First Results of WCET Estimation in SSQSA Framework

NATAŠA SUKUR, NEMANJA MILOŠEVIĆ, SAŠA PEŠIĆ, JOZEF KOLEK, GORDANA RAKIĆ AND ZORAN BUDIMAC, UNIVERSITY OF NOVI SAD

Set of Software Quality Static Analyzers (SSQSA) is a set of software tools for static analysis that has a final goal to enable consistent static analysis, which would improve the overall quality of the software product. The main characteristic of SSQSA is utilization of its unique structure which is an intermediate representation of the source code, called enriched Concrete Syntax Tree (eCST). The eCST enables language independency of analyses implementation in SSQSA, and therefore increases consistency of extracted results by using a single implementation approach for its analyzers. Since SSQSA is also meant to be involved in static timing analysis, one of dedicated tasks is Worst-Case Execution Time (WCET) estimation at code level. The aim of this paper is to describe the progress and the steps taken towards the estimation of the Worst-Case Execution Time in SSQSA framework. The key feature that makes this estimation stand out is its language independence. Although complex to conduct, the eCST intermediate structure is constantly improving in order to provide all the necessary data for successful and precise analyses, making it crucial for complex estimations such as WCET.

Categories and Subject Descriptors**: D.2.8 [Software engineering]**: Metrics - Performance measures

General Terms: Languages, Experimentation, Measurement

Additional Key Words and Phrases: worst case execution time, static analysis, static timing analysis, language independency

1. INTRODUCTION

Many deficiencies of a program can be invisible to the compiler. It optimizes the code and the resulting code could not reflect all the characteristics of the source code. The advantage of analyzing the machine code is, most certainly, the precision. But because we could greatly benefit from systems that could indicate the irregularities in the phase of writing the source code, static analysis has great significance. This is especially the case in very large systems, where the main difficulty would be finding the piece of code that is causing those irregularities. Static analysis helps us detect some commonly made mistakes very easily and therefore reduces time-to-market. However, we cannot completely rely on it. That means we cannot use it instead of testing or guarantee the quality of a program only by applying static analysis.

SSQSA framework gives us a great variety of possibilities for static analysis. Some of the planned functionalities were support for static timing analysis and Worst-Case Execution Time (WCET) estimation at code level [Wilhelm et al. 2008; Lokuciejewski and Marwedel 2009; Lokuciejewski and Marwedel 2011]. Worst-Case Execution Time is estimated as a part of timing analysis and provides the upper (worst) value of the execution time. It is important, since real-time systems must satisfy some time limitations. However, there are some problems regarding the estimation of the Worst-Case Execution Time. Some of them are making estimations on a program that does not have an ending (that contains infinite loops) or making predictions for input values. Estimating WCET would be much easier if the input values were known in advance or if we could say with most certainty which input values would cause the largest WCET results. By estimating WCET on various input values, it is possible to get very different results. That is because paths of a control flow graph are taken based on them and the difference in complexity between those paths could be very significant, which has direct influence on WCET value. Therefore, estimated WCET result is an estimation, which should be a very close approximation, but it must not be below the real WCET result. Despite these difficulties,

This work was partially supported by the Ministry of Education, Science, and Technological Development, Republic of Serbia (MESTD RS) through project no. OI 174023 and by the ICT COST action IC1202: TACLe (Timing Analysis on Code Level), while participation of selected authors in SQAMIA workshop is also supported by the MESTD RS through the dedicated program of support.

Authors addresses: Nataša Sukur, Nemanja Milošević, Saša Pešić, Jozef Kolek, Gordana Rakić, Zoran Budimac
University of Novi Sad, Faculty of Sciences, Department of Mathematics and Informatics, Trg Dositeja Obradovica 4, 21000 Novi Sad, Serbia; email: natasha.sukur@gmail.com, nmilosevnm@gmail.com, pesicsasa@outlook.com, jkolek@gmail.com, gordana.rakic@dmi.uns.ac.rs, zjb@dmi.uns.ac.rs

Copyright © by the paper's authors. Copying permitted only for private and academic purposes.
In: Z. Budimac, Z. Horváth, T. Kozsik (eds.): Proceedings of the SQAMIA 2016: 5th Workshop of Software Quality, Analysis, Monitoring, Improvement, andApplications, Budapest, Hungary, 29.-31.08.2016. Also published online by CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073)

WCET estimation on source code level is also significant because it sometimes allows us to perform this estimation in the earliest phases of software development, accomplished e.g. by performing timing model identification using a set of programs and WCET estimate calculation using flow analysis and the derived timing model. That is important because if the final executable violates the timing bounds assumed in earlier stages, the result could be a highly costly system re-design [Gustafsson et al. 2009].

