: Structure and dynamics. *Physics reports*, 424(4):175–308.

[Bourque et al., 2014] Bourque, P., Fairley, R. E., et al. (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0.* IEEE Computer Society Press.

[Capers, 1996] Capers, J. (1996). *Applied software measurement.* McGraw-Hill.

[Chidamber and Kemerer, 1994] Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

[Christodoulakis et al., 1989] Christodoulakis, D., Tsalidis, C., van Gogh, C., and Stinesen, V. (1989). Towards an automated tool for software certification. In *Proc. of the IEEE International Workshop on Architectures, Languages and Algorithms, Tools for Artificial Intelligence*, pages 670–676, Fairfax, Virginia, USA.

[Cohen and Havlin, 2010] Cohen, R. and Havlin, S. (2010). *Complex networks: structure, robustness and function.* Cambridge University Press.

[Dahm, 1998] Dahm, P. (1998). Parser description language – an overview. In Winter, A., Stasch, H., Gimnich, R., and Ebert, J., editors, *GUPRO – Generische Umgebung zum Programmverstehen*, pages 137–156.

[DeMarco, 1986] DeMarco, T. (1986). *Controlling software projects: management, measurement, and estimates.* Prentice Hall PTR.

[Ducasse et al., 1999] Ducasse, S., Rieger, M., and Demeyer, S. (1999). A language independent approach for detecting duplicated code. In *Proc. of the International Conference on Software Maintenance (ICSM)*, pages 109–118, Oxford, England, UK.

[Ebert et al., 2002] Ebert, J., Kullbach, B., Riediger, V., and Winter, A. (2002). {GUPRO} - generic understanding of programs an overview. *Electronic Notes in Theoretical Computer Science*, 72(2):47 – 56.

[Ebert et al., 2008] Ebert, J., Riediger, V., and Winter, A. (2008). Graph technology in reverse engineering, the TGraph approach. In *Proc. of the 10th Workshop Software Reengineering (WSR 2008)*, volume 126 of *GI Lecture Notes in Informatics*, pages 67–81, Bonn, Germany.

[Fenton and Neil, 1999] Fenton, N. E. and Neil, M. (1999). Software metrics: Success, failures and new directions. *Journal of Software and Systems*, 47(2-3):149–157.

[Fenton and Neil, 2000] Fenton, N. E. and Neil, M. (2000). Software metrics: roadmap. In *Proc. of the Conference on the Future of Software Engineering*, pages 357–370, Limerick, Ireland. ACM.

[Fischer et al., 2007] Fischer, G., Lusiardi, J., and von Gudenberg, J. W. (2007). Abstract syntax trees-and their role in model driven software development. In *Proc. of the International Conference on Software Engineering Advances (ICSEA)*, pages 38–38, Cap Esterel, French Riviera, France. IEEE.

[Fowler, 2010] Fowler, M. (2010). *Domain Specific Languages.* Addison-Wesley Professional, 1st edition.

[Gerlec et al., 2012] Gerlec, Č., Rakić, G., Budimac, Z., and Heričko, M. (2012). A programming language independent framework for metrics-based software evolution and analysis. *ComSIS: Computer Science and Information Systems*, 9(3):1155–1186.

[Gerlec and Živkovič, 2009] Gerlec, Č. and Živkovič, A. (2009). Software metrics repository architecture. In *Proc. of the 12th International Multi-conference on Information Society (IS 2009)*, pages 265–268, Ljubljana, Slovenia.

[Gerlec et al., 2011] Gerlec, v., Krajnc, A., Marjan, H., and Jan, B. (2011). Mining source code changes from software repositories. In *Proc. of the 7th Central and Eastern European Software Engineering Conference*, CEE-SECR '11, pages 1–5, Washington, DC, USA. IEEE Computer Society.

[Gilb, 1976] Gilb, T. (1976). *Software Metrics.* Chartwell-Bratt.

