BULGARIAN ACADEMY OF SCIENCES

CYBERNETICS AND INFORMATION TECHNOLOGIES • Volume **10**, No 2

Sofia • 2010

Source Code Analysis – An Overview

Radoslav Kirkov, Gennady Agre

Institute of Information Technologies, 1113 Sofia
E-mails: rg_kirkov@yahoo.com agre@iinf.bas.bg

Abstract: *In recent years the need of automatically source-code analysis tools has rapidly grown because of the significant increase of both the amount of the software programs and the program complexity. The present paper describes the main structure, algorithms and techniques implemented in some of the most popular tools for source-code analysis as well as an experimental comparison of such tools. The analysis process as well as a functionality of one of the tools is illustrated by an example of analyzing a sample program Finally some trends for development on modern source-code analysis systems are discussed.*

Keywords: *Source code analysis, code representation, software patterns, testing tools.*

1. Introduction

In the last few decades the amount of the software programs has rapidly increased - around 350 billion line of code existed in year 2000 in the global code base (60% are written in Cobol) [5]. In the year 2015 this value will gain approximately 500 billion and ten years later the global code base should top 1 trillion lines of code [2]. The huge number of software products leads to a growing demand for software developers and improvement of the developers' effectiveness and productivity. Besides the new product development, in many cases the existing programs have to be reengineered in order, for example, to improve the program performance, to

change the platform (operation system), to add new functionalities and technologies, to unbundled a monolithic system to several independent parts, etc.

Before starting to reengineer a software product it is necessary to analyze the source code in order to understand deeply the existing implementation. Unfortunately the documentation is often uncompleted, too old or does not cover the whole functionality and sometimes it is even missing. Moreover, the process of the software analysis in the big systems is usually a time-consuming task, which requires to be implemented by the most experienced programmers in the company. Therefore the source-code analysis is an expensive task and every useful tool automating (even partially) this process could decrease the time and price of the product and improve the overall productivity.

The automatic source code analysis is based on information representing a model (or models) of the program that can be constructed by means of automatic tools. Such models can be designed from the source code (textual, human readable code which is usually compiled to an executable program) or from its artefacts as byte code or execution traces [2]. The source information is presented in an abstract structure that allows further interpretations and manipulations. The analysis algorithms are looking at the model for various kinds of patterns describing possible problems and the final result is a list of warnings grouped by type and ordered by the value of importance.

In the present paper we try present an overview of the existing methods for automatic source code analysis. At the beginning we start with description of the anatomy of the source code analysis by presenting a general structure of a source-code analysis program. Then we illustrate such a structure by an example of source-code analysis checking if the exception (try-catch) block is appropriately defined in an illustrative program. Section 4 presents a brief comparison between the existing tools for automatic source-code analysis. Finally we have pointed some trends for development on modern source-code analysis systems.

2. Anatomy of a source code analysis

The structure of most of the existing automatic source code analysis programs [2, 15, 29, 40] could be separated in 4 main composite parts (blocks) – model construction, analysis and pattern recognition algorithms, patterns knowledge and result representation (Fig. 1).

Initially, an abstract model of a program is constructed from the source code or the binary file of the program under analysis. The analysis and pattern recognition algorithms are looking at the model in order to find some probable problems in the analyzed program – anti-patterns, bug patterns or digression from design patterns. The patterns are stored in a separate block called Patterns Knowledge. *Design patterns* describe some generic solutions and the best practices to recurring software problems including both structural and behaviour aspects of the program. *Anti-patterns* describe some recurring problems that are often solved in a wrong way. A *bug pattern* is a concept describing an abstraction of a recurring bug. It is a commonly occurring error in the implementation of the software design [12]. It

should be mentioned that different problems (instances of known patterns) are discovered by means of different code models (trees or graphs), most suitable for each concrete type of patterns. All problems found are ordered by a priority score and shown to the programmer by means of the result representation block.

