Two-dimensional Extensibility of SSQSA Framework

JOZEF KOLEK, GORDANA RAKIĆ AND MILOŠ SAVIĆ, University of Novi Sad

The motivation to improve systematic application of software analysis tools by improving characteristics of software analysis tools originates from important aspects of modern software development regarding complexity and heterogeneity; importance of analysis and control during this process; need to keep consistency of the followed results. During the identification of the factors affecting these process we identified two important characteristics of supporting tools: extensibility and adaptability. In this paper describe extensibility of the Set of Software Quality Static Analyzers (SSQSA) in two directions: to support new programming language and to support new analysis algorithm

Categories and Subject Descriptors: D.2.8 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms: Languages, Measurement

Additional Key Words and Phrases: SSQSA, eCST representation, eCSTGenerator

1. INTRODUCTION

Nowadays, large software projects are very complex. They consist of many components, usually very heterogonous ones and developed by usage of many different programming languages and many different programming paradigms. Therefore it is very useful to have some unique set of tools that enables consistent analysis, measurement and control during software development. It is important to support large set of programming languages with different programming paradigms and to provide extensibility and adaptability of the tools. Since almost every single project is unique and every individual or group working on a given project has its own specific needs, it is very important to have flexible set of tools. By flexible we mean that it should be easily extensible and easily adaptive to current needs. One of the main weaknesses of available tools in this field is strong dependency of applicability of software metrics on input programming language [Rakić and Budimac 2011c].

Furthermore, if we consider usage of several language-specific or paradigm-specific tools in development of a single project we meet another difficulty: inconsistency of the gained results. Namely, researches [Novak and Rakić 2011, Lincke et al. 2008] show that different tools ran on the same project may produce different results.

These important aspects of analysis modern software development are motivation to improve systematic application of software analysis tools [Rakić and Budimac 2011c]. by improving characteristics of software analysis tools [Budimac et al. 2012]. A language independent intermediate representation is introduced [Rakić and Budimac 2011a]. It is basis for development of tools to support consistent analysis during software development. These tools have some common characteristics inherited from the joint internal representation. These are language independency, extensibility, and adaptability.

In this paper we will briefly describe extensible Set of Software Quality Static Analyzers (SSQSA) - Section 2. Section 3 provides another side background by describing ANTLR – tool used to make extensibility stronger. In Section 4 we will demonstrate extensibility of the set of the tools on two levels. On lower level we provide process for introducing support of a new programming language which is provided in Section 4.1. On higher level we describe how to introduce new analysis as a new functionality to the set of the tools. This is provided by Section 4.2. Section 5 demonstrates how these processes work on real examples. For these purposes we add support for Delphi (Section 5.1) and calculation of Halstead metrics (Section 5.2.). Related work is provided by Section 6. Finally, conclusion and future work are given by Section 7.

Author's address: J. Kolek, G. Rakić, M. Savić, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia; email: jkolek@gmail.com, {goca, svc}@dmi.uns.ac.rs

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.
In: Z. Budimac (ed.): Proceedings of the 2nd Workshop of Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA), Novi Sad, Serbia, 15.-17.9.2013, published at http://ceur-ws.org

2. SSQSA

Set of Software Quality Static Analyzers (SSQSA) [Budimac et al. 2012] is a set of software tools for static analysis of programs. It is intended to be programming language independent: to support as many languages as possible, and (more importantly) to be easily extensible in this direction. Many different, not only programming, languages are supported at the moment, e.g. Java, C#, Modula-2, WSL, OWL, etc. Program sources of different programming languages are translated into unique intermediate structure named Enriched Concrete Syntax Tree (eCST), and this is what makes it language independent.

The eCST is a new type of syntax tree to be used as intermediate representation of the source code. This intermediate representation is suitable for implementation of many different algorithms for source code analysis. eCST contains unified set of universal nodes to mark different language elements. For example we have universal nodes for marking function calls, variable declarations, operators, etc. Therefore, already existing universal nodes are sufficient to implement a wide range of algorithms for static program analysis [Rakić and Budimac 2011a].

