Validation of Static Program Analysis Tools by Self-Application: A Case Study

MILOŠ SAVIĆ AND MIRJANA IVANOVIĆ, University of Novi Sad

The validation of static program analysis tools is an extremely hard and time consuming process since those tools process source code of computer programs that are usually extremely large and complex. In this paper we argue that static program analysis tools can be validated by self-application, i.e. by applying a source code analysis tool to its own source code. Namely, developers of a complex source code analysis tool are familiar with its source code and consequently in position to more quickly examine whether obtained results are correct. The idea is demonstrated by the application of SNEIPL, a language-independent extractor of dependencies between source code entities, to itself.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Validation*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms: Experimentation,Measurement

Additional Key Words and Phrases: Source code analysis, self-application, validation

1. INTRODUCTION

Static program analysis tools process source code of computer programs in order to extract information that can help software engineers in a variety of tasks ranging from program understanding to fault detection [Binkley 2007]. The automated extraction of information in static program analysis is done without executing program and relies only on source code or some intermediate representation. Software validation refers to the process of evaluation of a software system in order to check whether it works properly and according to its specification. The validation of static program analysis tools is an extremely important task since those tools are used to understand and improve software systems. On the other hand, real-world software systems are usually extremely large and hard to comprehend making the validation hard and time consuming.

The identification of dependencies between source code entities (functions, classes, modules, etc.) is one of fundamental problems in static program analysis. We use the generic term "software network" to denote directed graphs of dependencies between source code entities. The importance of software networks extraction spans multiple fields such as empirical analysis of complexity of software systems, their reverse engineering and computation of software design metrics [Savić et al. 2014]. In our previous works [Savić et al. 2012b; 2014] we introduced SNEIPL – a language-independent approach

This work was supported by the Serbian Ministry of Education, Science and Technological Development through project *Intelligent Techniques and Their Integration into Wide-Spectrum Decision Support*, no. OI174023. The authors would like to thank professor Zoran Budimac for valuable comments on an early version of this paper.
Author's address: M. Savić, M. Ivanović, Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia, email: {svc, mira}@dmi.uns.ac.rs.

In: Z. Budimac, M. Heričko (eds.): Proceedings of the 4th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications (SQAMIA 2015), Maribor, Slovenia, 8.-10.6.2015. Also published online by CEUR Workshop Proceedings (CEUR-WS.org, ISSN 1613-0073)

to the extraction of software networks representing software systems at various levels of granularity[1]. In the experiment described in [Savić et al. 2012b] we showed that SNEIPL extracts isomorphic networks from simple, but structurally and semantically equivalent software systems written in different programming languages. In the subsequent research [Savić et al. 2014] we demonstrated that SNEIPL is able to extract software networks from real-world software systems written in different programming languages (Java, Modula-2 and Delphi). Moreover, we showed that class collaboration networks extracted from 10 real-world Java software systems are highly similar to those obtained by a language-dependent tool which forms networks from byte code, and much precise than the networks obtained by a language-independent tool which employs a lightweight, fuzzy parsing approach to dependency extraction. The research presented in this paper follows previously conducted experiments related to the validation of SNEIPL. More specifically, we used SNEIPL to extract software networks from its own source code. Due to the familiarity with the implementation of SNEIPL we were in position to quickly determine whether obtained results are correct.

The rest of the paper is structured as follows. Related work is presented in Section 2. The short description of the tool is given in Section 3. Experiments and results are presented in Section 4. The last Section concludes the paper.

2. RELATED WORK

There is a large number of tools that identify dependencies between source code entities. Usually they are language-dependent (tied to a particular programming language). For example, the overview of existing static call graphs extractors for C/C++ can be found in [Murphy et al. 1998; Telea et al. 2009]. Software networks can also be extracted from software documentation such as JavaDoc HTML pages (up to a certain level of precision) or low-level intermediate representations such as Java byte code. Software networks extractors rely either on traditional parsing techniques or employ more lightweight, but less precise, approaches based on pattern matching [Kienle and Müller 2010].