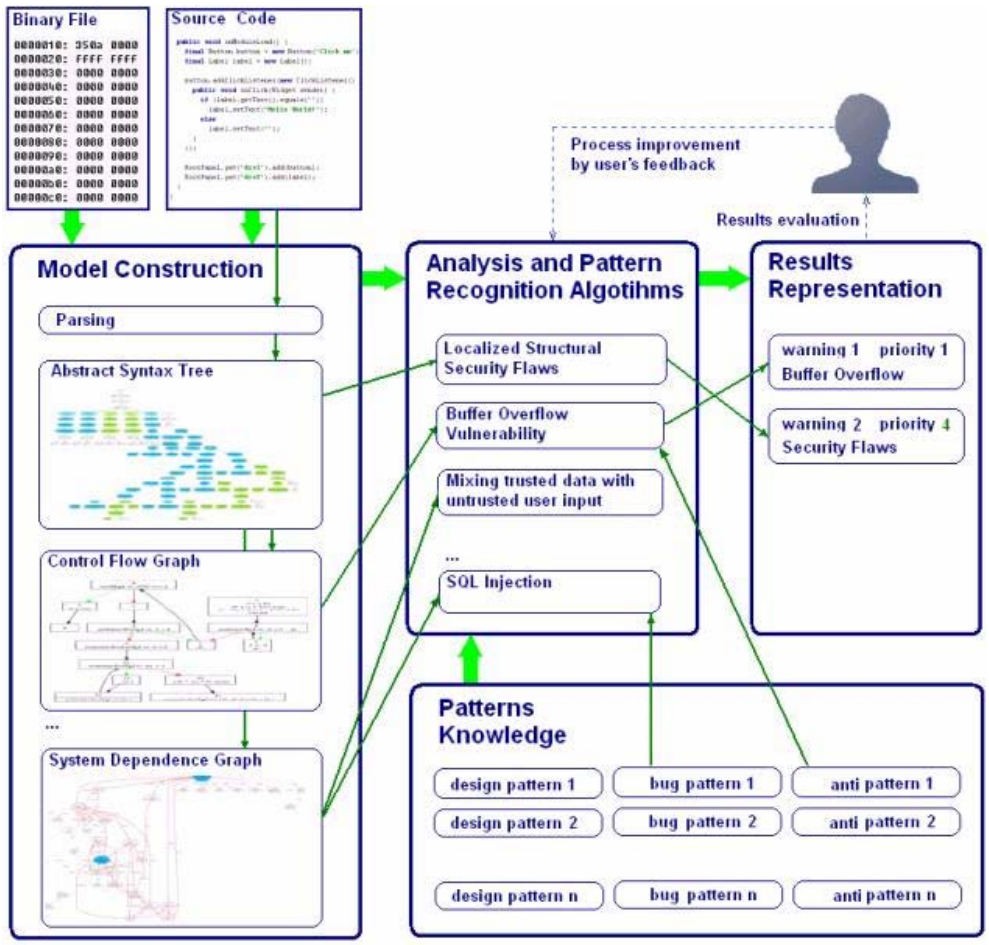

Fig. 1. A general structure of a source code analysis program

2.1. Model construction

Construction of an abstract model of a program is the first step of the source code dependent on a specific programming language can make such analysis easier and faster by processing the model instead of working directly with the source code. Several abstract models can be built from parsing the sources code of a program. They describe different aspects of the program behaviour and are constructed sequentially adding more and more complexity to the previous model.

In many cases [2, 6, 19, 20, 24] the first model created is an Abstract Syntax Tree (AST) – a tree where each node is a construct in the source code In contrast to

the parse tree (built by the parser during the compiling process), AST is insensitive to the grammar that produces it since its structure and elements do not reflect so concretely the syntax of the input language (e.g. some program elements like spacing, brackets, parentheses and comments are removed).

AST is usually used as a base for creating more complex graph structures (models) representing various aspects of the source code and therefore different models are used by different source code analysis algorithms [2]. For example, Control-Flow Graph (CFG) model represents all parts that might be traversed through a program during its execution [24]. Each node in the graph denotes a basic block and the directed edges are jumps in the control flow. *Trace Flow Graph* [2] is used for concurrent programs representation. It is based on CFG model extended with additional vertices and edges for inter-task control flow. Static Single-Assignment (SSA) model [2] simplifies and improves the precision of CFG – the source code variables (e.g. int, double, etc.) are assigned only once making the def-use chains explicit. Value Dependence Graph (VDG) [2, 27] improves some of the results achieved by SSA and simplifies analysis by representation of the control flow as data flow. In addition to the conventional direct-dependence edges *System Dependence Graph* model [11] contains some data-dependence edges representing *transitive* dependences due to the effects of procedure calls. The edges are constructed with the aid of an auxiliary structure that represents calling and parameter-linkage relationships. *Points-to Graph* model [28] serves as an abstraction of the run-time memory states of the analyzed program. Abstract Semantic Graph (ASG) [10] is a data structure that derives the semantic of an expression in a programming language.

2.2. Patterns knowledge

Patterns knowledge is used to represent and store the information about the potential problems in a program source code. Such problems could vary from very simple bugs to sophisticated problems difficult to be found. Some of these problems can be represented by templates or descriptions called patterns that show a problem, or a solution that could be used to overcome the problem. A pattern is a recurring motif, event or structure that occurs over and over again [4]. For source code analysis the most frequently used types of patterns are design patterns, anti-patterns and bug patterns.

2.2.1. Design patterns

A *design pattern* is a generic solution for recurring design problems [26]. The usage of the design patterns improves the effectiveness software development process. The design patterns could be classified into 3 main groups – Creational, Structural and Behavioral [6, 13].

