periodica polytechnica

Electrical Engineering
55/3-4 (2011) 127–132
doi: 10.3311/pp.ee.2011-3-4.05
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

4D Ariadne the Static Debugger of Java Programs

Zalán Szűgyi / *István* Forgács / *Zoltán* Porkoláb

Received 2012-07-03

Abstract

Development environments support the programmer in numerous ways from syntax highlighting to different refactoring and code generating methods. However, there are cases where these tools are limited or not usable, such as getting familiar with large and complex source codes written by a third person; finding the complexities of huge projects or finding semantic errors.

In this paper we present our static analyzer tool, called 4D Ariadne, which concentrates on these problems. 4D Ariadne is a static debugger of Object Oriented applications written in Java programming language It calculates data dependencies of objects being able to compute them both forward and backward. As 4D Ariadne provides only the direct influences to the user, it can be considered as an alternative of traditional debuggers, without executing the code. 4D Ariadne also provides dynamic call graphs representing polymorphic properties of objects.

Keywords

4D Ariadne · static analysis · Java

Zalán Szűgyi

Department of Programming Languages and Compilers, Eötvös Loránd University, H-1117 Budapest, Pázmány Péter sétány 1/C, Hungary

István Forgács

4D SOFT Kft, H-1096 Budapest, Telepy u. 24., Hungary

Zoltán Porkoláb

Department of Programming Languages and Compilers, Eötvös Loránd University, H-1117 Budapest, Pázmány Péter sétány 1/C, Hungary

1 Introduction

During software development maintaining and refactoring programs or fixing bugs are essential part of this process. Although there are prevalent, good quality tools for the latter, there are only a few really reliable tools to maintain huge projects: finding the complexity of projects, finding the dependencies of different modules, understanding large software codes written by third party programmers or finding semantic errors.

In this paper we present 4D Ariadne [21] that helps the programmer or the software architect to deal with maintenance and program comprehension. 4D Ariadne is a static debugger tool of Object Oriented programs written in Java programming language and it is based on data dependencies of the object. With the aid of 4D Ariadne the programmer obtains the forward and backward data dependencies of objects, browse the call graph, extended with dynamic information at run time, or check the complexity and the modification cost of one part of the program, which can range from a single statement to a whole compilation unit.

In this paper we focus on the background of 4D Ariadne. We present the base data structures and methods used by the upper layers of 4D Ariadne such as its Symbol Table, the 4D Ariadne Syntax Tree, the Control Flow Graph, etc.

Our paper organizes the following way: Chapter 2 describes the base structure of 4D Ariadne. The detailed presentation of 4D Ariadne comes in Chapter 3. Then the most important data structures are described as the following: the 4D Ariadne Syntax Tree in Chapter 4 , the Control Flow Graph in Chapter 5 , the way to handle arrays and collections in Chapter 6 , and the connection between 4D Ariadne Syntax Tree and Control Flow Graph in Chapter 7. The experimental results are presented in Chapter 8. Chapter 9 shows the related work and Chapter 10 the conclusion.

2 Background

The main task of the `Parser` module of 4D Ariadne is to parse the Java project and build up the background data structures for the `Points To Analysis` (PTA) module. There are four main data structures: the symbol table (ST), the `4D`

Ariadne Syntax Tree (DST), the Control Flow Graph (CFG) and the Def Use Info (DUI).

The symbol table stores all the symbols discovered during the parse phase of the source code. The algorithms of PTA calculate both the aliases (reference variables referencing the same objects) and the data dependencies of the variables, thus they heavily read the symbol tables. Each compilation unit has an own symbol table, which makes the search efficient. There is an additional symbol table, called special symbol table, which stores the symbols not related to compilation units (e.g. the class variable of the built in type Object). We differentiate three kinds of symbols: variables, methods and types. To enhance the search the symbol table maintains three maps for these symbol types. To provide fast access to the symbol tables, they are always kept in the memory.