Current architecture (Figure 1.) of the SSQSA is provided by [Rakić et al. 2013] eCST is generated by the central component of the SSQSA framework called eCSTGenerator. So far SSQSA consists of three fully functional tools:

- SMIILE - Software Metrics Independent of Input LanguagE [Rakić and Budimac 2011b],
- SNEIPL - Software Networks Extractor Independent of Programming Language [Savić et al. 2012]
- SSCA - Software Structure Change Analyser [Gerlec et al. 2012]

Other tools are in the development phase. Development of new programming language support and new tools should be straightforward. We describe these procedures in the following sections.

3. ANTLR

ANother Tool for Language Recognition (ANTLR) [Parr 2007] parser generator is the primary tool for adding and maintaining language supports. It takes the grammar specification of the language and produces scanner and parser written in different target languages.

ANTLR[i] is externally produced powerful tool that can process various types of files into syntax trees.

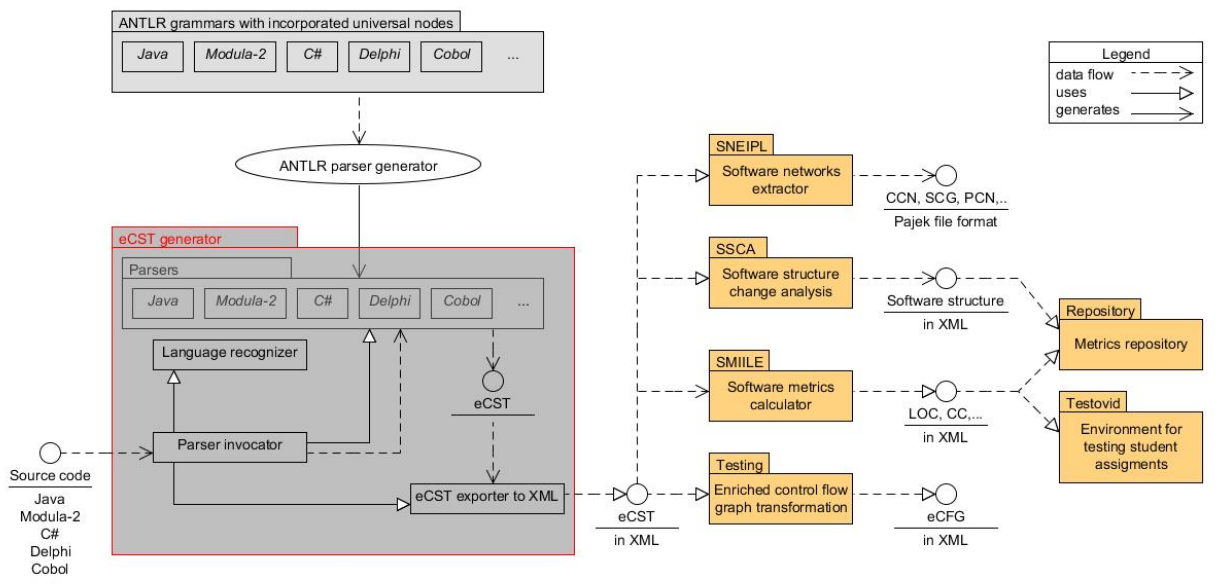

Figure 1. SQSSA architecture

One very important aspect of ANTLR parser generator tool is its ability to rewrite the grammar rules.

[i] ANTLR project site http://www.antlr.org/

This is the way the universal nodes are incorporated into generated syntax trees. This is done on a declarative level without coding in the target language. In the Section 4.1. and its subsections we provide step by step example.

However ANTLR is LL(*) parser generator, the number of look-ahead tokens can vary from rule to rule. Also, ANTLR has the very powerful feature based on backtrack algorithm to match specified rules. But this backtrack feature should be avoided when it is possible because it can be very memory- and time-consuming, and ANTLR grammar becomes hard to debug.