- *Creational design patterns* are related to the creation of classes and objects. Among the most frequently used creational design patterns are *Abstract Factory* (creates an instance of several families of classes), *Builder* (separates object construction from its representation), *Factory Method* (creates an instance of several derived classes), *Object Pool* (avoids expensive acquisition and release of

resources by recycling objects that are no longer in use), *Prototype* (different kinds of objects are specified by a prototypical instance, and the new objects are created by copying this prototype) and *Singleton* (allows only one object of a class to be created).

- *Structural design patterns* describe the composition of classes and objects. Examples of such patterns are *Adapter* (matches interfaces of different classes), *Bridge* (separates an object's interface from its implementation), *Composite* (a tree structure of simple and composite objects), *Decorator* (adds dynamically responsibilities to objects), *Facade* (a single class that represents an entire subsystem), *Flyweight* (a fine-grained instance used for efficient sharing), *Private Class Data* (restricts accessor/mutator access), and *Proxy* (an object representing another object).
- *Behavioral design patterns* are connected with communication between objects. Among them are: *Chain of responsibility* (a way of passing a request between a chain of objects), *Command* (encapsulates a command request as an object), *Interpreter* (a way to include language elements in a program), *Iterator* (sequentially accesses the elements of a collection), *Mediator* (defines a simplified communication between classes), *Memento* (captures and restores an object's internal state), *Null Object* (designed to act as a default value of an object), *Observer* (a way of notifying change to a number of classes), *State* (alters an object's behavior when its state changes), *Strategy* (encapsulates an algorithm inside a class), *Template method* (defers the exact steps of an algorithm to a subclass) and *Visitor* (defines a new operation to a class without change).

2.2.2. Anti-patterns

While a pattern represents the "best practice", an anti-pattern represents the "lesson learned." The notion of "anti-patterns" has two meanings [13]: a) those that describe a bad solution to a problem, which can lead to a bad situation, and b) those that describe how to get out of a bad situation and how to proceed from there to a good solution. Anti-patterns can be valuable because it is often just as important to see and understand bad solutions, as to see and understand the good ones. Sometimes a particular solution seems reasonable at the beginning and it is difficult for the developers to see the problems that can occur. Some of the software anti-patterns are:

- *Software design anti-patternsq e.g.. Gas factory* – an unnecessarily complex design.
- *Object-oriented design anti-patterns*, e.g. *God object* – concentrating too many functions in a single part of the design (class).
- *Programming anti-patterns*, e.g. *Magic numbers* – including unexplained numbers in algorithms.
- *Methodological anti-patterns*, e.g. *Copy and paste programming* – copying (and modifying) existing code rather than creating generic solutions

2.2.3. Bug patterns

Bug patterns are recurring correlations between signalled errors and underlying bugs in a program that describe a commonly occurring error in the implementation of the software design [12]. In contrast to anti-patterns, bug patterns are patterns of erroneous program behaviour correlated with programming mistakes. Some examples of bug patterns are *Dereferencing a null pointer*; *An impossible checked cast*; *Methods, whose return value should not be ignored* and *Infinite recursive loop* [7].

Source Code Analysis programs could automatically find some of the above listed patterns describing unsecured parts of the code, bugs and bad practices [20]. The representation of the patterns can vary depending of the pattern recognition methods chosen. For example, the patterns could be represented as logical conditions grouped into rules or structured into classes for object-oriented programming.

2.3. Analysis and pattern recognition algorithms

The general goal of pattern recognition is the classification of objects into a number of categories. In the software analysis the process is based on matching the patterns against an abstract model, which represents the source code. The more appropriate model could be different for the different kinds of patterns. For example, we could use AST model for finding a problem related to a localized structural security flow. By the help of CFG model we can find a buffer overflow occurrence and the SDG model can be used for searching an inappropriate user input.

Software analysis methods can be classified along different dimensions [2]. One of them splits the methods on static versus dynamic. *Static* analysis does not account for program input; thus the result must be applicable to all executions of the program [1]. It is suitable for the structural recognition, but, in most of the cases, is not appropriate for behaviour aspects. For example, the static analysis can show if a given method could be called, but it cannot provide information how often or even if it will be called at runtime. In contrast, *dynamic* analysis takes into account the program input (typically a single input). This allows greater precision; however, the results are only guaranteed to be correct for the particular input. Another disadvantage of the dynamic approach is the big amount of resources required by the system.

Some techniques lie between the above mentioned. They take into consideration the collection of initial states that, for example, satisfy a predicate. In such situations the static analysis could identify the design pattern candidates and the dynamic approach could then be used to limit the amount of candidates.

Bellow we will give more information about three different kinds of static analysis algorithms – logical queries, relational queries and algorithm based on graph-rewrite rules.

Logical queries algorithms perform logical queries (usually written in a Prolog-like logical language) over the AST model. The descriptions of implementation patterns are expressed as logical conditions grouped into reusable logic rules. The search for solutions is initiated by launching a logic query.