[Gustafsson et al., 2009] Gustafsson, J., Ermedahl, A., Lisper, B., Sandberg, C., and Källberg, L. (2009). ALF–a language for WCET flow analysis. In *Proc. of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET2009)*, pages 1–11, Dublin, Ireland.

[Heckerman et al., 2001] Heckerman, D., Chickering, D. M., Meek, C., Rounthwaite, R., and Kadie, C. (2001). Dependency networks for inference, collaborative filtering, and data visualization. *The Journal of Machine Learning Research*, 1:49–75.

[Heričko et al., 2007] Heričko, M., Živkovič, A., and Porkolb, Z. (2007). A method for calculating acknowledged project effort using a quality index. *Informatica*, 31(4):431–436.

[Kan, 2002] Kan, S. H. (2002). *Metrics and models in software quality engineering.* Addison-Wesley Longman Publishing Co., Inc.

[Kienle and Mller, 2010] Kienle, H. M. and Mller, H. A. (2010). Rigian environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming*, 75(4):247 – 263.

[Kolek, 2014] Kolek, J. (2014). Translation of scheme programming language to universal intermediate representation eCST (prevođenje programskog jezika scheme u univerzalnu među-reprezentaciju eCST). MSc thesis, Faculty of Sciences, University of Novi Sad, Serbia (in Serbian).

[Kolek et al., 2013] Kolek, J., Rakić, G., and Savić, M. (2013). Two-dimensional extensibility of SSQSA framework. In *Proc. of the 2nd Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications(SQAMIA)*, pages 35–43, Novi Sad, Serbia.

[Koschke et al., 2006] Koschke, R., Falke, R., and Frenzel, P. (2006). Clone detection using abstract syntax suffix trees. In *Proc. of the 13th Working Conference on Reverse Engineering, (WCRE '06)*, pages 253–262, Benevento, Italy.

[Lanza and Marinescu, 2006] Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer.

[Lattner and Adve, 2004] Lattner, C. and Adve, V. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, page 75, Palo Alto, California. IEEE Computer Society.

[Lincke et al., 2008] Lincke, R., Lundberg, J., and Löwe, W. (2008). Comparing software metrics tools. In *Proc. of the International Symposium on Software Testing and Analysis*, ISSTA '08, pages 131–142, Seattle, WA, USA. ACM.

[Lisper, 2014] Lisper, B. (2014). SWEET - a tool for WCET flow analysis (extended abstract). In Margaria, T. and Steffen, B., editors, *Leveraging*

*Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications*, volume 8803 of *Lecture Notes in Computer Science*, pages 482–485. Springer Berlin Heidelberg.

[Lorenz and Kidd, 1994] Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics: a practical guide.* Prentice-Hall, Inc.

[Madhavji et al., 2006] Madhavji, N. H., Fernandez-Ramil, J., and Perry, D. (2006). *Software Evolution and Feedback: Theory and Practice.* John Wiley & Sons.

[McCabe, 1976] McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320.

[Mernik et al., 2005] Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

[Merrill, 2003] Merrill, J. (2003). Generic and gimple: A new tree representation for entire functions. In *Proc. of the GCC Developers Summit*, pages 171–179, Ottawa, Ontario, Canada.

[Morris, 1989] Morris, K. L. (1989). *Metrics for object-oriented software development environments.* PhD thesis, Massachusetts Institute of Technology.

[Myers, 2003] Myers, C. R. (2003). Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):046116.

[N. Fenton, 1996] N. Fenton, S. L. P. (1996). *Software Metrics: A Rigorous and Practical Approach.* Thomson Computer Press.

[Newman, 2010] Newman, M. (2010). *Networks: An Introduction.* Oxford University Press, Inc., New York, NY, USA.

[Newman, 2003] Newman, M. E. (2003). The structure and function of complex networks. *SIAM review*, 45(2):167–256.

[Novak and Rakić, 2010] Novak, J. and Rakić, G. (2010). Comparison of software metrics tools for: net. In *Procedings of 13th International Multiconference Information Society (IS'10)*, pages 231–234, Ljubljana, Slovenia.