4. ADDING NEW FUNCTIONALITIES

In this Section we will describe general steps necessary to be made to add two new facilities to SSQSA architecture. First one is introducing support for new input language. The second one is new functionality of the tools using eCST as internal representation. This can be applied independently of whether we want to add new analysis to the one of operational tools incorporated in SSQSA or we want to add completely new tool as a component in our framework.

4.1 Introducing Support for New Input Language

To add support for new input language we will primarily need language specification. The best option is to find formal language specification by language grammar. Since ANTLR parser generator is the primary tool for adding and maintaining language supports, the most suitable form of the initial specification of new language is EBNF (Extended Backup-Naur Form) notation[i]. It is much easier to rewrite (rewrite in sense of writing generic ANTLR grammar with respect to the language definition) and modify this kind of specification in ANTLR notation. Counterpart example is language specification in BNF notation, where left recursion is natural feature and repetitions are expressed with recursion. This is kind of grammar that ANTLR notation cannot handle and hence one who introduce new language has to work harder on translation.

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

(1) translate the given language specification to grammar in ANTLR notation (write the ANTLR grammar),
(2) add a rule for syntax tree generation to the grammar (rewrite the rules),
(3) add the universal nodes to the syntax tree (extend the rules),
(4) generate the parser and the scanner
(5) add language to the XML configuration file for supporting languages

We will describe each of these steps in detail.

4.1.1. Write the ANTLR Grammar

Structure of the ANTLR grammar and rule syntax are described on wiki pages on the ANTLR project site. Characteristic of rule syntax is that it is comparable to EBNF notation. Table 1 provides the parallel preview of the main part of the rule syntax in EBNF and in the ANTLR notation.

Table 1. Preview of EBNF and ANTLR notation

EBNF	***ANTLR***	**meaning**
`A	B`	`A	B`	alternative (A or B)
`{A}`	`A*`	zero or more repetition of A
`A{A}`	`A+`	One or more repetition of A
`[A]`	`A?`	One or zero appearance of A

[i] The International standard (ISO 14977) definition of the EBNF
http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip

We provide example of simple rule representing while statement.

Rule for *while* statement in ANTLR notation

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

4.1.2. Rewrite the Rules

When generic ANTLR grammar is completed, then rewriting of rules is the next step. Rule rewriting is a technique of changing the output structure of existing rules. This is the way to create Syntax Trees. This tree is Abstract Syntax Tree (AST) in terms of ANTLR community, but in our case they are actually Concrete Syntax Trees (CST) because we do not omit any elements of the source code and all source code elements are preserved. So, the important issue in our case is that all of the syntax elements must be kept.

We extend simple rule example provided above by adding syntax tree generation

Rule for *while* statement in ANTLR notation with syntax tree generation

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

4.1.3. Extend the Rules

When rewriting of rules is done, the universal nodes can be added. This is actually the conversion from syntax tree to the enriched Concrete Syntax Tree (eCST). Let us demonstrate this step on our simple example.

Rule for *while* statement in ANTLR notation with eCST generation

```
whileStatement : WHILE expression DO statement
                 -> ^( LOOP_STATEMENT
                        ^( KEYWORD WHILE )
                        ^( CONDITION ^( EXPR expression ) )
                        ^( KEYWORD DO )
                        statement);
```

So far catalog of universal nodes consists of more than 30 nodes. For example we have universal nodes to describe function and procedure calls, variable declarations, branch and loop statements, etc. Previous version of catalog is available at [Gerlec et al., 2012]

4.1.4. Generate the Parser and the Scanner

This step is very straightforward by running ANTLR parser generator. It can be done manually via console or it can be done automatically with help of some integrated development environment.

4.1.5. Add a XML Support for the Language

SSQSA eCSTGenerator dynamically recognizes input language in the input file based on the extension of the input file. It calls appropriate scanner and parser and generates eCST. For this purposes we store all needed information about supported languages to the XML file. It contains all information needed to recognize language, call scanner and parser and generate the tree without interaction with the user. XML Schema for storing configuration data about supported input languages is provided by Figure 2.