Examples of such Prolog based design patterns detection tools are Pat [45], Pattern-Lint [44] and Goose [43]. In order to increase the flexibility and to be able to search not only for strictly matched patterns, some of the programs use “fuzzy” rules [36] (each rule has a certainty degree). Another language suitable for static analysis is SOUL (Smalltalk Open Unification Language) [30, 36, 37], which is similar to Prolog, but includes some specialized features for meta-programming. The main advantages of this language are: expressiveness of logical languages by nature, built-in pattern matching abilities, backtracking, recursion, flexibility and reuse [36]. As a main shortcoming of SOUL it could be mentioned that for the case of large systems execution of programmes it becomes too slow and the queries become too complex and therefore too difficult for maintenance.

Relational queries algorithms transform the project source code into relations between the elements, for example inheritance, caller/callee, delegations and so on. The “facts” are extracted from the code and then could be queried with relational queries [25]. The analyses algorithms are formalized as relational queries, which are used for detection of design patterns, patterns of problematic design, code clones etc. The approach resembles the logical queries method, but the relations that could be extracted from the source code are more restricted in the sense that not everything contained in the AST could be represented as a relation. The approach could be easily implemented by SQL, has insufficient performance over large graphs (among other inconveniences is the lack of a transitive closure operator [25]).

Algorithms based on graph-rewrite rules use transformational rules over the ASG model. There are different approaches for this method, but we will explain shortly one of them. The first step in it is the source code parsing and the creation of ASG. After that the algorithm searches for the defined design patterns in the graph. Once a pattern is found, the graph is annotated by adding additional nodes and edges that indicate which sub-graph of an ASG corresponds to the specific pattern [40]. As an example of a program that uses this approach, we could point FUJABA [41]. It is a tool providing developers with support for model-based software engineering and re-engineering.

2.4. Results representation

All discovered bugs and anti-patterns are stored as a list ordered according to two parameters – a value of the probability for presence of each issue and its importance reflecting the level of danger of the examined problem.

This list of all issues sorted by the values of probability and importance are the actual output of the system, which presented to the user (developer) via an appropriate user interface (see, as an example, Fig. 2). Usually, at the top of the screen the most danger warnings with the highest probability are displayed. Each of these warnings could contain information about the file name, a line in a file, the class name, the function name where the problem occurs and a user-friendly description of the bug with information what is the best solution for the specific problem.

Fig. 2. Screen shot of FindBugs program, which looks for bugs in Java code

The correctness of the software analysis is usually measured by means of false positive and false negative rates calculating according the following formulas [3]:

$$\text{False Positive Rate (FPR)} = \frac{\text{Number of Incorect Pattern Instances}}{\text{Number of All Detected Pattern Instances}} \times 100\%,$$

$$\text{False Negative Rate (FNR)} = \frac{\text{Undetected Correct Pattern Instances Number}}{\text{Number of Correct Pattern Instances}} \times 100\%.$$

Unfortunately, the only way to evaluate the number of false negatives or false positives is manual analysis of the source code done by an experienced developer.

3. An example of source code analysis

In order to illustrate the above described model of the process of static source code analysis let's consider the following sample problem – there is a small peace of code (Fig. 3), which should be analyzed if the exception (try-catch) block is appropriately defined. More precisely, we will try to find a type of the anti-pattern (Catch for Generic Exception, 2009) that shows if the exception cached is too general.

The try-catch block in Java documentation is described as: A try statement executes a block. If a value is thrown and the try statement has one or more catch clauses that can catch it, then control will be transferred to the first such catch clause. If the try statement has a finally clause, then another block of the code is executed, no matter whether the try block completes normally or abruptly, and no matter whether a catch clause is first given control [14].

The process of finding the solution of this sample problem is based on using the source code analysis system Smart Source Analyzer (SSA) [42] developed by Musala Soft Ltd. **http://www.musala.com/** and its main features include reports generation for software metrics, problem detection and test data set generation.

```
1 import java.io.*;
2 public class Foo
3 {
        private byte[] b;
        private int length;

        Foo()
        {
            length = 40;
            b = new byte[length];
        }

        public void openFile()
        {
            int y;
            try
            {
                    FileInputStream x = new FileInputStream(“z”);
                    x.read(b,0,length);
                    x.close();
            }
            catch(Exception e)
            {
                    System.out.println(“Oopsie”);
            }

            for(int i = 1; i <= length; i++)
            {
                    if (Integer.toString(50) == Byte.toString(b[i]))
                            System.out.print(b[i] + “ ”);
            }
        }
22 }
```

Fig. 3. The sample source describes a simple class that contains a Java function, which reads a text from a file system and displays the text into a console

3.1. Model construction

The first step of the analysis is to parse the code and to transform it onto AST representation. Each element of the source file is represented as a subclass of AST node providing specific information about the object it represents (Fig. 4).