In research works that deal with the analysis of software systems under the framework of complex network theory software networks are usually extracted using language-specific tools:

—in [Valverde and Solé 2007; de Moura et al. 2003] analyzed networks were extracted by lightweight, handmade parsers of C/C++ header files,

—in [Jenkins and Kirk 2007; Louridas et al. 2008] by tools that rely on Java bytecode parsers,

—in [Wang et al. 2013] by parsing C source code using a modified version of the GCC compiler,

—in [Taube-Schock et al. 2011] by extending the standard Java parser of the Eclipse IDE,

—in [Wheeldon and Counsell 2003] by using Java Doclet capabilities to inspect the source code structure,

—in [Puppin and Silvestri 2006] by parsing JavaDoc HTML pages,

—in [Savić et al. 2011; Savić et al. 2012a] by a tool that relies on Java parser generated by JavaCC.

The identification of dependencies among source code entities in existing language-independent reverse engineering tools can be classified into two categories:

—Tools that have separate fact extractors (source code models in terms of software networks) for each supported language. Examples of such tools are Rigi [Kienle and Müller 2010], GUPRO [Ebert et al. 2002] and Moose [Ducasse et al. 2000].

[1]The tool can be downloaded at https://code.google.com/p/ssqsa/

—Tools that realize partially language-independent fact extraction. This means that for a subset of supported languages software networks are formed from a low-level (statement-level) language-independent source code representation, while for other supported languages there are separate fact extractors. An example of a tool that belongs to this category is Bauhaus [Raza et al. 2006].

The validation of dependency extraction in aforementioned tools was conducted on several real-world computer programs, none of them being the reverse engineering tool itself.

3. SNEIPL TOOL

SNEIPL has been implemented as one of the back-ends of the SSQSA framework [Rakić et al. 2013; Budimac et al. 2012]. The whole SSQSA framework is organized around the enriched Concrete Syntax Tree (eCST) representation of source code [Rakić and Budimac 2011a; 2011b] that is produced by the SSQSA front-end known as eCSTGenerator. The eCST representation is a language-independent source code representation and makes SSQSA back-ends independent of programming language. The concept of universal nodes introduced in the eCST representation is what makes it substantially different from other tree representations of source code. Namely, eCST universal nodes are predefined language-independent markers of semantic concepts expressed by concrete language constructs. One universal node in an eCST denotes particular semantic concept realized by the syntax construction embedded into the eCST sub-tree rooted at the universal node.

From an input set of eCSTs SNEIPL forms a heterogeneous software network that is known as *General Dependency Network* (GDN). GDN shows dependencies among software entities reflecting the design structure of a software system [Savić et al. 2014]. Nodes in a GDN represent architectural elements of a software system: packages, classes/modules, interfaces, functions/methods and global variables/class attributes. GDN links represent various types of relations: CALLS relations between functions, REFERENCES relations between package-level entities, REFERENCES relations between class-level entities, relations that represent different forms of class coupling, USES relations between functions and variables, and CONTAINS relations that reflect the hierarchy of entities.

The set of eCST universal nodes, among others, contains entity-level universal nodes which mark definitions/declarations of software entities. SNEIPL deduces vertical dependencies (CONTAINS relations) from the hierarchy of entity-level eCST universal nodes in input eCSTs. Calls relations between functions are recognized by analysis of sub-trees rooted at the FUNCTION_CALL universal node which marks function calls. Relations among class-level entities are identified by analysis of sub-trees rooted at the TYPE universal node which marks type identifiers. Finally, relations between functions and global variables are identified by analysis of sub-trees rooted at the NAME universal node which marks all identifiers present in the source code. To match an identifier (name of variable, type or invoked function) with its definition SNEIPL uses the name resolution algorithm that is based on several components: previously identified vertical dependencies, information contained in import statements (statements that are marked with the IMPORT_DECL universal node), information contained in local symbol tables that are attached to FUNCTION_DECL (marks function definitions) and BLOCK_SCOPE (marks block of statements) universal nodes, lexical scoping rules and rapid type analysis [Bacon and Sweeney 1996] that is adopted for the eCST representation. The more detailed description of the name resolution algorithm is given in [Savić et al. 2014].