[Parr and Fisher, 2011] Parr, T. and Fisher, K. (2011). Ll (*): the foundation of the antlr parser generator. In *ACM SIGPLAN Notices*, volume 46, pages 425–436. ACM.

[Parr et al., 2014] Parr, T., Harwell, S., and Fisher, K. (2014). Adaptive ll (*) parsing: the power of dynamic analysis. In *Proc. of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pages 579–598, Portland, Oregon, USA. ACM.

[Páter-Részeg, 2013] Páter-Részeg, A. (2013). Mapping the semantic program graph of a functional programming language to the syntax tree of imperative languages (Funkcionális program szemantikus program gráfjának leképezése imperatív program szintaxisfájára) . TDK Thesis, Scientific Students' Associations Conference, ELTE, Budapest, Hungary (in Hungarian, received second prize).

[Pérez-Castillo et al., 2011] Pérez-Castillo, R., De Guzman, I. G.-R., and Piattini, M. (2011). Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Computer Standards & Interfaces*, 33(6):519–532.

[Porubän et al., 2010] Porubän, J., Forgáč, M., Sabo, M., and Běhálek, M. (2010). Annotation based parser generator. *ComSIS: Computer Science and Information Systems*, 7(2):291–307.

[Pribela et al., 2012] Pribela, I., Budimac, Z., and Rakić, G. (2012). First experiences in using software metrics in automated assessment. In *Proc. of the 15th International Multiconference on Information Society (IS 2012)*, pages 250–253, Ljubljana, Slovenia.

[Pribela et al., 2011] Pribela, I., Ivanović, M., and Budimac, Z. (2011). System for testing different kinds of students programming assignments.

In *Proc. of the 5th International Conference on Information Technology ICIT*, Amman, Jordan.

[Raja and Lakshmanan, 2010] Raja, A. and Lakshmanan, D. (2010). Domain specific languages. *International Journal of Computer Applications*, 1(21):99–105.

[Rakić, 2010] Rakić, G. (2010). Software metrics tool independent of programming language (oruđe za softverku metriku nezavisno od programskog jezika). MSc thesis, Faculty of Sciences, University of Novi Sad, Serbia (in Serbian).

[Rakić and Budimac, 2010] Rakić, G. and Budimac, Z. (2010). Problems in systematic application of software metrics and possible solution. In *Proc. of The 5th International Conference on Information Technology (ICIT)*, Amman, Jordana.

[Rakić and Budimac, 2011] Rakić, G. and Budimac, Z. (2011). Introducing enriched concrete syntax trees. In *Proc. of 13th International Multiconference Information Society (IS'13)*, pages 211–214, Ljubljana, Slovenia.

[Rakić and Budimac, 2011] Rakić, G. and Budimac, Z. (2011). SMIILE prototype. *AIP Conference Proceedings*, 1389(1):853–856.

[Rakić et al., 2013a] Rakić, G., Budimac, Z., and Bothe, K. (2013a). Introducing recursive complexity. *AIP Conference Proceedings*, 1558(1):357–361.

[Rakić et al., 2013b] Rakić, G., Budimac, Z., and Savić, M. (2013b). Language independent framework for static code analysis. In *Proc. of the Balkan Conference in Informatics, BCI'13*, pages 236–243, Thessaloniki, Greece. ACM.

[Rakić et al., 2011] Rakić, G., Črt Gerlec, Novak, J., and Budimac, Z. (2011). XML-Based integration of the SMIILE tool prototype and software metrics repository. *AIP Conference Proceedings*, 1389(1):869–872.

[Rattan et al., 2013] Rattan, D., Bhatia, R., and Singh, M. (2013). Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165 – 1199.

[Raza et al., 2006] Raza, A., Vogel, G., and Plödereder, E. (2006). Bauhaus–a tool suite for program analysis and reverse engineering. In *Reliable Software Technologies–Ada-Europe 2006*, pages 71–82. Springer.

[Roy et al., 2009] Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470 – 495. Special Issue on Program Comprehension (ICPC 2008).