This work is mostly oriented towards language independence, which makes it uncommon when compared to other approaches in solving this problem. The expected outcome of this particular work was planned to be reflected in as successful estimation as possible, no matter the input language. This kind of analysis goes quite deep into detail and in order to perform the estimation in a thorough and proper manner, evolution of parts of the SSQSA framework is considered inevitable.

The rest of the work deals with the selection of approach and its implementation. Related work is provided in section 2. Section 3 deals with undertaken work for WCET estimation on control flow graphs. The obtained results are given in section 4. Limitations that were met are within section 5 and the conclusion is described in the last section.

2. RELATED WORK

Worst-Case Execution Time estimation is an ongoing problem and many different approaches have been taken in solving it. Here are some notable solutions.

aiT[1] WCET Analyzers [Lokuciejewski and Marwedel 2011] offer a solution which determines WCET in tight bounds by using binary executable and reconstructs a control-flow graph. WCET determination is separated into several consecutive phases - value analysis, cache analysis, pipeline analysis, path analysis and analysis of loops and recursive procedures. The difference is in the approaches is using eCST over binary code before conversion to control-flow graph. Additionally, the analysis aiT performs on ranges of values, access to main memory and some other features are far more complex than what we have accomplished. aiT WCET analyzer is a commercial tool already used by some enterprises, mostly for embedded systems testing.

Bound-T[2] is a tool that also uses machine code in its static analysis approach. Similarly, it uses control-flow graphs and call trees for WCET estimation. Therefore, it is also language-independent and does not require any special annotations in the source code. It also has an option to take into consideration user defined assertions, for example, upper loop count. Bound-T is cross-platform and can analyze code that targets multiple processors. It is a free, open-source tool. The main difference between Bound-T and SSQSA WCET analyzer prototype is that Bound-T uses compiled machine code for analysis, while SSQSA works with program source code.

SWEET[3] (SWEdish Execution Time tool) is a research prototype tool with the flow analysis as the main function. The result obtained by flow analysis is information used further for WCET estimation. The analysis is performed on programs in ALF (ARTIST2 Language for WCET Flow Analysis) internal intermediate program language format. Different sources, such as C code or assembler code, could be transformed into ALF code, leading to generation of flow facts, further used in low level analysis or in WCET estimation. Using ALF was also an approach that was taken into consideration for implementation of WCET estimation in SSQSA, but was left because the estimation was already thought to be possible by converting the eCST structure.

Heptane[4] static WCET estimation tool is a stable research prototype written in C++. However, it runs only on Linux and Mac OS X and supports MIPS and ARM v7[5] architectures. SSQSA is written

[1] http://www.absint.com/ait/
[2] http://www.bound-t.com/
[3] http://www.mrtc.mdh.se/projects/wcet/sweet/
[4] https://team.inria.fr/alf/software/heptane/
[5] MIPS is a processor architecture which is used for the purpose of digital home, home networking devices, mobile applications and communication products, while ARM (Advanced RICS Machine) is an entire family of architectures for computer

in Java and therefore it is possible to run its code anywhere. Heptane analyzes source code written in C and is not able to deal with pointers, *malloc* or recursion. SSQSA is language independent and is, in theory, able to deal with recursion. Heptane transforms the binary code into a control-flow graph in XML format. Afterwards, it is able to do low-level and high-level analysis, such as WCET.

Chronos[6] is an open source software that deals with static analysis and WCET estimation. It includes both the data input and the hardware into WCET estimation. After gathering program path information (derived from source program or the compiled code) and instruction timing information, it performs WCET estimation. The difference from our approach is generating the control-flow graph from the binary code written only in C. On the contrary, in SSQSA control-flow graph is derived from eCST, created from the source code of the supported languages without compilation. The underlying hardware, however, is not considered in SSQSA WCET estimation.