4. EXPERIMENT AND RESULTS

SNEIPL has been written in the Java programming language. The implementation of SNEIPL consists of 6876 lines of code (without empty lines) which means that SNEIPL is a non-trivial software system of moderate size. Using eCSTGenerator we transformed the source code into the eCST representation.

The eCST representation of SNEIPL consists of 54 eCSTs (one eCST correspond to one compilation unit). Then we employed SNEIPL to extract software networks from the SNEIPL implementation. Obtained GDN has 584 nodes and 2285 links.

4.1 Recovery of SNEIPL's architecture

Package collaboration networks (PCN) show dependencies among packages and thus represent the architecture of software systems at the highest level of abstraction. Figure 1 shows the SNEIPL PCN extracted using SNEIPL. Due to the familiarity with the implementation of SNEIPL we were in position to extremely quickly validate that the extracted PCN is actually correct.

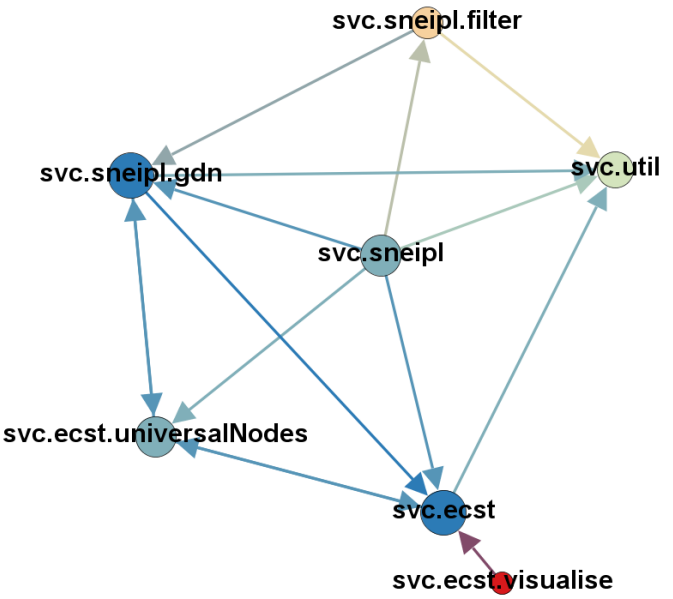

Fig. 1. The package collaboration network of SNEIPL.

The package **svc.sneipl** encompasses core SNEIPL's classes – classes which identify software entities and dependencies between them relying on input eCSTs. This package depends on all other SNEIPL packages except the **svc.ecst.visualise** package. **svc.ecst.visualise** contains source code of a simple and standalone GUI application that visualizes eCST trees. We mainly used it during the development and testing of SNEIPL when it was necessary to have input XML files visually observable. In contrast to **svc.sneipl**, **svc.util** groups simple utility classes. Therefore, this package does not depend on any other package.

The package **svc.ecst** contains classes which provide functionalities related to the eCST representation. The eCST representation of one compilation unit is internally stored as an object of **svc.ecst.ECSTTree** class. This class defines methods which load eCST from an input XML file produced by eCSTGenerator and holds the reference to the root node. Each node in loaded eCST is represented as an object of **svc.ecst.ECSTNode** class which holds the content of the node, the reference to the parent node and the list of references to child nodes. The package **svc.ecst.universalNodes** groups classes that represent different eCST universal nodes used by SNEIPL. One eCST universal node is represented by a class that directly or indirectly extends **svc.ecst.ECSTNode** class. Class

svc.ecst.ECSTTree makes objects representing universal nodes and consequently **svc.ecst** depends on **svc.ecst.universalNodes**. The dependency between **svc.ecst** and **svc.ecst.universalNodes** is reciprocal because universal nodes are instances of **svc.ecst.ECSTNode**.