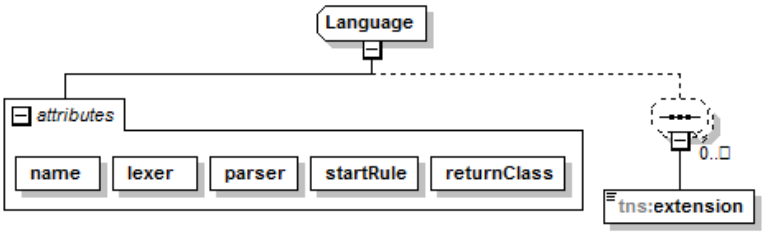

Figure 2. XML Schema for storing configuration data about supported input languages

Finally, eCST representation of the source code is saved in the XML file as well, and it is ready to be used by available tools. For every compilation unit one XML file is created.

4.2 Adding an Analysis

Let us assume that we already have some set of input languages supported by SSQSA environment. Let us consider what steps are needed to implement new functionality on these languages.
To accomplish this task we need to follow this procedure:

(1) define the set of universal nodes needed to implement wanted algorithm
(2) if necessary add the new nodes to existing ANTLR grammar as it is described before (for all languages)
(3) if grammar has been modified generate the scanner and the parser (for all languages)
(4) traverse the eCST, and use the incorporated universal nodes to accomplish the analysis.

4.2.1. Determine the Set of Needed Universal Nodes

The very first step in introducing new functionality is to analyze the algorithm to be implemented. This should lead to determining the set of nodes we need to implement the algorithm. During this analysis we have to think of existing nodes in the catalog. Often existing nodes can be reused and this is very desirable to do so, because we do not want to have two or more different nodes for similar or identical purpose. Our goal is to keep set of universal nodes as minimal as possible and to cover our needs as much as possible. However, if the set of existing nodes does not fit into our requirements, then addition of the new nodes should be considered.

4.2.2. Extend the Existing ANTLR Grammars

Conditionally, in the case when existing universal nodes do not satisfy our needs the new nodes need to be added. This step is already explained in Section 3.1.4. It is very important to do this for all supported languages to keep consistency and completeness. The other important note is to save the nodes previously introduced because we need them for existing implementations.

4.2.3. Generate Scanners and Parsers

Generation of the Scanner and the Parser can be done as explained in 3.1.5. After generation of scanner and parser, they can be used to parse input files and generate eCST also for the already existing functionality because previously used nodes haven-t been modified. However, this step is needed only in case when the grammar file has been modified. In that case we will regenerate scanner and parser for all modified grammars.

4.2.4. Implement the Analysis Algorithm

After an eCST is generated by eCSTGenerator, we can implement an algorithm that traverses the tree, collects the information and does wanted analysis.

5. EXAMPLE

In this Section we will demonstrate described extensibility on the example. We introduce new language (Section 4.1.) and implementation of new software metric calculation (Section 4.2.).

5.1. Delphi

To demonstrate extensibility of SSQSA framework to introduce support for new input language we introduce support for Delphi. Delphi is characteristic in a way that it has elements of both, structural and object-oriented language.

As we mentioned before, first step in adding new language to SSQSA architecture is to find language specification in EBNF notation. As a starting point Delphi 6[i] language specification in EBNF notation has been used. Since Delphi language is derivation of the Object Pascal, earlier versions are very similar to this language, and therefore have LL(1) property in most of the rules, which is suitable to translate to our ANTLR notation. However, translating of given Delphi language specification to our ANTLR notation was not so straightforward, because some parts of given Delphi language specification are very complicated. At some places ANTLR backtrack option was used. During the translating to ANTLR notation Delphi grammar was constantly tested on quite large set of test cases to ensure correctness of the derived grammar.