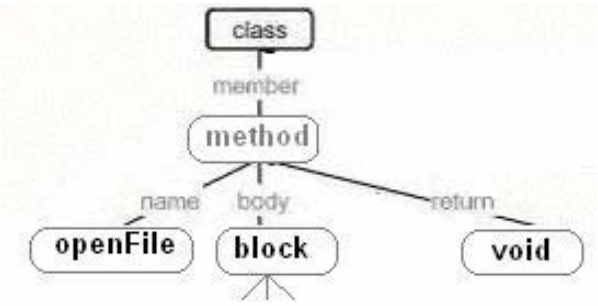

Fig. 4. The root of the AST representation of the openFile function

The next step is conversion of the constructed AST to ASG model. In recent years OMG consortium [10], has designed modeling specifications for ASG called Knowledge Discovery Metamodel (KDM) [10] that is used for generating the ASG model in SSA tool [42]. One of the benefits of KDM is that it standardizes existing approaches to knowledge discovery in software engineering artefacts and allows interchange between different tools. The standard XML format called XML Metadata Interchange (XMI) implements this program interchange possibility.

In ASG each code element is represented by a node (vertice). The try-catch block has nodes in the graph for the following blocks – a try block, every catch block and a final block. The nodes are connected by arcs (edges) that represent the relations between them. In KDM standard each exception block has several relations called Flows:

- *EntryFlow* – from the try action element to the try block.
- *ExceptionFlow* – from the try block to each catch block.
- *ExitFlow* – from the try block to the final block.
- *Flow* – from the try block and each catch block to the final block.
- *Reads* – from the try action element to the result data element of the variable declaration.

Once the ASG model or the example Java function has been built it can be used for further manipulations analysis – in our tool we have implemented a functionality, which allows creation of a Control Flow Graph (CFG). The CFG model of our Java function is presented in Fig. 5.

Fig. 5. CFG model of the function used in the example. Each of the nodes represents an element in the source code and the edges are jumps in the control flow (in our case, there are different types of edges – Flow, Exception Flow, True Flow, etc.)

3.2. Patterns knowledge

In our example we are looking for one type of anti-pattern – declaration of catch for *Generic Exception*. Exceptions are used in a program to notify that an error or exceptional situation has occurred and that it doesn't make sense to continue the program flow until the exception has been handled. For example, if we try in Java to open a file that doesn't exist, an exception of the type *FileNotFoundException* will be thrown.

Many different kinds of *Exceptions* can be ordered in a hierarchal structure (Fig. 6).

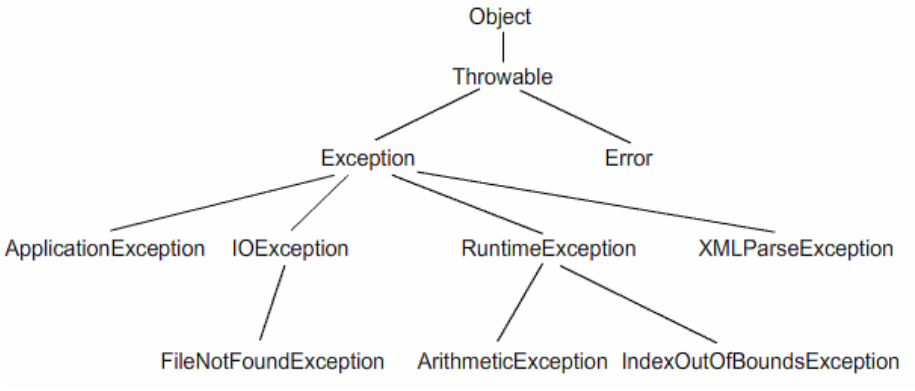

Fig. 6. Part of Java exception tree

In general, the exception definition contains try, catch and final blocks (catch and final are not obligatory, but at least one of them must exist) (Fig. 7).

```
try
{
        FileInputStream x = new FileInputStream("c:\test.txt");
        x.read(b,0,length);
        x.close();
}
catch (Exception e) {…}
```

Fig. 7. An example of an exception block

The catch block handles the exceptions thrown in the try block. It could handle the exact type of the thrown exception (e.g. *FileNotFoundException*) or an exception from its parent types (*IOException*, *Exception*, etc.). Catching a too general exception is a bad practice since it can obscure exceptions that deserve special treatment or that should not be caught at this point in the program. In order to avoid handling the exception in the function, one should set throwing statement in the function definition for the specific exception.

The source code analysis tool used (SSA) stores all kinds of *Patterns* in a specific format, in which each Pattern definition has four elements – set of characteristics, algorithms used for finding the places of the bad practices; filters for omitting all elements that are not in the scope of interest for the specific Pattern; and aggregator, combining results of different algorithms used for finding the Pattern.

In the example analyzed the try-catch pattern has the following characteristics:

- **try** – describes specific information of the try block;
- **catch** – catches the exception, but it is not interested in the specific exception type;

- **throwing** – defines exceptions that will be skipped and will not be processed.
- **exception** – defines generic type of the caught exception.