[Savić et al., 2013] Savić, M., Budimac, Z., Rakić, G., Ivanović, M., and Heričko, M. (2013). SSQSA ontology metrics front-end. In *Proc. of the Second Workshop on Software Quality Analysis, Monitoring, Improvement and Applications (SQAMIA)*, page 95, Novi Sad, Serbia.

[Savić and Ivanović, 2014] Savić, M. and Ivanović, M. (2014). Graph clustering evaluation metrics as software metrics. In *Proc. of the 3rd Workshop on Software Quality Analysis, Monitoring, Improvement and Applications (SQAMIA)*, pages 81–89, Lovran, Croatia.

[Savić et al., 2011] Savić, M., Ivanović, M., and Radovanović, M. (2011). Characteristics of class collaboration networks in large Java software projects. *Information Technology And Control*, 40(1):48–58.

[Savić et al., 2012] Savić, M., Rakić, G., Budimac, Z., and Ivanović, M. (2012). Extractor of software networks from enriched concrete syntax trees. *AIP Conference Proceedings*, 1479(1):486–489.

[Savić et al., 2014] Savić, M., Rakić, G., Budimac, Z., and Ivanović, M. (2014). A language-independent approach to the extraction of dependencies between source code entities. *Information and Software Technology*, 56(10):1268 – 1288.

[Scotto et al., 2006] Scotto, M., Sillitti, A., Succi, G., and Vernazza, T. (2006). A non-invasive approach to product metrics collection. *Journal of Systems Architecture*, 52(11):668–675.

[Shaw and Garlan, 1996] Shaw, M. and Garlan, D. (1996). *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs.

[Tichelaar et al., 2000a] Tichelaar, S., Ducasse, S., and Demeyer, S. (2000a). FAMIX and XMI. In *Proc. of Seventh Working Conference on Reverse Engineering (WCRE)*, pages 296 –298, Brisbane, Australia.

[Tichelaar et al., 2000b] Tichelaar, S., Ducasse, S., Demeyer, S., and Nierstrasz, O. (2000b). A meta-model for language-independent refactoring. In *Proc. of International Symposium on Principles of Software Evolution*, pages 154 –164, Kanazawa, Japan.

[Tóth et al., 2015] Tóth, M., Páter-Részeg, A., and Rakić, G. (2015). Introducing support for erlang into ssqsa framework. *AIP Conference Proceedings*, 1648(1):310012.

[Škatarić, 2012] Škatarić, V. (2012). Visual eCST editor (vizuelni eCST editor). MSc thesis, Faculty of Sciences, University of Novi Sad, Serbia (in Serbian).

[Šubelj and Bajec, 2012] Šubelj, L. and Bajec, M. (2012). Software systems through complex networks science: Review, analysis and applications. In *Proc. of the First International Workshop on Software Mining*, SoftwareMining '12, pages 9–16, Beijing, China. ACM.

[Wagner, 2014] Wagner, C. (2014). *Model-Driven Software Migration: A Methodology: Reengineering, Recovery and Modernization of Legacy Systems.* Springer Science & Business Media.

[Wilhelm et al., 2008] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., et al. (2008). The worst–case execution–time problem-overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36.

## Short biography

Gordana Rakić was born on October 27, 1981 in Novi Sad. After finishing secondary school in economics (in 2000) she enrolled studies at Faculty of Sciences, Univeristy of Novi Sad. She obtained bachelor degree in *Business Informatics* in 2006 with average mark of 8.58 (max. 10) and defended bachelor thesis entitled "Example of the cost estimation with function points method" (in Serbian) with mark 10. Gordana completed master studies in *Business Informatics* in 2010. with the average mark 9.80 (max. 10), defending the masters thesis entitled "Programming language independent software metrics tool", with the highest mark. In 2008, she enrolled PhD Studies in Informatics at the Faculty of Sciences in Novi Sad. She has passed all the exams with the highest marks (10).