The DST represents a Java compilation unit for the 4D Ariadne. It is built in parallel with the symbol table during the parse phase of the source code. Each compilation unit has its own DST. The reader can learn more about DST in Chapter 4. Control Flow Graphs (CFGs) represent the control structure of the methods in a program. Thus, each method has an own CFG while every statement in a method refers to a subgraph of a CFG. While in theory more statements in a basic block are collected into one node, in 4D Ariadne this is not possible otherwise some information could be lost. More information is to be found about CFG in Chapter 5.

To calculate data dependencies PTA needs to know where and which variables are read or written. The Def Use Info (DUI) provides this information. The variables in def set are modified by the corresponding subexpression, while the variables in use set are read. A variable can be placed both in def set and use set. For example the value of the variable in the expression x++ is both read and written. If we do not know yet which will happen to the variable we put it in the undef set. The nodes of either DST or CFG hold the DUI information.

3 4D Ariadne

4D Ariadne is a static debugger tool of Object Oriented programs written in Java programming language. It is based on data dependencies of the object. In contrast with traditional debuggers, 4D Ariadne does not need to execute the program to debug it, instead it analysis the source code. The key advantage of static debuggers is to start static debuging from a variable at any location. 4D Ariadne provides only direct influences to the user, who can explore the code step-by-step. The major difference between static and traditional debuggers is that while traditional debuggers explore only one execution path for a given input, the static debugger reveals all the possible dependence context starting from a selected program location and variable.

4D Ariadne has been developed as an Eclipse plugin and it is connected to the build system of Eclipse. This means that, when Eclipse builds a compilation unit, 4D Ariadne parses it as well. The build system of Eclipse by default applies the incremental build mechanism, i.e. it immediately builds a compilation unit if it has changed during the coding phase. Thus, the project is always ready to run. With this mechanism the building time of a project is split into small parts, and the programmer does not need to wait for the end of a building process before running the project. While 4D Ariadne is connected to the build system of Eclipse, the project is always parsed and ready to start static debugging.

4D Ariadne special PTA method calculates direct dependences just in time (JIT). Since incremental build is also JIT, thus the whole process is very fast.

4D Ariadne is able to calculate both forward and backward data dependencies. The result of a forward dependence calculation is a set of variables, which depend on the value of the starting variable. The result of the backward dependence calculation is that set of variables, which affect the value of the starting variable. In Fig. 1 a screenshot can be seen of the last step of the forward debug. A forward debugging has been started from variable input. Those statements are shown on the right hand side, which are affected by the starting variable. On the left hand side, in the editor, the variables which are affected by input are marked with green. The dependence chain is finished with variable data. If we start a backward debug from data, we get the same result in reverse order.

4D Ariadne also presents the run time calling context without actually executing the code even at a phase when the code has not been executable. This is called dynamic call graph.

4D Ariadne provides a Magic Score feature which measures the maintenance complexity of a module [7, 12]. The Magic Score estimates how much work is needed to modify the module including regression testing and bug fixing.

With the aid of 4D Ariadne the programmer can get familiar with large third party source codes. It helps to measure and discover the complexity of huge projects, and it is very useful in finding errors, maintaining understanding source code.

4 4D Ariadne Syntax Tree

The 4D Ariadne Syntax Tree (DST) represents a Java compilation unit for the 4D Ariadne. The basic structure of the DST is similar to the Abstract Syntax Tree (AST) [10] built by the compiler, however we store different information on DST nodes. In some cases the AST is too detailed for us, so we contract or skip some AST nodes when creating the DST. In other cases we need more detailed information of a piece of the source code, thus we add extra nodes into the DST, which do not exist in the AST. For example we handle all the literals in the same way, however we need extra DST nodes for logical expressions to be able to treat short-cut operators.

DST nodes can hold three different main information types: the Def Use Info, the connection to Control Flow Graphs, and the File Info.

Only the nodes representing an expression or subexpression contain Def Use Info (DUI). The DUI is null for statements.