When generic grammar in ANTLR notation was done, the next step was to rewrite the grammar rules. This step was pretty straightforward. After this step generation of Syntax Tree was supported. In the terms of ANTLR notation it is called Abstract Syntax Tree, but since we keep all of the syntax elements it was closer to Concrete Syntax Tree. After this, grammar was ready for incorporation of the universal nodes. At this step, at some places it was necessary to restructure some of the grammar rules to fit our needs. After universal nodes were incorporated, the next step was to generate the scanner and the parser, and to add needed information to the "Languages.XML" configuration file.

First run [Rakić et al. 2013] of eCSTGenerator on large Delphi project "DelphiProp" consisting of 104438 Lines of Code gave results presented in the Table 2. The number of produced eCST trees (compilation units), the total number of eCST nodes contained in produced trees, running time in seconds and the storage size needed to export eCST trees into the XML files produced by eCST Generator are provided. It can be seen that the transformation of source code into the eCST representation lasted less than a minute where produced eCST trees contains nearly of more than one million nodes.

Furthermore, Table 2 provides the results of running SNEIPL tool [Savić et al. 2012,] which is part of SSQSA environment [Rakić et al. 2013] on this project. This tool generated General Dependency Network (GDN) of the project. We provide the number of GDN nodes, the number of GDN links and the time needed to generate the network. The experiment was performed on AMD Athlon 3200+ processor with 1GB RAM memory.

Table 2. Results of running of eCSTGenertor and back/end SSQSA tools on DelphiProp project

SMIILE	***eCSTGenerator***				***SNEIPL***		
LOC	**#eCST**	**#nodes**	**T[s]**	**S[MB]**	**#GDN nodes**	**#GDN links**	**T[s]**
104438	491	1099961	31	61.7	13721	17745	81

5.2. Halstead Metrics

Halstead metrics express program size and complexity which is evaluated directly from source code. Calculation of Halstead metrics are based on number of occurrences of the operators and the operands. In SSQSA environment the calculation of Halstead metrics can be divided into two phases:

(1) parsing the source code and generation of the corresponding eCST,

(2) calculations of Halstead metric.

First phase takes Delphi source code as input, and generates the eCST, which is then input of the second phase. This is to be done by eCSTGenerator. Second phase, which is actually the core of Halstead metrics

[i] Delphi 6 starting grammar web site http://dgrok.excastle.com/Grammar.html

calculations, traverses this eCST, calculates the Halstead metrics and outputs the results. While traversing the tree, the implemented algorithm must count the total and distinct number of occurrences of the operators and the operands.
The Halstead's operators are: keywords, operators such as "+" and "-", and separators. On the other hand Halstead's operands are: identifiers, 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_TOKEN, DIRECTIVE and CONST universal nodes are used to identify operands. It is important to note the difference between the following universal nodes:

- TYPE marks identifiers representing user-defined data types, and
- TYPE_TOKEN marks primitive, built-in types provided by a programming language.

Universal node NAME is used to collect information about identifiers.

Initial test cases were selected in that way that generated values can be manually verified. Table 3 provides results for one of the largest test cases from this category.

Table 3. Halstead metric values for a test case

Distinct Operators (n1)	*63*	***Program Vocabulary (n)***	*164*	***Program Difficulty (D)***	**105,41584**
Distinct Operands (n2)	101	**Program Length (N)**	1081	**Programming Effort (E)**	838426,3
Total Operators (N1)	743	**Program Volume (V)**	7953,5137	**Programming Time (T)**	46579,24
Total Operands (N2)	338	**Program Level (L)**	0,00948624	**Intelligent Content (I)**	75,448944

6. RELATED WORK

Since the main feature of SSQSA system is its language independency of its internal representation of the source code and therefore extensibility in that direction, we concentrate mainly on this feature in analysis of similar achievements.
Following our overall goal we come to only one related research and development project[i] [Bär 1999]. FAMIX - family of meta-models [Lanza, and Marinescu, 2006] and MOOSE – an extensive platform for software and data analysis [Ducasse et al., 2000] have the most similar general goals to our project. Their strength is mainly in language independency. They support OO design (at the interface level of abstraction) for a wide range of input programming languages. Different input languages are supported by separate tools so-called importers for filling in the meta-model with the information from the source code. This means that it is needed to implement importer tool for each new language which is to be supported. By usage of eCSTGenerator and eCST representation of the source code we enable user just to prepare appropriate grammar to get the full support of needed language. We believe that our approach is more general and more flexible. eCST used to represent the source code covers all aspects of source code and not only the design. It is thus equally appropriate to support broader set of static analysis algorithms. However, it also fully supports procedural languages, including the legacy ones (e.g., COBOL), but also Domain Specific Languages such is OWL and WSL.