From 2006 to 2009 he was engaged in teaching and research at the Faculty of Sciences in Novi Sad with several roles as a master and PhD student. Since 2009 she has been employed at the position of an assistant for the scientific field of computer science at the Faculty of Sciences in Novi Sad, Department of Mathematics and Informatics. She is involved in courses for students of computer science on the bachelor and master level.

She has participated in the organization of several international scientific conferences, symposia and workshops, as a member, secretary, co-chairman, and chairman of the organizing committees.

In the period 2011 - 2013 she was one of the managing editors of the international scientific journal "ComSIS: Computer Science and Information Systems" in a role of managing editor.

She actively participates in research projects funded by the Ministry of Education, Science and Technological Development of the Republic of Serbia. She participated in several ongoing and completed research projects of bilateral and multilateral cooperation, an international project supported by the DAAD foundation, one Tempus, one COST project, as well as in the CEEPUS research networks.

As a part of research activities, in the period from 2006 until nowadays she has actively participated in the exchange of ideas and results with colleagues from country and abroad. She has received grants research for stays in Germany (Berlin and Munich), Austria (Linz), Slovenia (Maribor), Hungary (Budapest), Bulgaria (Plovdiv) and Italy (Venice).

She has published (as author or co-author) more than 20 scientific papers in the field of software metrics, software quality and software engineering.

She has received awards for outstanding scientific or professional work for papers written in the academic year 2006/2007. She was awarded by the University of Novi Sad, for the paper entitled "Requirement Engineering for Critical Systems".

Novi Sad, March 2015 Gordana Rakić

## Kratka biografija

Gordana Rakić je rođena 27. oktobra 1981. godine u Novom Sadu. 2000. godine je završila je srednju ekonomsku školu. Školovanje nastavlja na Prirodno−matematičkom fakultetu u Novom Sadu u oblasti poslovne informatike. Diplomirala je 2006. godine sa prosečnom ocenom 8.58, odbranivši diplomski rad pod naslovom "Primer procene troškova metodom funkcijskih bodova" sa ocenom 10.

Master studije završila je sa prosečnom ocenom 9.80, dok je master rad pod naslovom "Oruđe za softversku metriku nezavisno od programskog jezika" odbranila 2010. godine ocenom 10. 2008. godine upisuje doktorske studije informatike na Prirodno-matematičkom fakultetu u Novom Sadu. Sve ispite predviđene planom i programom za doktorske studije položila je ocenom 10.

Od 2006. do 2009. godine je bila angažovana u nastavi i istraživačkom radu na Prirodno−matematičkom fakultetu u Novom Sadu po nekoliko osnova kao student master i doktorskih studija. Od 2009. godine je zaposlena na radnom mestu asistenta za užu naučnu oblast Računarske nauke na Departmanu za matematiku i informatiku Prirodno−matematičkog fakulteta u Novom Sadu. Uključena je u izvođenje nastave za osnovnih i master studija informatike.

Učestvovala je u organizaciji nekoliko međunarodnih naučnih konferencija, simpozijuma i radionica kao član, sekretar i ko−predsedavajući i predsedavajući organizacionog odbora.

U periodu 2011. 2013. bila je angažovana na administrativnim poslovima vezanim za uređivanje međunarodnog časopisa (eng. managing editor) "ComSIS: Computer Science and Information Systems"

Aktivno učestvuje na naučnim projektima koje finansira Ministarstvo prosvete, nauke i tehnološkog razvoja Republike Srbije. Učesnik je više tekućih i okončanih bilateralnih naučnih projekata i projekata multilateralne saradnje, jednog međunarodnog projekta podržanog od strane DAAD fondacije, jednog Tempus, jednog COST projekta i CEEPUS istraživačke mreže.

U okviru naučno−istraživačkog rada, uperiodu od 2006. godine do danas aktivno je učestvovala u razmeni ideja i rezultata sa kolegama iz zemlje i inostranstva. Iza sebe ima istraživačke boravke i usavršavanja u Nemačkoj (Berlin i Minhen), Austriji (Linc), Sloveniji (Maribor), Mađarskoj (Budimpešta), Bugarskoj (Plovdiv) i Italiji (Venecija).