3. WCET ESTIMATION IN SSQSA

For the needs of WCET estimation, two approaches were taken into consideration. The first one was using the existing universal intermediate structure, eCST (on which SSQSA framework is based) [Rakić 2015]. Another one was using ALF as a domain specific language [Gustafsson et al. 2009; Rakić and Budimac 2014]. ALF can be used as an intermediate language and several tools for converting popular programing languages code to ALF code already exist. ALF is specifically designed for flow analysis and WCET. It is also designed in such a way that it is very easy to use it for all kinds of code analysis.

Many languages (Java, C#, Modula-2, Delphi, ...) were already supported by SSQSA and they can be transformed into eCST structure. Since the WCET estimation without involving ALF appeared possible, the first approach was selected. Also, this meant the validation could be more precise by implementing WCET on eCST, since the sample of supported languages was bigger and there were only a few translators for ALF.

3.1 Conversion from eCST to eCFG

The work was done on a structure called eCST and its nodes. A node in eCST contains important pieces of information, stored in a token. It holds information on the node type, which is of high importance in conversion from eCST to eCFG (enriched Control Flow Graph), on which the WCET estimation would be later done. The approach was the following: using the Depth-First Search algorithm, eCST was traversed and its nodes were used for generating eCFG. It is important to mention that not all of the nodes are necessary for the conversion to eCFG. Only a subset of all the universal nodes have been selected as of importance to WCET estimation. Those included in the resulting eCFG are the following:

- COMPILATION_UNIT,
- FUNCTION_DECL,
- FUNCTION_CALL,
- VAR_DECL,
- STATEMENT,
- ASSIGNMENT_STATEMENT,
- LOOP_STATEMENT,
- BRANCH_STATEMENT,
- BRANCH,
- CONDITION.

---

processors, named RICS after reduced instruction set computing. ARMv7 is used mostly for high-end mobile devices, such as smartphones, laptops, tablets, Internet of Things devices etc.

[6] http://www.comp.nus.edu.sg/~rpembed/chronos/

An eCFG node stored the following information on the type of the node, optional additional information, whether a node is initial/final and its child nodes. Due to the need for later calculation of condition values in specific nodes (loop and branch), a simple symbol map was also implemented.

3.2 WCET analysis on eCFG

For the need of estimating the Worst-Case Execution Time, a wrapper around a simple node was made. It stored some additional information on the node: bestTime, worstTime and averageTime. Since the eCFG structure was simple for iteration, each of the nodes was easily analyzed. Usage of a symbol map made it possible to restore values necessary for the calculation of various conditions in loop and branch nodes.

The calculation is performed the following way: each of the eCFG nodes has a WCET value, which is added to the total value. However, WCET estimation on loop and branch statements is somewhat more complex. The value of nodes contained within the loop statement body is being summed and multiplied by the count of loop statement executions. Therefore, it is also necessary to perform evaluation of loop statement condition expression in order to estimate the number of executions. On the other hand, the branch statement's WCET value is chosen by estimating separate WCET values of each branch. Upon separate estimations, the largest WCET value is considered the worst and assigned as branch statement's WCET value.

For regular nodes (non-composite ones, all except branch and loop nodes), the worst and best times are equal to 1. This means that one line of code equals to one time unit. For more complicated (composite) nodes, the value depends on the related nodes. For branch nodes, the worst time is the worst time of the "worst" branch, while the best time is the best time of the "best" branch. For loop nodes, the worst case value is calculated by evaluating the conditional expression, which then becomes multiplied with the worst execution time of the inner-loop nodes. The best time of a loop node is always one, based on the fact that if the condition cannot be evaluated or if it is false, the loop body will not be executed. In all cases, the average time is calculated by dividing the sum of worst and best time by two. So, average time is the average value of best and worst case execution time.

3.3 Analyzing function declarations and function calls

Specification of function declarations and function calls which would enable establishing connections among numerous functions was necessary. The missing node that was a part of eCST, but not yet transformed and added into eCFG was the *function call node*. The problem with implementing this node was that many modern languages allow method (function) overriding, which means that the function name is not enough to identify which function is going to be called. To make this prototype work, a simple solution in form of a structure which holds all the function declarations and their respective parameter numbers was implemented. For this prototype, only the number of the arguments was considered and not their type, but with remark that this is needed for a more precise evaluation in the future. Afterwards, a function call hierarchy was produced and all the function calls were taken into consideration, which meant that the prototype was no longer bound to analyzing single methods in a compilation unit.