The applied filter will help us to find all the elements in the source code corresponding to the characteristics of the type “try”. All such elements will be stored in a list and the rest of the code will be ignored.

3.3. Analysis and Pattern Recognition algorithms

The algorithm used in SSA system searches the ASG or CFG model for specific elements of a specified type. In our case, with the assistance of the concrete filter element specifying the Pattern definition, it returns a list of all try-catch blocks providing us with all elements that can catch an *Exception* and are included in the *try* element. As it can be seen from the source code of the example, it contains one constructor and two functions in the try block (see Fig. 6). Based on Java documentation these elements can throw only *FileNotFoundException* and *IOException*:

public ***FileInputStream****(String name) throws* ***FileNotFoundException***

public int ***read****(byte[] b, int off, int len) throws* ***IOException***

public void ***close****() throws* ***IOException***

Since *FileNotFoundException* is a kind of *IOException*, which is a successor of *Exception* in the exception tree (Fig. 5), and taking into account that the most general exception in our example is the *IOException*, the algorithm proposed has found out that the *Exception* used in the catch block is too generic. In other words, we have found an instance of the anti-pattern for inappropriate try-catch definition.

3.3. Results representation

At the end of our cycle of the program, we receive a warning for the presence of anti-pattern of the type “Catch for Generic Exception”. In the dialog of the form detailed information is shown about the problem and the exact place where it is found in the code. If we have more than one issue they will be grouped and sorted by the value of importance and probability.

4. Existing programs and tools

An experimental evaluation of several publicly available bug-finding tools is presented in [7]. For their evaluation the authors have selected five well-known tools – PMD [8], FindBugs [38], JLint [9], ESC/Java [39] and Bandera [40], that have been tested on several tests. Among them is the Java code example (shown in

Fig. 3) used as an illustration in this paper. The warnings found by the tools are shown in Fig. 7:

PMD

Warning (line 15): “Avoid unused local variables” for variable y.

FindBugs

Warning (line 19): “Method ignores results of InputStream.read()”. This function returns the number of bytes read, which could be less than the expected number

Warning (line 20): “Method may fail to close stream on exception”

Warning (line 29): Wrong usage of “==” for String objects comparison

ESC/Java

Warning (line 29): “Array index possibly too large” for array “b” with index *i*

Warning (line 29): Possible null dereference for array “b”.

JLint

Warning (line 29): Wrong usage of “==” for String objects comparison

Fig. 7. The list of warning issued after analyzing the example [7]

As one can see, different tools have found different types of problems. For example, PMD found the unused variable y on line 15, which is an instance of false negative while FindBugs, which was oriented for searching such type of bugs, missed it. On the other side the second warning of ESC/Java is the false positive – there is nothing wrong in the code since b is initialized in the constructor and it can not be null on line 29. “Wrong string comparison” is the only issue found by more than one tool (FindBugs and JLint) – other warnings are not overlapped.

The types of problems found by the systems used in the experimental comparison are presented in Table 1.

One of the explanations of these results is that the tools used in the experiments explore different methods for analyzing the source code. FindBugs, JLint and PMD apply the syntactic bug pattern detection. JLint and FindBugs also use a dataflow component. ESC/Java applies the theorem proving method, while Bandera implements the model checking.

The tools have been also compared by the number of generated warnings and running time over 5 open-source projects (Azureus [31], Art of Illusion [32], Tomcat [33], Jboss [34], and Megamek [35]) (Table 2). It can be seen that ESC/Java is the slowest program while JLint is the fastest one.

Table 1. Types of bugs found by the tools (V – tool checked for bugs in this category *– tool checked for this specific example) [7]

Bug Category	Example	ESC/Java	FindBugs	JLint	PMD
General	Null dereference	V*	V*	V*	V
Concurrency	Possible deadlock	V*	V	V*	V
Exceptions	Possible unexpected	V*			
Array	Length may be less than zero	V		V*	
Mathematics	Division by zero	V*		V	
Conditional, loop	Unreachable code due to constant		V		V*
String	Checking equality using == or !=		V	V*	V
Object overriding	Equal objects must have equal		V*	V*	V*
I/O stream	Stream not closed on all paths		V*		
Unused or duplicate statement	Unused local variable		V		V*
Design	Should be a static inner class		V*		
Unnecessary statement	Unnecessary return statement				V*

Table 2. Running times and warnings generated by each tool [7]

Name	NCSS (Lines)	Class Files	Time (min:sec.csec)				Warning Count			
			ESC/ Java	FindBugs	JLint	PMD	ESC/ Java	Find Bugs	JLint	PMD
Azureus 2.0.7	35,549	1053	211:09.00	01:26.14	00:06.87	19:39.00	5474	360	1584	1371
Art of Illusion 1.7	55,249	676	361:56.00	02:55.02	00:06.32	20:03.00	12813	481	1637	1992
Tomcat 5.019	34,425	290	90:25.00	01:03.62	00:08.71	14:28.00	1241	245	3247	1236
JBoss 3.2.3	8,354	274	84:01.00	00:17.56	00:03.12	09:11.00	1539	79	317	153
Megamek 0.29	37,255	270	23:39.00	02:27.21	00:06.25	11:12.00	6402	223	4353	536

5. Future work

The modern tendencies for improving the existing automatic source code analysis system can be searched in application of algorithms and techniques from such areas of Computer Science as Information Retrieval, Machine Learning and Data Mining.