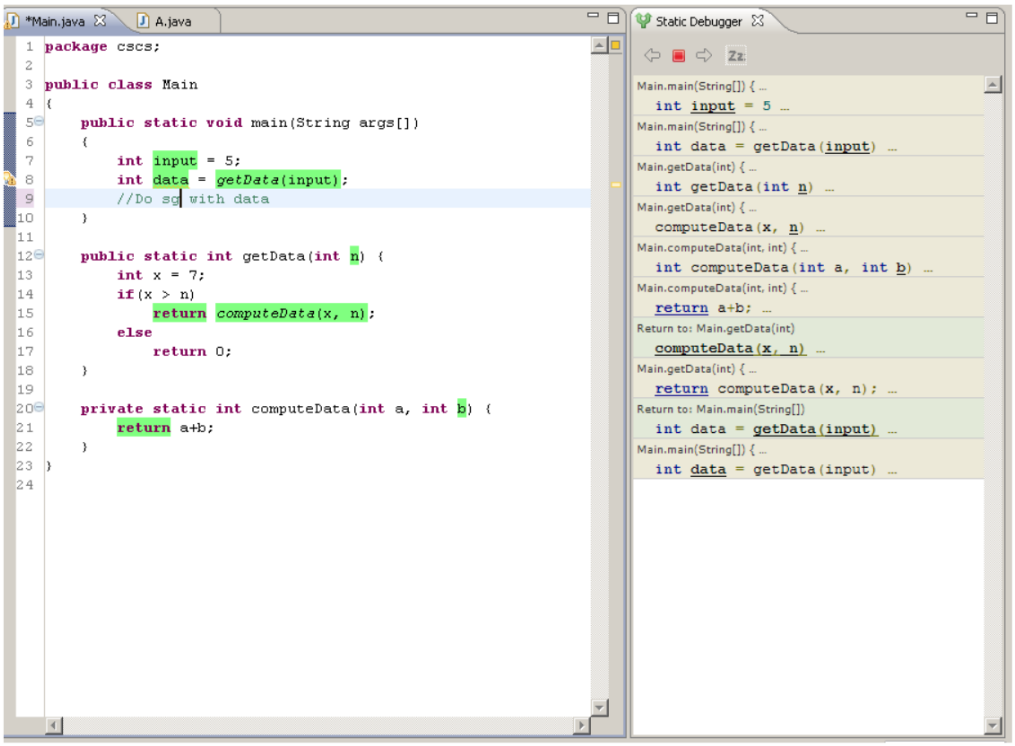

Fig. 1. Forward static debug

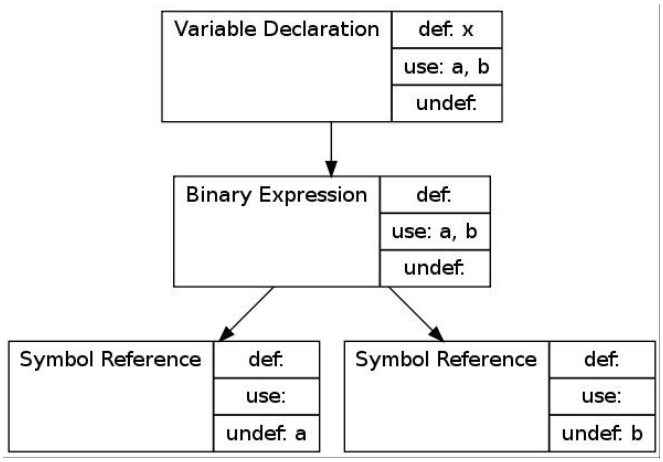

Fig. 2. Forward static debug

The DUI contains three set of variables: the `def set`, the `use set` and the `undef set`. Below we can see a code snippet and its corresponding piece of DST in Fig. 2:

```
int x = a + b;
```

The root of the subtree is a `Variable Declaration` node. Its child node is a `Binary Operation` which refers to the initializer of variable declaration. The children of `Binary Operation` node are the `Symbol References` which are the arguments of the addition operation. The two `Symbol Reference` nodes just refer to the variables `a` and `b`. At the `Symbol Reference` we do not know anything about the variables' future, thus we put them into the `undef set`. When we parse the parent of these nodes, we realize that the two variables are an argument of an addition, hence we put them into the use set. In the root of this subtree we find a `Variable Declaration` node, where the declared variable is `x`, and the initializer is the sum of the variables `a` and `b`. Thus, the `x` is put into the `def set` and `a` and `b` are still in the `use set`.

Every DST node belonging to a statement contains a reference to their corresponding `head` and `tail` CFG node. We describe the CFG in detail in the next chapter.

