Practically Applicable Formal Methods

Jędrzej Fulara and Krzysztof Jakubczyk

Institute of Informatics
University of Warsaw
ul. Banacha 2
02-097 Warsaw, Poland

Abstract. Formal methods are considered to be highly expensive. Therefore, they are currently applied almost only in high risk software development. In this paper, we show that formal techniques can be also efficiently used in standard large-scale applications. We focus on the generation of specifications which state the termination condition of `for` loops in Java code (expressed as so called Java Modeling Language *decreases* clauses). We demonstrate that with help of relatively simple techniques it is possible to successfully generate the clauses for almost 80% of the loops in a number of widely deployed applications. Moreover, it turns out that the remaining 20% cases contain loops which should be carefully reviewed by software quality assurance personnel. The results show that our technique might be helpful in spreading the usage of formal methods onto typical business software.
Keywords: formal methods, annotation generation, Java, termination

1 Introduction

Commercial applications often do not make use of specification and verification techniques, because people believe that such theoretical solutions are not applicable to software created for the real market. Business managers know that employing formal methods would increase quality of produced software, but these methods are considered to be very expensive and impractical. Therefore program specification and verification methods are used only in critical components or high risk software, e.g. avionics [10] or in nuclear power plants control software [23], places where correctness and reliability is crucial. It is considered to be not profitable to apply these techniques to standard code.

In standard program production unit tests [4] are used to assure clients that created code meets their requirements and to assure code producers that the code is faultless. The structure of the majority of specifications languages, such as JML [19] for Java or Microsoft Code Contracts for .NET platform [3] follows the structure of programming languages, so it seems that specifications should be written by the same team that has developed the code. Since the most expensive resource in the code production is programmers' time, letting them write annotations along with the code would increase overall costs, making company not competitive on the market. To reduce costs of verification techniques, one

can generate annotations automatically. They can be used later by static analysis tools (e.g. ESC/Java 2 [9]). Automatic generation of annotations has already been explored and several generators were presented, e.g. CANAPA [8] for JML, Houdini [15] for ESC/Java or Daikon [14] for various programming languages.

In this paper, we present an approach to develop formal techniques which may be useful in typical code. This approach relies on existing tools that check if the code realises given specifications, like ESC/Java 2 or JmlRAC [7] but it broadens their applicability by automatic generation of specifications based on the existing code. If the code is mature and its functionality was well tested using common testing techniques, generated specifications may still be helpful, when someone wants to turn the code into a library or may act as documentation for maintenance. If the code is immature, automatic generation of specifications describing correctness properties of the program (such as loop termination, absence of NullPointerExceptions and ArrayIndexOutOfBoundsExceptions, etc.) combined with static checking can give feedback on what errors are present in the code. With these scenarios in mind we have developed a tool which helps to build annotations generators and reveals code fragments that are not obvious and should be reviewed by a human. In addition we have conducted an experiment which shows that our approach, using even very basic techniques, can give promising results on real, large-scale code.

Applicability of formal methods has been discussed in [5], but the considerations are mostly limited to safety-critical systems. A great number of examples of formal method applications can be found in [12]. There exist also tools that analyse various source code properties. For example, Eclipse Metrics plugin [22] helps to find unnecessarily complex places in the code. Semmle, a more powerful, commercial code analyser, comes with its own Query Language [21]. It allows to create and run your checks to enforce specific architecture rules and coding standards. FindBugs [17] detects code instances that are likely to be errors. Our *CodeStatistics* is designed to find any user defined patterns and is able to insert annotations into the code. In [2], statistics collected on Java libraries are used as a motivation to focus on particular aspect of the code in termination analysis. In our approach statistics are used to estimate coverage of a given formal method.

1.1 JML

The Java Modelling Language is a behavioural specification language for Java programs [18]. It allows to write specifications in the design-by-contract fashion, introduced by B. Meyer for Eiffel [20]. JML includes annotations which make possible to describe the invariant properties that are maintained by objects, method specifications (pre- and postconditions) and some lower level properties of the code (e.g. loop invariants). JML annotations are written in Java comments, so they do not disturb standard Java compilers. JML already has very rich tool support [6].