SNEIPL recognizes software entities and dependencies among them according to the language-independent procedure that relies only on eCST universal nodes. Thus, the core package depends on **svc.ecst.UniversalNodes**. When an entity/dependency is recognized the appropriate node/link in the GDN is created. Thus, the core package depends on **svc.sneipl.gdn**. Each GDN node corresponds to one eCST universal node marking the definition/declaration of an software entity and consequently **svc.sneipl.gdn** depends on **svc.ecst** and **svc.ecst.universalNodes**. The dependency between **svc.sneipl.gdn** and **svc.ecst.universalNodes** is reciprocal since local symbol tables can be attached to some universal nodes. Local symbol tables contain local variables defined in functions and block statements. Those variables can have types that correspond to GDN nodes and consequently **svc.ecst.universalNodes** depends on **svc.sneipl.gdn**.

Finally, the main SNEIPL class contained in the core package instantiates filters from **svc.sneipl.-filter** to isolate specific software networks from the formed GDN. Thus, the core package depends on the filter package, while the filter package depends on the gdn package.

4.2 Isolated entitites

Isolated nodes in extracted software networks can indicate missing links, and thus can point to errors in the implementation of software networks extraction tool. Therefore, we determined and examined characteristics of isolated nodes in the networks representing SNEIPL. The package and class collaboration network of SNEIPL do not contain isolated nodes (unused packages and classes). On the other hand, isolated nodes can be observed in the SNEIPL static call graph (SCG) and FUGV (Function Uses Gloval Variable) network.

The SNEIPL SCG contains 34 isolated nodes (10.36% of the total number). Table I shows the list of isolated nodes. To each method SNEIPL assigns a name that can be described by the following regular expression:

$$F\text{'.'}\ M\ (\text{'@'}\ T)^*\ \text{'\#'}\ R,$$

where F denotes the fully qualified name of a class/interface which declares/defines the method, m is the name of the method, T denotes the type of a formal argument of m, while R is the return type of m. Seven methods listed in the table are methods that are unused – methods that are never called by other methods, nor they call other methods defined in the SNEIPL source code. Those methods can be safely removed from the SNEIPL source code distribution. Three isolated nodes represent method declarations from *SymTab* – the only interface defined in SNEIPL. The *SymTab* interface is implemented by classes representing eCST universal nodes to which local symbol tables can be attached. Those three nodes are isolated simply because in-coming links are given to the nodes representing implementations of those declarations in classes that implement the *SymTab* interface. Three isolated methods listed in Table I are so called call-back methods – methods defined in the SNEIPL source code that are called only from methods contained in external frameworks. Namely, SNEIPL defines four transform methods that are called by the JUNG library to export extracted software networks in the Pajek network file format. One of those methods calls one method defined in SNEIPL, and consequently it is represented by a non-isolated node in the SNEIPL SCG. Other three transform methods do not rely on SNEIPL methods and consequently they are isolated nodes. One method listed in the table is GUI method – method that is activated when a button is clicked in the GUI application that visualizes eCST trees.

Table I. Isolated nodes in the SNEIPL static call graph.