The `File Info` stores the position of the code snippet belonging to the DST node in a source file. We use this information to keep the connection between the CFG and the source file.

5 Control Flow Graph

4D Ariadne's CFG representation has exactly one start node, called `entry`, and at least one end node, called `exit`. If there are uncatched exceptions thrown in a method, an extra end node is added to the graph. Almost all the nodes have one incoming and one outgoing edge. The exception is the `entry` node, which has no incoming edge, the `exit` node which has no outgoing edges, and the `predicate` nodes (`pnode` for short), which represent a decision in the program, thus they have two outgoing edges. The nodes of the CFG contain further information:

- def use info of the variables
- name of the called method, the actual parameters, etc.

See the following method and its corresponding CFG in Fig. 3.

The child node of the `entry` node is a `declaration and assignment` node representing the first line of a method: vari-

```
int f(int a) {
  int x = 7;
  if (a < x)
    x = a + x;
  else
    x = g(a);
  return x;
}
```

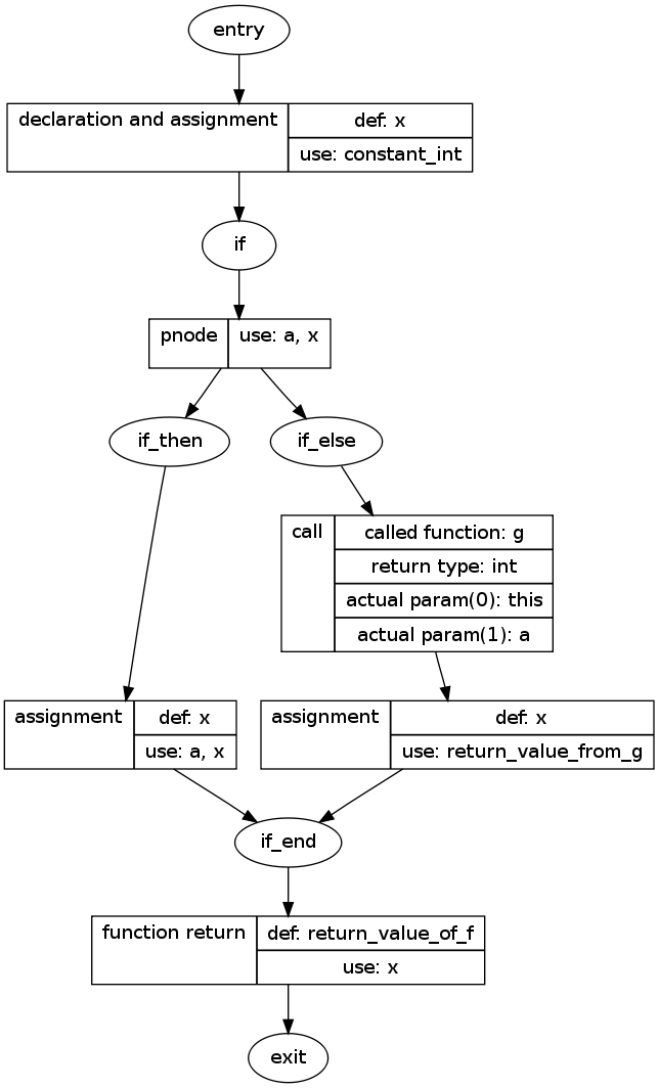

Fig. 3. The CFG of method f

able declaration and initialization. Then the `if statement` is represented by CFG nodes between nodes `if` and `end_if`. The `predicate` node (pnode) represents the condition part of the `if statement` where the variables `a` and `x` are read. In the `then` branch, there is only one `assignment` node where the variable `x` is written and the variables `x` and `a` are read. In the `else` branch first the method `g` is invoked, then the return value of `g` is assigned to `x`. The `call` and the assignment nodes represent this in Fig. 3.