We can also discuss situation in the domains that our corresponding back-end tools cover, e.g. software metrics and network extraction.

Similar approach to ours was detected in the ATHENA project [Christodoulakis et al., 1989]. It was tool for assessing the quality of software and the final goal of the tool was to generate a report that describes the quality. ATHENA 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 inbuilt in parser implementation. The generated trees were structured in such a way that the metric algorithms were

[i] FAMOOSE project web site http://scg.unibe.ch/archive/famoos/

easily applied. This is the week point regarding extensibility if we have in mind that for each new language one have to develop new parser with inbuilt metric algorithms in opposite to our approach to generate parsers by parser generator in order to automate process of adding support for new language. Furthermore, eCST is richer representation then AST. Finally, ATHENA was only executable under the UNIX operating system and its official support is not available anymore. SSQSA framework and SMIILE tool, its equivalent to ATHENA (by purpose) is implemented in Java and therefore it can be used on broader range of platforms.

Another tool with similar approach is the CodeSquale[i] metrics. This project was based on a similar idea and the same final goal - language independency. The authors developed a system based on the representation of a source code by AST 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 tool to other programming languages was described. Unfortunately, later results were not published. However, week point of this project was usage of AST for representing the source code. By using eCST we get broader set of algorithms implementable independently of programming language.

Let us look at the field of software networks extraction tools. There is a variety of software networks extractors, but in most cases their usage is restricted to a particular programming language and extract just one type of software network (for example, review of static call graphs extractors for C programming language can be found in [Murphy et al 1998]). It can be concluded that there is no a language independent tool for extraction of software networks covering different levels of abstractions. Furthermore, no tool has possibility to introduce support for new language. If we have in mind limitation concerning range of covered languages and types of supported networks we consider that no available tool for network extraction can meet characteristics that SNEIPL posses and that is extensibility and wide application.

7. CONCLUSION AND FUTURE WORK

In this paper we described SSQSA environment and step by step demonstrated its extendibility. This was also represented by appropriate examples. The approach by usage of eCST applied in SSQSA is very flexible and extensible. Furthermore, eCST representation of the source code is suitable to implement a wide range of algorithms for static code analysis. This means that eCST can be used in some other projects with minor or none modifications. Finally, it can be extended with totally new nodes to satisfy various needs. However, characteristics of SSQSA environment can gain additional value by engaging more automated processes in the whole idea.

Since there are many different grammar notations, both attributed grammars for corresponding parser generators and generic EBNF notations that serves as documentation of some programming languages, it would be useful to have some kind of automatic translators from other grammar notations to ANTLR notation and vise versa. This idea introduces many new questions to the subject, for example how to translate notations with left recursion allowed to notation with LL(*) property (like the ANTLR notation is), e.g. how to deal with actions in attributed grammar notations.

Also, the usage of alternative parser generator can considered to make it easier to find grammar and generate parser. Sometimes it is easy to find grammar for the particular language, but its translation to the ANTLR notation requires hardworking. The previous idea for automated translation between notations of the different parser generators can also be incorporate here.

Moreover, introducing the tool that enables visual creation of the grammar could be more than helpful. If it would be possible to generate parser by only drawing the syntax diagrams or editing the rules in some alternative visual representation this would add important advantage to our approach.

REFERENCES

H Bär. The FAMOOS Object Oriented Reengineering Handbook. Edited by Stéphane Ducasse. Forschungszentrum Informatik an der Univ., 1999.