By adding function calls to eCFG, it was possible to create an interprocedural control flow graph (ICFG) [Lokuciejewski and Marwedel 2011] . What makes an ICFG different from a regular control flow graph is that it deals with control flow of an entire program, rather than a single function. A function call node is connected to the entry node of the function that is invoked and after reaching the function exit node, the ICFG returns to the node following the original function call node.

4. RESULTS

XML is the default form of exchanging information among different components of SSQSA. It is generated by performing a recursive walk through eCFG. The XML that is generated from eCFG slightly differs from the one generated as a result of WCET estimation. The source code, its control flow graph and resulting XML containing WCET-related data are shown on an example that performs

“divide and conquer” problem solving approach: a problem is divided into simpler sub-problems until a certain threshold is reached and the sub-problems are simple enough to be solved.
On the left side of Figure 1, the Divide and Conquer source code can be seen. The graph on the right side represents nodes of the generated Control Flow Graph. This figure has been made in the way that it is simple to compare a line of code on the left hand side and, at the same level, the produced nodes of the Control Flow Graph for that specific line of code. The structure of this Control Flow Graph is actually the XML file given below.

In this XML file, lines with more complex structures, like BRANCH_STATEMENT and LOOP_STATEMENT, have been bolded in order to emphasize the way the inner nodes affect these statements’ worst case values.

In our example, the goal of the program is to set the value of a variable to zero, but only if its value is smaller than a certain threshold. If the value is greater than the threshold, the program will “divide” the variable by performing subtraction by one. The following eCFG shows only the most important, `divideAndConquer` method.