Though 4D Ariadne does not use constant propagation, we need to handle short-cut logical operation. The following code snippet shows the problem:

```
if( x != null && x.isValid() )
```

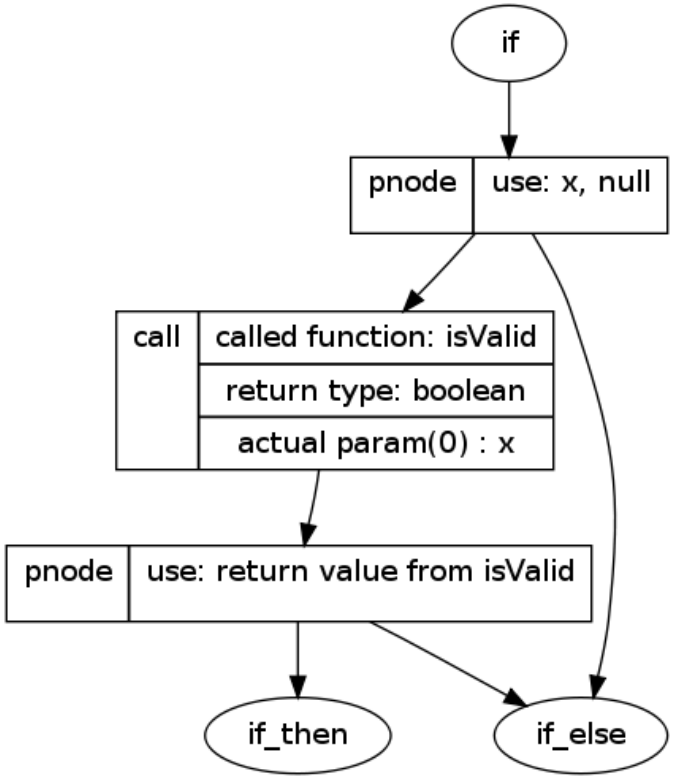

Fig. 4. The CFG of a compound short-cut logical expression

We should statically simulate all the possible cases at run time, therefore the related CFG contains the case whether `x` is not `null`, and also the compound case whether `x` is not `null` and `x.isValid` is true. To simulate this behavior we need as many `predicate` nodes in the CFG, as many arguments we have in a short-cut logical expression. Fig. 4 illustrates the CFG part of the code snippet above.

6 Handling Arrays and Containers

4D Ariadne handles arrays and containers through a common interface. This interface has three logical methods: `set`, `get` and `update`. The `set` method writes the arrays and containers, the `get` reads them and the `update` reads and writes them. 4D Ariadne uses the update method for compound assignments (like: +=) and prefix, postfix operations that can be either increment or decrement. These methods appear only in the internal representation of the project, which do not affect the source code.

For arrays Table 1 shows the usage of the logical interface, where `t` denotes an array of `int` and `x` is a variable of `int`.

Tab. 1. Array handling

source	internal representation
x=t[1];	x = get(t,1);
t[1] = x;	set(t,1,x);
t[1] += x	update(t,1,x);

For containers of the standard library of Java the parser puts `get` and `set` flags to those methods of a container, which reads

or writes the elements. Then the corresponding logical methods are used instead of the flagged methods. In contrast with arrays, in case of containers the `update` logical method is not used. Table 2 shows how the parser flags the method of `LinkedList` container.

Tab. 2. Handling linked list

method	flag
get	get
getfirst	get
getlast	get
set	set

7 Connection Between DSTs and CFGs

DST is an internal representation of the Java project and each compilation unit has an own DST. The nodes of a DST refer to the corresponding part of the CFG. Node `method declaration` refers to the whole CFG while its children refer to the proper part of the CFG. The ancestors of node `method declaration`, however, refers to class-level entities. See the following example and Fig. 5, which shows the connection between DST and CFG.

```
int f() {
  int x = 7;
  return g(x);
}
```

On the left hand side of the figure there is the subtree of DST. On the right hand side there is the CFG. Node `MethodDeclaration` refers to the whole CFG (from the `entry` node to the `exit` node). Node `Block` refers only to the statements in the method (from the `declaration and assignment` to the `function return`). The subgraph of CFG from node `call` to node `function return` belongs to the `ReturnStatement` DST node, and DST node `MethodInvocation` refers only to the `call` node in the CFG.