Existing applications of Information Retrieval (IR) techniques to source-code analysis include automatic link extraction, concept location, software and website modularization, reverse engineering, software reuse impact analysis, quality assessment, and software measurement [2]. In such applications the code is treated as text instead of considering its structure. For example, the source code could be divided into two documents: one includes the comments and another – the executable source code. The cosine similarity between the two documents is measured and used as a proxy for evaluating the program quality. Some empirical evidence supports this technique [2] in cases where automated techniques have been found lacking and where direct human assessment is too expensive.

At the moment the application of IR has concentrated on processing the text constructed from the source and non-source software artifacts (which can be just as important as the source) using only a few well-developed IR techniques. Having in mind the growing importance of non-source documents, in the near future the source-code analysis should develop new IR-based algorithms specifically designed for dealing with the source code.

Application of Data Mining (DM) algorithms promises to improve the overall process of source code analysis. DM techniques, such as neural networks, association rules and decision trees have advanced dramatically in recent years and can be used for mining of software related data [2]. For example, decision trees can be used to discover classification rules for chosen attributes of a data set by systematically subdividing the information contained in this set. At the present moment some researchers try to reuse simple data mining techniques, such as association mining and clustering from the source code analysis [2], but more advanced data mining methods could also be useful for solving this specific task.

Machine Learning algorithms can be used for improving the values of importance and probability of the issues found by the source code analysis tools. For example, the developers' feedback on inferred warnings can be used to adjust automatically the priority of the warnings. Another possibility for priority evaluation of the warnings is to store statistical information every time when the tool is executed over a specific source code [18]. The information can be used to check which issue has been resolved so far by the developer and which one still exists assuming that such an issue is probably not a real issue or is not so important and therefore its priority could be decreased.

Acknowledgements. We would like to thank members of NIF project in Musala Soft Ltd. for providing useful articles for our research. Special thanks to Haralambi Haralambiev, Bojidar Penchev and Svilen Marchev for explaining the details of Smart Source Analyzer structure and functionality. This work is partially supported by the European Social Fund and Bulgarian Ministry of Education, Youth and Science under Operative Program "Human Resources Development", Grant BG051PO001-3.3.04/40, and by the project IIT-010096 "Methods and Tools for Knowledge Processing".

References

1. Chess, B., J. West. Secure Programming With Static Analysis. Indiana, USA, Addison-Wesley, 2008.
2. Binkley, D. Source Code Analysis: A Road Map. Future of Software Engineering. L. Briand and A. Wolf, Eds. IEEE-CS Press, 2007.