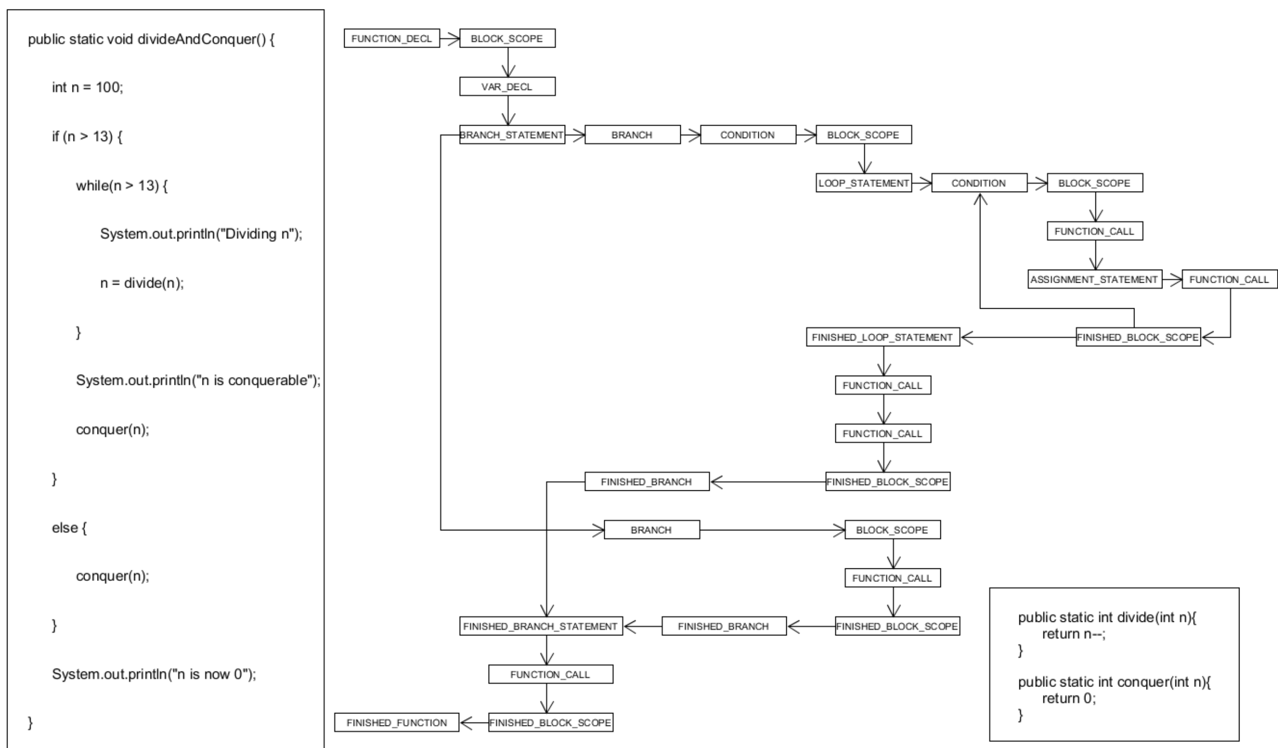

Figure 1 – Source code and its Control Flow Graph representation


```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<CFG-WCET WCET="359">
<COMPILATION_UNIT average="1.0" best="1" children="2" id="1" info="" worst="1"/>
<FUNCTION_DECL average="1.0" best="1" children="3" id="2" info="divideAndConquer" worst="1"/>
<BLOCK_SCOPE average="0.0" best="0" children="4" id="3" info="" worst="0"/>
<VAR_DECL average="1.0" best="1" children="5" id="4" info="int, n, 100" worst="1"/>
<BRANCH_STATEMENT average="178.0" best="1" children="6,21" exit="26" id="5" info="n,GREATER,13"
worst="355"/>
<BRANCH average="355.0" best="355" children="7" id="6" info="" worst="355"/>
<CONDITION average="1.0" best="1" children="8" id="7" info="n,GREATER,13" worst="1"/>
<BLOCK_SCOPE average="0.0" best="0" children="9" id="8" info="" worst="0"/>
<LOOP_STATEMENT average="349.0" best="349" children="10" exit="16" id="9" info="n,GREATER,13" worst="349"/>
<CONDITION average="1.0" best="1" children="11" id="10" info="n,GREATER,13" worst="1"/>
<BLOCK_SCOPE average="0.0" best="0" children="12" id="11" info="" worst="0"/>
<FUNCTION_CALL average="1.0" best="1" children="13" id="12" info="System.out.println" worst="1"/>
<ASSIGNMENT_STATEMENT average="1.0" best="1" children="14" id="13" info="n, FUNCTION_CALL" worst="1"/>
```

```
<FUNCTION_CALL average="1.0" best="1" children="15" id="14" info="divide" worst="1"/>
<FINISHED_BLOCK_SCOPE average="0.0" best="0" children="16,10" id="15" info="" worst="0"/>
<FINISHED_LOOP_STATEMENT average="0.0" best="0" children="17" id="16" info="" worst="0"/>
<FUNCTION_CALL average="1.0" best="1" children="18" id="17" info="System.out.println" worst="1"/>
<FUNCTION_CALL average="1.0" best="1" children="19" id="18" info="conquer" worst="1"/>
<FINISHED_BLOCK_SCOPE average="0.0" best="0" children="20" id="19" info="" worst="0"/>
<FINISHED_BRANCH average="0.0" best="0" children="26" id="20" info="" worst="0"/>
<BRANCH average="2.0" best="2" children="22" id="21" info="" worst="2"/>
<BLOCK_SCOPE average="0.0" best="0" children="23" id="22" info="" worst="0"/>
<FUNCTION_CALL average="1.0" best="1" children="24" id="23" info="conquer" worst="1"/>
<FINISHED_BLOCK_SCOPE average="0.0" best="0" children="25" id="24" info="" worst="0"/>
<FINISHED_BRANCH average="0.0" best="0" children="26" id="25" info="" worst="0"/>
<FINISHED_BRANCH_STATEMENT average="0.0" best="0" children="27" id="26" info="" worst="0"/>
<FUNCTION_CALL average="1.0" best="1" children="28" id="27" info="System.out.println" worst="1"/>
<FINISHED_BLOCK_SCOPE average="0.0" best="0" children="29" id="28" info="" worst="0"/>
<FINISHED_FUNCTION average="0.0" best="0" children="30" id="29" info="" worst="0"/>
<FINISHED_COMP_UNIT average="0.0" best="0" id="30" info="" worst="0"/>
</CFG-WCET>
```


The developed prototype is able to give correct WCET estimations only for a certain subset of possible examples, which is explained further in the next section. Working on the prototype however was hard, because of limited amount of information which was provided to the tool because of the conversion of source code to eCST. It is clear that WCET analysis would be much more precise when the estimations are performed on machine code, like some of the already mentioned projects. Nevertheless, it is possible to analyze and estimate WCET even on the source code level which can be of great benefit.