Method name	Explanation
svc.ecst.ECSTNode.disposeChilds#void	Unused
svc.ecst.ECSTNode.emptySubtree#boolean	Unused
svc.ecst.ECSTNode.findAllAtFirstLevel@String#LinkedList	Unused
svc.ecst.ECSTNode.findFirstAtFirstLevel@String#ECSTNode	Unused
svc.ecst.ECSTNode.rewriteToken@String#void	Unused
svc.ecst.ECSTTypedNode.ECSTTypedNode@String@ECSTNode@boolean@boolean#void	Called only via Reflection API
svc.ecst.universalNodes.Argument.Argument@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.ArgumentList.ArgumentList@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.AttributeDecl.AttributeDecl@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.BlockScope.BlockScope@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.ConcreteUnitDecl.ConcreteUnitDecl@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.Extends.Extends@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.FormalParamList.FormalParamList@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.FunctionCall.FunctionCall@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.FunctionDecl.FunctionDecl@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.Implements.Implements@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.ImportDecl.ImportDecl@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.Instantiates.Instantiates@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.InterfaceUnitDecl.InterfaceUnitDecl@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.Name.Name@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.PackageDecl.PackageDecl@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.ParameterDecl.ParameterDecl@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.SymTab.getType@String#GDNNode	Interface declaration
svc.ecst.universalNodes.SymTab.getTypeAsStr@String#String	Interface declaration
svc.ecst.universalNodes.SymTab.nameDeclaredHere@String#boolean	Interface declaration
svc.ecst.universalNodes.Type.Type@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.TypeDecl.TypeDecl@String@ECSTNode#void	Called only via Reflection API
svc.ecst.universalNodes.VarDecl.VarDecl@String@ECSTNode#void	Called only via Reflection API
svc.ecst.visualise.Visualiser.browseBtnClicked#void	GUI method
svc.sneipl.filter.SoftNet.LinkTransformer.transform@SNLink#Number	Call back method
svc.sneipl.filter.SoftNet.NodeTransformer.transform@SNNode#String	Call back method
svc.sneipl.filter.SoftNet.SNNode.getType#GDNNodeType	Unused
svc.sneipl.FuncCallResolver.dumpCandidates@LinkedList#void	Unused
svc.sneipl.gdn.GDN.LinkTransformer.transform@GDNLink#Number	Call back method

As it can be observed the majority of isolated nodes given in Table I are actually constructors defined in classes that represent different eCST universal nodes. The main characteristic of those classes is that they only define a constructor which invokes the constructor of the super class. At the moment SNEIPL resolves only explicit function calls (calls for which the name of invoked is explicitly stated), while indirect function calls (e.g., through function pointers, variables of procedural data types or via language-specific keywords such as *super* and *this* in the case of Java) are not yet supported. Objects representing universal nodes in eCSTs are instantiated using configurable factory pattern and the Java Reflection API:

—There is a mapping of SNEIPL relevant eCST universal nodes to fully qualified names of SNEIPL classes representing eCST universal nodes.

—The class that loads an eCST relies on the previously mentioned map to determine the name of the class representing currently processed eCST node. The constructor of the class is invoked using the Java Reflection API.

In other words, the calls to constructors which instantiate universal nodes of loaded eCSTs cannot be detected by any static source code analysis method. Generally speaking, function calls via reflection

are hard to detect statically. On the other hand, static analysis tools should be able to detect indirect function calls made via language-specific keywords. In the case of SNEIPL, the problem of indirect function calls via language-specific keywords can be solved by introducing new eCST universal nodes that are specializations of the FUNCTION_CALL universal node or even more simply by eCST post-processing which does not change the structure of eCST: each *this*/*super* token in an eCST can be simply rewritten by the name of class/super-class (those names are marked by CONCRETE_UNIT_DECL and EXTENDS universal nodes, respectively) in order to make the call explicit by name.

The SNEIPL FUGV network contains 82 isolated nodes (15.81% of the total number of nodes in the network). 4 isolated nodes represent variables, while 78 nodes represent functions. Table II lists isolated nodes that represent class attributes in the SNEIPL FUGV network. As it can be seen only one class attribute defined in SNEIPL is actually unused, while the other three are automatically generated serialization identifiers that are not used by SNEIPL methods.

Table II. Isolated class attributes in the SNEIPL FUGV network.