3. Lee, H., H. Youn, E. Lee. A Design Pattern Detection Technique that Aids Reverse Engineering. – International Journal of Security and its Applications, Vol. **2**, January 2008, No 1, 1-12.
4. Demeyer, S., S. Ducasse, O. Neirstrasz. Object Oriented Reengineering Patterns. Switzerland, Square Bracket Associates, 2008.
5. Wilkinson, S. From the Dustbin, Cobol Rises. – eWeek, Vol. **18**, May 28, 2001, No 21, p. 58.
6. Gamma, E., R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object-Oriented Software. J. C. Escher / Cordon Art – Baarn – Holland, 1994.
7. Rutar, N., C. B. Almazan, J. S. Foster. A Comparison of Bug Finding Tools for Java. – In: Proc. of the 15th International Symposium on Software Reliability Engineering (ISSRE'04), 2004, 245-256.
8. Copeland, T. PMD/Java, 2005.
http://pmd.sourceforge.net
9. Artho, C. JLint, 2001.
http://artho.com/jlint
10. KDM 1.0 specification
http://www.omg.org/technology/documents/modernization_spec_catalog.htm
11. Horwitz, S., T. Reps, D. Binkley. Interprocedural Slicing Using Dependence Graphs. – ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. **12**, January 1990, Issue 1, 26-60.
12. Farchi, E., Y. Nir, S. Ur. Concurrent Bug Patterns and How to Test Them. – In: Proc. of the International Parallel and Distributed Processing Symposium (IPDPS'03), 2003, p. 286b.
13. Appleton, B. Patterns and Software: Essential Concepts and Terminology, 2000.
http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html
14. The Java Language Specification. Third Edition. Sun Microsystems, Inc., 2004.
http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html
15. Larson, E. SUDS: An Infrastructure for Dynamic Software Bug Detection Using Static Analysis. – ACM SIGSOFT Software Engineering Notes, Vol. **31**, November 2006, Issue 6, 1-2.
16. Okun, V., W. Guthrie, R. Gaucher, P. Black. Effect of Static Analysis Tools on Software Security: Preliminary Investigation. Conference on Computer and Communications Security. – In: Proc. of the 2007 ACM workshop on Quality of protection, ACM New York, 2007, 1-5.
17. Ware, M., C. Fox. Securing Java Code: Heuristics and An Evaluation of Static Analysis Tools. Conference on Programming Language Design and Implementation. – In: Proc. of the 2008 Workshop on Static analysis, 2008, 12-21.
18. Kim, S., M. Ernst. Which Warnings Should I Fix First? – In: Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, 2007, 45-54.
19. Hovemeyer, D., W Pugh. Finding Bugs is Easy. – ACM SIGPLAN Notices, Vol. **39**, December 2004, Issue 12, 92-106.
20. Quinlan, D., R. Vuduc, G. Misherghi. Techniques for Specifying Bug Patterns. – In: Proc. of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging, London, United Kingdom, 2007, 27-35.
21. Kim, S., K. Pan, J. Whitehead. Memories of Bug Fixes. – In: Proc. of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Portland, Oregon, USA, 2006, 35-45.
22. Foster, J., M. Hicks, W. Pugh. Improving Software Quality with Static Analysis. – In: Proc. of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, San Diego, California, USA, 2007, 83-84.
23. Cheng, H., D. Lo, Y. Zhou, X. Wang, X Yan. Identifying Bug Signatures Using Discriminative Graph Mining. – In: Proc. of the Eighteenth International Symposium on Software Testing And Analysis, Chicago, IL, USA, 2009, 141-152.
24. Howarth, N. 1995. Abstract Syntax Tree Design, Cambridge CB3 0RD, United Kingdom, August 1995.

25. Beyer, D., A. Noack, C. Lewerentz. Simple and Efficient Relational Querying of Software Structures. – In: Proc. of the 10th Working Conference on Reverse Engineerin, IEEE Computer Society Washington, DC, USA, 2003, p. 216.
26. Duda, R., P. Hart, D. Stork. Pattern Classification. 2nd Edition. Wiley, New York, 2001.
27. Harrold, M., G. Rothermel. Notes on Representation and Analysis of Software. Georgia Institute of Technology, Oregon State University. December 2002.
28. Milanova, A., A. Rountev, B. Ryder. Parameterized Object Sensitivity for Points-to Analysis for Java. – ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. **14**, January 2005, Issue 1, 1-41.
29. Niere, J., M. Meyer, L. Wendehals. User-Driven Adaption in Rule-Based Pattern Recognition. Technical Report tr-ri-04-249, University of Paderborn, Paderborn, Germany, June 2004.
30. Smalltalk Open Unification Language (SOUL).
http://soft.vub.ac.be/SOUL/
31. Azureus.
http://azureus.sourceforge.net/
32. Art of Illusion.
http://www.artofillusion.org/
33. Tomcat.
http://tomcat.apache.org/
34. Jboss.
http://www.jboss.org/jbossas
35. Megamek.
http://megamek.sourceforge.net/
36. Roover, C., J. Brichau, T. D'Hondt. Combining Fuzzy Logic and Behavioral Similarity for Non-Strict Program Validation. – In: Proc. of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, Venice, Italy, 2006, 15-26.
37. De Roover, C., T. D'Hondt, J. Brichau, C. Noguera, L. Duchien. Behavioral Similarity Matching using Concrete Source Templates in Logic Queries. – In: Proc. of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Nice, France, 92-101.
38. FindBugs.
http://findbugs.sourceforge.net/
39. ESC/Java.
http://secure.ucd.ie/products/opensource/ESCJava2/
40. J. Niere, W. Schäfer, J. Wadsack, L. Wendehals1, J. Welsh1. Towards Pattern-Based Design Recovery. – In: Proc. of the 24th International Conference on Software Engineering, Orlando, Florida, 2002, 338-348.
41. FUJABA.
http://www.fujaba.de/
42. Smart Source Analyzer (SSA).
http://www.musala.com/ssa/
43. GOOSE.
http://esche.fzi.de/PROSTextern/software/goose/
44. Sefika, M., A. Sane, R. H. Campbell. Monitoring Compliance of a Software System with its High-Level Design Models. – In: Proc. of the 18th International Conference on Software Engineering (ICSE 1996), 1996, 387-396.
45. Krämer, C., L. Prechelt. Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. – In: Proc. of the 3rd Working Conference on Reverse Engineering (WCRE 1996), 1996, 208-215.