5. LIMITATIONS

Although some estimations are successfully performed and some progress was made, this project is only a prototype that does not cover all the given test examples. For now, it only works on simple pieces of code. There are some major points for improving in continuing the research on this project:

- Condition evaluation in WCET analyzer

Currently, while performing WCET estimations, only simple conditions are successfully evaluated, such as i < 5 or i < j. There are certain complications regarding the complex conditions, for example

i < j + k || i < func(a, b).

- Evaluation of complex expressions when initializing or assigning value to a variable

This prototype currently works only with a simple statements when generating eCFG from assignment statement or declaration of variables, such as int i = 5. The conversion works also for an assignment statement of this kind: int j = i, but not for more complicated, such as int k = j + i. This assignment statement cannot be evaluated because evaluation of arithmetic statements is not implemented yet.

- Determination of function call target should be more precise

As already mentioned, currently only the number of function parameters is taken into consideration instead of checking their types, names etc. This should be improved by involving the parameter types into deducing the paths of the control-flow graph more precisely which can be done by reuse of Static Call Graph generation implemented in eGDNGenerator component of SSQSA [Rakić 2015].

6. CONCLUSION

The undertaken work is only a path towards a language independent WCET estimation. However, it is only the first phase towards its more precise estimation. The second is introducing the platform variable to the estimation, which means domain specific universal nodes could also become a part of the eCST structure. Upon success in two mentioned phases, the result validation is to be done by making comparison to the result generated by the SWEET tool. Also, an interesting proposal is to see

if a model similar to Statecharts (enriched Statechart) could be included in SSQSA framework and used for WCET analysis. The Statechart implementation has already begun but it is in the early stage of development.

The motivation behind this work is involving the static timing analysis in SSQSA framework. Upon finishing this first phase of research and implementation, it is clear that working further in this direction could lead us to some meaningful and accurate results, in the sense of WCET estimation working successfully on complex programs. The problems that were met are mostly related to solving some implementation issues explained in section Limitations in more detail. Upon their resolving, a highly functional language independent WCET estimation on source code level could be performed. Therefore, the future work towards introducing the platform variable could be undertaken.

REFERENCES

J. Gustafsson, A. Ermedahl, B. Lisper, C. Sandberg, L. Källberg. 2009. ALF–a language for WCET flow analysis. In Proc. 9th International Workshop on Worst-Case Execution Time Analysis (WCET'2009), Dublin, Ireland, pp. 1-11

J. Gustafsson, P. Altenbernd, A. Ermedahl, B. Lisper. 2009. Approximate Worst-Case Execution Time Analysis for Early Stage Embedded Systems Development. Proc. of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009). Lecture Notes in Computer Science (LNCS), Springer, pp. 308-319

P. Lokuciejewski, P. Marwedel. 2009. Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. In Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems (ECRTS '09). IEEE Computer Society, Washington, DC, USA, pp. 35-44

P. Lokuciejewski, P. Marwedel, P. 2011. Worst-case execution time aware compilation techniques for real-time systems. Springer.

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

G. Rakić, Z. Budimac. 2013. Language independent framework for static code analysis, In Proceedings of the 6th Balkan Conference in Informatics (BCI '13). Thessaloniki, Greece, ACM, New York, NY, USA, pp. 236-243

G. Rakić, Z. Budimac. 2014. Toward Language Independent Worst-Case Execution Time Calculation, Third Workshop on Software Quality, Analysis, Monitoring, Improvement and Applications (SQAMIA 2014). Lovran, Croatia, pp. 75-80

G. Rakić. 2015. Extendable and Adaptable Framework for Input Language Independent Static Analysis, Novi Sad Faculty of Sciences, University of Novi Sad, doctoral dissertation, p. 242