8 Experimental Results

In this chapter we present our experimental result about effectiveness of 4D Ariadne. 4D Ariadne has two main modules, the parser and the analyzer. The effectiveness depends on this two modules. The analyzer works as a Just In Time analyzer returning prompt answer to any static debug query. The parser module feeds the analyzer module, thus for the first time the whole project has to be parsed before the analysis starts.

Table 3 shows the time consumed to parse projects of different size.

To parse a half a million line project takes about quarter an hour. It might be a bit slow, but the parser module is relying on the incremental build system of Eclipse, which means that the whole project needs to be parsed only once. Later it is already enough to re-parse the modified compilation units only, which is significantly faster.

Tab. 3. Elapsed time to parse projects

Lines of code	time
10000	10 sec
100000	2 min
500000	15 min

Overall, with the incremental build and the on demand analyzer we get a useful and effective static analyzer tool for Java projects.

9 Related Work

In Eötvös Loránd University a refactoring tool is being developed for Erlang programming language, called RefactorErl [8, 11]. They build control flow graphs from program slicing module and they detect parallelizable source code with it [13].

Coverity [17] is developed to find defects in code during the early phase of development. It is able to analyze codes written in C, C++, C# and Java languages. Coverity has both static and dynamic analyzers.

Klocwork [19] developed a source code analysis product suite that is used to mitigate critical issues in code early in the development process. Relying static analysis techniques on C, C++, Java, and C# source code, it provides accurate detection of quality and security issues prior to code check-in.

Parasoft [20] provides a fully-integrated suite for automating a broad range of practices proven to improve software development team productivity and software quality. It covers tools from static analysis, to peer code review, to unit/component testing, to runtime error detection at the unit and application level. It supports Java, C, C++ and .NET languages.

FindBugs [18] is an open source program developed in University of Maryland which looks for bugs in Java code. It uses static analysis to identify hundreds of different potential types of errors in Java programs. FindBugs operates on Java bytecode rather than source code. The software is distributed as a stand-alone GUI application. There are also plug-ins available for Eclipse, Netbeans, IntelliJ IDEA, and Hudson.

Lint [3, 9] is a program to detect suspicious and non-portable constructs in C language source and it is perform static analysis of source code.

Columbus developed by University of Szeged establishes source code quality assurance solutions such as static and dynamic source code analysis [4, 5], measurement and auditing, reverse engineering and re-documentation [2, 14], support for change management [6], assessment and optimization of software testing and continuous measurement [1].

Neither of these tools, however, support static debugging based on just in time data dependence analysis of object oriented code, therefore their usage is different from 4D Ariadne.

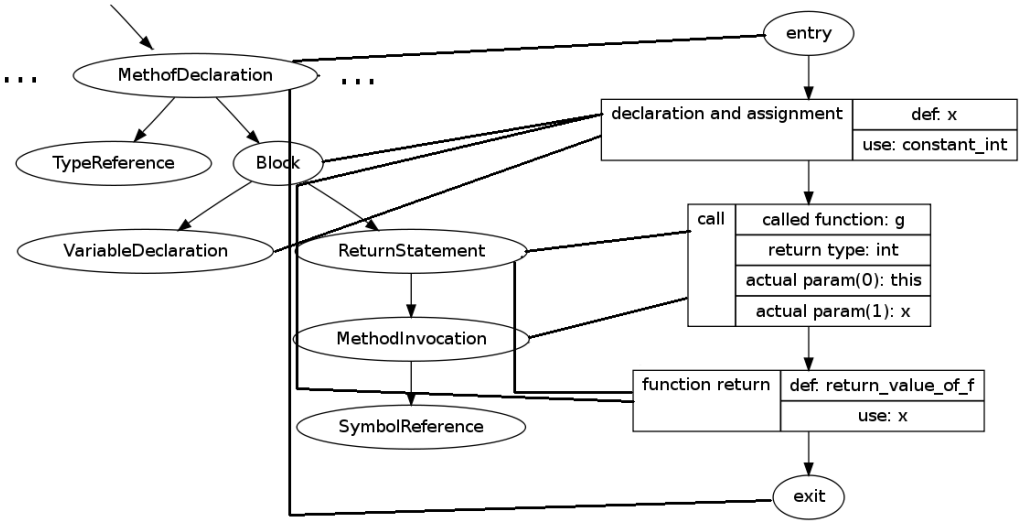

Fig. 5. Connection between DST and CFG

10 Conclusion

In this paper we presented our tool called 4D Ariadne, which is a static debugger based on static analysis and data dependencies of Object Oriented programs written in Java programming language. With the aid of this tool, programmers can easily understand the source code written by a third party programmer, even if it is huge and complex. The tool helps the programmer discover the maintenance complexity of the code. Finding complex semantic errors is also efficient with 4D Ariadne.