Zoran Budimac, Gordana Rakić, Marjan Heričko, Črt. Gerlec.. Towards the Better Software Metrics Tool, In Proc of 16th European

[i] CodeSquale project web site http://code.google.com/p/codesquale/

Conference on Software Maintenance and Reengineering (CSMR), Szeged, Hungary, March 27-30, 2012, pp. 491-494.

Zoran Budimac, Gordana Rakić, Miloš Savić, SSQSA architecture, In Proc. Of the Fifth Balkan Conference in Informatics (BCI 2012), Novi Sad, Serbia, September 16-20 2012, ISBN: 978-1-4503-1240-0 doi: 10.1145/2371316.2371380, pp. 287-290

D. Christodoulakis, C. Tsalidis, C. van Gogh, V. Stinesen, Towards an automated tool for software certification. In Proc of Tools for Artificial Intelligence, 1989., IEEE International Workshop on Architectures, Languages and Algorithms. pp. 670–676.

S Ducasse, M Lanza, S. Tichelaar, Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In Proc. of 2nd International Symposium on Constructing Software Engineering Tools (CoSET), Limerick Ireland, 4 – 11 June 2000

Črt Gerlec, Gordana Rakić, Zoran Budimac, Marjan Heričko, 2012. A programming language independent framework for metrics-based software evolution and analysis. ComSIS journal, Vol. 9, No. 3, 1155-1186. (2012).

M. Lanza, and R. Marinescu. Object-oriented metrics in practice: using software metrics to characterize, evaluate, and improve the design of object-oriented systems. Springer, 2006

R Lincke, J Lundberg, W Löwe, Comparing software metrics tools. In Proc. of the international symposium on Software testing and analysis(ISSTA '08). ACM, New York, NY, USA, 131-142. (2008)

Gail C. Murphy, David Notkin, William G. Griswold and Erica S. Lan. 1998 An empirical study of static call graphs extractors. ACM Transactions on Software Engineering and Methodology. 7, 2 (April 1998) 158-191

Jernej Novak, Gordana Rakić, Comparison of Software Metrics Tools for :NET, In Proc. Of 13th International Multiconference Information Society (IS), Collaboration, Software And Services In Information Society (CSS), October 11-15, 2011, Ljubljana, Slovenia, vol. A, pp. 231-234

T. Parr, The Definitive ANTLR Reference - Building Domain-Specific Languages, The Pragmatic Bookshelf, USA, 2007, ISBN: 0-9787392-5-6.

Gordana Rakić, Zoran Budimac, 2011a. Introducing Enriched Concrete Syntax Trees, In Proc. of the 14th International Multiconference on Information Society (IS), Collaboration, Software And Services In Information Society (CSS), (Ljubljana, Slovenia, October 10-14), Volume A, pp. 211-214,

Gordana Rakić, Zoran Budimac, SMIILE Prototype, 2011b. In Proc. Of International Conference of Numerical Analysis and Applied Mathematics ICNAAM2011, Symposium on Computer Languages, Implementations and Tools (SCLIT), (Halkidiki, Greece, September 19-25, 2011) ISBN 978-0-7354-0956-9, pp. 853-856.

Gordana Rakić, Zoran Budimac, 2011c. Problems In Systematic Application Of Software Metrics And Possible Solution, In Proc. of The 5th International Conference on Information Technology (ICIT) (Amman, Jordan, May 11-13, 2011).

Gordana Rakić, Zoran Budimac, Miloš Savić, Language Independent Framework for Static Code Analysis, In Proc. Of the Sixth Balkan Conference in Informatics (BCI 2013), Thessaloniki, Greece, September 19-21 2013, in print

Miloš Savić, Gordana Rakić, Zoran Budimac and Mirjana Ivanović, Extractor of Software Networks from Enriched Concrete Syntax Trees, In Proc. Of International Conference of Numerical Analysis and Applied Mathematics ICNAAM2012, 2nd symposium on Computer Languages, Implementation and Tools (SCLIT), AIP Conference Proceedings, Sep. 2012, vol. 1479, pp. 486–490