Attribute name	Explanation
svc.ecst.ECSTTypedNode.typeResolved	Unused
svc.ecst.visualise.Show.serialVersionUID	Serial version UID
svc.ecst.visualise.Visualiser.serialVersionUID	Serial version UID
svc.ecst.ECSTLoaderException.serialVersionUID	Serial version UID

We manually inspected 78 nodes representing methods that are isolated in the FUGV network:

—20 of them represent methods that are defined in classes which do not define any class attributes. A majority of them are nodes representing constructors of classes that correspond to eCST universal nodes (classes that only define a constructor). Other nodes from this category correspond to method declarations in the only interface contained in SNEIPL (*SymTab*).

—Other 58 methods (17.68% of the total number) are methods that are defined in classes which have class attributes, but do not use them. Those are local (private) methods which only process their arguments, methods which do not use class attributes but invoke other methods which access to class attributes, and simple static utility methods.

5. CONCLUSIONS

In this paper we demonstrated that the source code of a static program analysis tool can be used to validate the tool. More specifically, we applied SNEIPL, extractor of software networks, to its own source code. The analysis of extracted package collaboration network from SNEIPL source code showed that SNEIPL is able to recover its own architecture on the highest level of abstraction. We also performed the analysis of isolated nodes in obtained networks. This analysis revealed unused software entities defined in SNEIPL enabling us to improve its implementation.

REFERENCES

David F. Bacon and Peter F. Sweeney. 1996. Fast static analysis of C++ virtual function calls. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*. ACM, New York, NY, USA, 324–341. DOI:http://dx.doi.org/10.1145/236337.236371

David Binkley. 2007. Source Code Analysis: A Road Map. In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 104–119. DOI:http://dx.doi.org/10.1109/FOSE.2007.27

Zoran Budimac, Gordana Rakić, and Miloš Savić. 2012. SSQSA architecture. In *Proceedings of the Fifth Balkan Conference in Informatics (BCI '12)*. ACM, New York, NY, USA, 287–290. DOI:http://dx.doi.org/10.1145/2371316.2371380

Alessandro P. S. de Moura, Ying-Cheng Lai, and Adilson E. Motter. 2003. Signatures of small-world and scale-free properties in large computer programs. *Phys. Rev. E* 68, 1 (Jul 2003), 017102. DOI:http://dx.doi.org/10.1103/PhysRevE.68.017102

Stphane Ducasse, Michele Lanza, and Sander Tichelaar. 2000. MOOSE: an extensible language-independent environment for reengineering object-oriented systems. In *2nd International Symposium On Constructing Software Engineering Tools (COSET 2000)*.

Jrgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. 2002. GUPRO: generic understanding of programs – an overview. In *Electronic Notes In Theorethical Computer Science*, Vol. 72. 47–56. DOI:http://dx.doi.org/10.1016/S1571-0661(05)80528-6

S. Jenkins and S. R. Kirk. 2007. Software architecture graphs as complex networks: a novel partitioning scheme to measure stability and evolution. *Information Sciences* 177 (June 2007), 2587–2601. Issue 12. DOI:http://dx.doi.org/10.1016/j.ins.2007.01.021

Holger M. Kienle and Hausi A. Müller. 2010. Rigi - an environment for software reverse engineering, exploration, visualization, and redocumentation. *Science of Computer Programming* 75, 4 (2010), 247–263. DOI:http://dx.doi.org/10.1016/j.scico.2009.10.007

Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. 2008. Power laws in software. *ACM Trans. Softw. Eng. Methodol.* 18, 1, Article 2 (Oct. 2008), 26 pages.

Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. 1998. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7, 2 (1998), 158–191. DOI:http://dx.doi.org/10.1145/279310.279314

Diego Puppin and Fabrizio Silvestri. 2006. The social network of Java classes. In *Proceedings of the 2006 ACM symposium on Applied computing (SAC '06)*. ACM, New York, NY, USA, 1409–1413. DOI:http://dx.doi.org/10.1145/1141277.1141605

G. Rakić and Z. Budimac. 2011a. Introducing enriched concrete syntax trees. In *Proceedings of the 14th International Multiconference on Information Society (IS), Collaboration, Software And Services In Information Society (CSS)*. 211–214.