In this paper we focused on the compilation part of 4D Ariadne. We detailed the methods and data structures which parse the source code and feed the upper layers that compute the data dependencies and process static analysis.

Finally, we presented our benchmark results, which proved that our tool – with the aid of the incremental build technique and JIT dependence calculation – is efficiently usable with huge industrial projects.

References

1 **Bakota T., Beszédes A, Ferenc R, Gyimóthy T**, *Continuous Software Quality Supervision Using SourceInventory and Columbus*, Companion Material of the 30th International Conference on Software Engineering (ICSE'08), Informal Research Demonstrations (May 2008), 931–932.

2 **Beszédes A, Ferenc R, Gyimóthy T**, *Columbus: A Reverse Engineering Approach*, Pre-Proceedings of IEEE Workshop on Software Technology and Engineering Practice (STEP'05) (September 2005), 93–96.

3 **Darwin I F**, *Checking C Programs with Lint*, O'Reilly, 1991.

4 **Ferenc R, Beszédes A, Gyimóthy T.**, *Fact Extraction and Code Auditing with Columbus and SourceAudit*, Proceedings of the 20th International Conference on Software Maintenance (ICSM'04) (September, 2004), 513.

5 **Ferenc R, Beszédes A, Gyimóthy T.**, *Extracting Facts with Columbus from C++ Code*, Tool Demonstrations of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04) (March, 2004), 4–8.

6 **Ferenc R, Gustafsson J, Müller L., Paakki J**, *Recognizing Design Patterns in C++ programs with the integration of Columbus and Maisa*, Acta Cybernetica **15** (2002), 669–682.

7 **Hericko M, Zivkovic A, Porkoláb Z**, *A Method for Calculating Acknowledged Project Effort Using a Quality Index*, Informatica (Slovenia) **31** (2007), 431-436.

8 **Horváth Z, Lövei L, Kozsik T, Kitlei R, Vig A., Nagy T, Tóth M., Király R.**, *Modeling semantic knowledge in Erlang for refactoring In Knowledge Engineering: Principles and Techniques*, Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT 2009 **54** (Jul 2009), 7–16.

9 **Johnson S**, *Lint, a C program checker*, Computer Science Technical Report 65 (December 1977).

10 **Jones J**, *Abstract Syntax Tree Implementation Idioms*, Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003).

11 **Kozsik T, Csörnyei Z, Horváth Z, Király R, Kitlei R., Lövei L, Nagy T, Tóth M, Vig A.**, *Use cases for refactoring in Erlang*, Central European Functional Programming School **5161/2008** (2008), 250–285.

12 **Sipos A, Pataki N, Porkoláb Z**, *On multiparadigm software complexity metrics*, Pu.M.A **17** (2006), no. 3-4, 469-482.

13 **Tóth M, Bozo I., Horváth Z, Tejfel M**, *nth order flow analysis for Erlang*, 8th Joint Conference on Mathematics and Computer Science, MACS 2010 (2010).

14 **Vidács L, Beszédes A, Ferenc R**, *Columbus Schema for C/C++ Preprocessing*, Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR'04) (March 2004), 75–84.

15 **Weiser M.**, *Program slicing*, Proceedings of the 5th International Conference on Software Engineering (March 1981), 439–449.

16 **Weiser M.**, *Program slicing*, IEEE Transactions on Software Engineering **10** (July 1984), no. 4, 352–357.

17 `http://www.coverity.com/`.

18 `http://findbugs.sourceforge.net`.

19 `http://www.klocwork.com/`.

20 `http://www.parasoft.com/`.

21 `http://www.deeptest.com/`.