Autor je i ko−autor više od 20 naučnih radova u oblasti softverskih metrika, kvaliteta softvera i softverskog inženjerstva.

Dobitnik je Izuzetne nagrade za naučne i stručne radove studenata napisane u akademskoj 2006/2007. godini koju dodeljuje Univerzitet u Novom Sadu, za rad Requirement Engineering for Critical Systems.

Novi Sad, March 2015 Gordana Rakić

**University of Novi Sad**
**Faculty of Science**
**Key Words Documentation**

| | |
|---|---|
| Accession number: | |
| NO | |
| Identification number: | |
| INO | |
| Document type: | Monograph documentation |
| DT | |
| Type of record: | Textual printed material |
| TR | |
| Contents code: | Doctoral dissertation |
| CC | |
| Author: | Gordana Rakić |
| AU | |
| Advisor: | Dr.Zoran Budimac |
| MN | |
| Title: | Extendable and Adaptable Framework for Input Language Independent Static Analysis |
| TI | |
| Language of text: | English |
| LT | |
| Language of abstract | Serbian/English |
| LA | |
| Country of publication: | Republic of Serbia |
| CP | |
| Locality of publication: | Vojvodina |
| LP | |
| Publication year: | 2015 |
| PY | |

| | |
|---|---|
| Publisher: | Author's reprint |
| PU | |
| Publ. place: | Novi Sad, Trg D. Obradovića 4 |
| PP | |
| | |
| Physical description: | 6/XX+214/82/40/27/0/2 |
| (no. of chapters/pages/bib. refs/tables/figures/graphs/appendices) | |
| PO | |
| Scientific field: | Computer Science |
| SF | |
| Scientific discipline: | Software Engineering |
| SD | |
| Subject/Key words: | software quality |
| | static analysis |
| | intermediate representation |
| SKW | |
| UC | |
| Holding data: | |
| HD | |
| Note: | |
| N | |

Abstract: In modern approach to software development, a great importance is given to monitoring of software quality in early development phases. Therefore, static analysis becomes more important. Furthermore, software projects are becoming more complex and heterogeneous. These characteristics are reflected in a diversity of functionalities and variety of computer languages and the technologies used for their development. Because of that consistency in static analysis becomes more important than it was earlier.

In this dissertation SSQSA: Set of Software Quality Static Analyzers is described. The aim of the SSQSA framework is consistent static analysis. This goal is reached by introducing new intermediate source code representation called eCST: enriched Concrete Syntax Tree. The dissertation mostly focuses on eCST, intermediate representations derived from it, and their generation with description of the tools involved in it.

The main characteristic of eCST is language independence which gives to SSQSA framework two-level extensibility: supporting a new language and supporting a new analysis. This leads to efficiency of adding both level supports and consistency of added functionalities.

To prove the concept, support for more than 10 characteristic languages was introduced. Furthermore, characteristic static analysis techniques (software metrics calculation, code-clone detection, etc.) were implemented and integrated in the framework.

Established SSQSA framework provides the infrastructure for the further development of the complete platform for software quality control.

AB
Accepted by Scientific Board on: 11.06.2014.
AS
Defended:

# Univerzitet u Novom Sadu
# Prirodno-matematički fakultet
# Ključna dokumentacijska informacija

| | |
|---|---|
| Redni broj: | |
| RBR | |
| Identifikacioni broj: | |
| IBR | |
| Tip dokumentacije: | Monografska dokumentacija |
| TD | |
| Tip zapisa: | Tekstualni štampani materijal |
| TZ | |
| Vrsta rada: | Doktorska disertacija |
| VR | |
| Autor: | Gordana Rakić |
| AU | |
| Mentor: | dr Zoran Budimac |
| MN | |
| Naslov rada: | Proširiv i prilagodljiv okvir za statičku analizu nezavisnu od ulaznog jezika |
| NR | |
| Jezik publikacije: | engleski |
| JP | |
| Jezik izvoda: | srpski/engleski |
| JI | |
| Zemlja publikovanja: | Republika Srbija |
| ZP | |
| Uže geografsko područje: | Vojvodina |
| UGP | |
| Godina: | 2015 |
| GO | |