G. Rakić and Z. Budimac. 2011b. SMIILE Prototype. In *Proceedings of the International Conference of Numerical Analysis and Applied Mathematics (ICNAAM), Symposium on Computer Languages, Implementations and Tools (SCLIT)*. 544–549. DOI:http://dx.doi.org/10.1063/1.3636867

Gordana Rakić, Zoran Budimac, and Miloš Savić. 2013. Language independent framework for static code analysis. In *Proceedings of the 6th Balkan Conference in Informatics (BCI '13)*. ACM, New York, NY, USA, 236–243. DOI:http://dx.doi.org/10.1145/2490257.2490273

Aoun Raza, Gunther Vogel, and Erhard Plödereder. 2006. Bauhaus: a tool suite for program analysis and reverse engineering. In *Proceedings of the 11th Ada-Europe international conference on Reliable Software Technologies (Ada-Europe'06)*. Springer-Verlag, Berlin, Heidelberg, 71–82. DOI:http://dx.doi.org/10.1007/11767077_6

Miloš Savić, Mirjana Ivanović, and Miloš Radovanović. 2011. Characteristics of Class Collaboration Networks in Large Java Software Projects. *Information Technology and Control* 40, 1 (2011), 48–58. DOI:http://dx.doi.org/10.5755/j01.itc.40.1.192

Miloš Savić, Miloš Radovanović, and Mirjana Ivanović. 2012a. Community detection and analysis of community evolution in Apache Ant class collaboration networks. In *Balkan Conference in Informatics, 2012, BCI '12, Novi Sad, Serbia, September 16-20, 2012*. 229–234. DOI:http://dx.doi.org/10.1145/2371316.2371361

Miloš Savić, Gordana Rakić, Zoran Budimac, and Mirjana Ivanović. 2012b. Extractor of software networks from enriched concrete syntax trees. *AIP Conference Proceedings* 1479, 1 (2012), 486–489. DOI:http://dx.doi.org/10.1063/1.4756172

Miloš Savić, Gordana Rakić, Zoran Budimac, and Mirjana Ivanović. 2014. A Language-independent Approach to the Extraction of Dependencies Between Source Code Entities. *Inf. Softw. Technol.* 56, 10 (Oct. 2014), 1268–1288. DOI:http://dx.doi.org/10.1016/j.infsof.2014.04.011

Craig Taube-Schock, Robert J. Walker, and Ian H. Witten. 2011. Can We Avoid High Coupling? In *ECOOP 2011 Object-Oriented Programming*, Mira Mezini (Ed.). Lecture Notes in Computer Science, Vol. 6813. Springer Berlin Heidelberg, 204–228. DOI:http://dx.doi.org/10.1007/978-3-642-22655-7_10

A. Telea, H. Hoogendorp, O. Ersoy, and D. Reniers. 2009. Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study. In *5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2009)*. 81–88. DOI:http://dx.doi.org/10.1109/VISSOF.2009.5336419

S. Valverde and V. Solé. 2007. Hierarchical small worlds in software architecure. *Dyn. Contin. Discret. Impuls. Syst. Ser. B: Appl. Algorithms* 14(S6) (2007), 305–315.

Lei Wang, Pengzhi Yu, Zheng Wang, Chen Yang, and Qiang Ye. 2013. On the evolution of Linux kernels: a complex network perspective. *Journal of Software: Evolution and Process* 25, 5 (2013), 439–458. DOI:http://dx.doi.org/10.1002/smr.1550

R. Wheeldon and S. Counsell. 2003. Power law distributions in class relationships. In *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation*. 45–54. DOI:http://dx.doi.org/10.1109/SCAM.2003.1238030