| | |
|---|---|
| Izdavač: | autorski reprint |
| IZ | |
| Mesto i adresa: | Novi Sad, Trg D. Obradovića 4 |
| MA | |
| Fizički opis rada: | 6/XX+214/82/40/27/0/2 |
| (broj poglavlja/strana/lit. citata/tabela/slika/grafika/priloga) | |
| FO | |
| Naučna oblast: | Računarske nauke |
| NO | |
| Naučna disciplina: | Softversko inženjerstvo |
| ND | |
| Predmetna odrednica/ Ključne reči: | kvalitet softvera statička analiza međureprezentacija |
| PO | |
| UDK | |
| Čuva se: | |
| ČU | |
| Važna napomena: | |
| VN | |

Izvod:

U modernim pristupima razvoju softvera veliki značaj pridaje se kontroli kvaliteta softvera u ranim fazama razvoja. Zbog toga, statička analiza postaje sve značajnija. Takođe, softverski proizvodi postaju sve kompleksniji i heterogeni. Ove karakteristike se ogledaju u raznovrsnosti jezika i tehnologija koje se koriste u procesu razvoja softvera. Zbog toga, konzistentnost u statičkoj analizi dobija veći značaj nego što je to bio slučaj ranije.

U ovoj disertaciji opisan je SSQSA skup statičkih analizatora za kontrolu kvaliteta (eng. Set of Software Quality Static Analyzers). Namena SSQSA okvira je konzistentna statička analiza. Cilj se postiže uvođenjem nove međureprezentacije izvornog koda nazvane eCST (obogaćeno konkretno sintaksno stablo, eng. enriched Concrete Syntax Tree). Fokus disertacije je primarno na eCST reprezenataciji koda, reprezentacijama izvedenjim iz eCST i procesu njihovog generisanja, sa opisom oruđa angažovanim u ovim procesima.

Osnovna i najbitnija karakteristika eCST reprezenatacije je nezavisnost od jezika u kom je izvorni kod pisan, što SSQSA okviru daje proširivost na dva nivoa: kroz podršku za nove jezike i kroz podršku za nove analize. Ovo dovodi do efikasnog uvođenja funkcionalnosti na oba navedena nivoa, kao i do kozistentnosti uvedenih funkcionalnosti.

Kao dokaz ispravnosti koncepta, podrška za vizvse od 10 ulaznih jezika je uvedena. Takođe, implementirane su karakteristične tehnike statičke analize (izračunavanje oftverskih metrika, otkrivanje duplikata u kodu, itd.) i integrisane u SSQSA okvir.

Na opisani način, postavljanjem SSQSA okvira, obezbeđena je infrastruktura za dalji razvoj kompletne platforme za kontrolu kvaliteta softvera.

IZ

Datum prihvatanja teme od strane

|  |  |
|---|---|
| NN veća: | 11.06.2014. |
| DP |  |
| Datum odbrane: |  |
| DO |  |
| Članovi komisije: |  |
| (Naučni stepen/ime i prezime/zvanje/fakultet) |  |
| KO |  |
| Predsednik: | dr Vladimir Kurbalija, vanredni profesor, Prirodno-matematički fakultet, Univerzitet u Novom Sadu |
| Mentor: | dr Zoran Budimac, redovni profesor, Prirodno-matematički fakultet, Univerzitet u Novom Sadu |
| Član: | dr Mirjana Ivanović, redovni profesor, Prirodno-matematički fakultet, Univerzitet u Novom Sadu |
| Član: | dr Marjan Heričko, redovni profesor, Fakultet za elektrotehniku, računarstvo i informatiku, Univerzitet u Mariboru |