In our research presented in Section 3 we focus on generation of *decreases* clauses to prove loop termination. These clauses consist of the *decreases* JML keyword followed by expression whose value is decreased in each loop iteration

(and cannot be smaller than 0). Figure 1 shows a simple loop together with a valid *decreases* formula.

```
//@ decreases 15 - i;
for (int i = 0; i < 15; i++) {
    // Loop body that does not change i
}
```

Fig. 1: Simple for loop with *decreases* formula.

1.2 Formal Methods

This work is an attempt to answer the question what does it mean to create specification-based formal methods suitable for typical large-scale business applications, where code correctness is not considered to be critical. A formal method designed for the real market cannot increase development costs too much. The "80/20" rule applies to the development process: 20% of work produces 80% of a project — big parts of systems can even be generated automatically from requirements or specifications [16], so business managers may expect the same from formal methods. Thus we should provide them solutions that cover automatically at least 80% of the code and leave the remaining 20% (often unclear, messy and unnecessarily complicated) to be reviewed by the programmers.

If we could show that a formal method gives acceptable results on various, existing projects, then the code producers might expect that this method would be applicable also to their software. Thus, to rate the created formal method, a large set of well known projects should be used.

Modern programming languages offer developers big flexibility in creating the code. This may lead to some inconsistencies in the code that can result in errors which are hard to find. Using tests we can check, if the code meets functional requirements, but to find programming errors (such as null pointer dereferences, not terminating loops etc.) that occur in rare, but sometimes critical situations, we should employ other techniques. This is especially important when the code changes its original context of use, what often happens in case it is turned to a library or is subject to maintenance tasks. Generating logical conditions assuring absence of such inconsistencies may be the first step in introducing formal methods in real, large-scale code. Such conditions can be safely generated automatically. However specifications that describe program semantics (for example method pre- or postconditions) might be generated only for mature and well tested code. When they would be generated from incorrect code, they could be even misleading.

We suggest building simple verification methods iteratively. This process involves following steps:

1. Select code constructions you want to handle — e.g. for loops

2. Execute created method on the selected set of projects.
3. Find cases that are not handled yet.
4. From these cases pick up one for which you are able to provide generic solution.
5. Mark as solved cases covered by the solution.
6. Generate statistics and calculate effectiveness.
7. If effectiveness does not meet the selected threshold, go to step 2.

To apply this strategy in practice, one has to identify interesting constructions and their frequencies in the real code. Collecting frequencies should be done automatically, based on a big, representative sample of software. In our work we have developed a flexible tool to compute necessary statistics on the code and estimate how effective given annotation generation (and verification) method is.

For example let us classify Java `for` loops in the context of generating termination condition. First step will be to identify all interesting code constructions – that is all `for` loops that appear in the code. Next we need to take a look at all loops and try to find patterns, for which we could easily generate the termination condition. The simplest pattern is a `for` loop which increments a counter up to some constant, see Figure 1. Termination condition for this case is trivial.

Now we should generate statistics — how many loops matching this pattern are present in the code and discover what part of all `for` loops is covered by this case. After this we know for how many loops we can generate termination condition. If this coverage does not meet the selected threshold, we should do another iteration — look at the `for` loops that are not covered and produce new patterns.

Each iteration of the presented methodology can take advantage of different strategy and technique as far as the goal — achieving desired threshold, is accomplished. The main advantage of the presented procedure is a possibility to use different verification techniques, each applied where it performs the best.

A formal method created using this approach will give satisfying results on a large and representative set of projects. It should convince managers and developers that such solution can be successfully applied in their work.

In our research we have checked if it is possible to achieve, using simple methods, the expected by the market 80% coverage for a practical problem. We have applied the above described methodology to generate annotations for loop termination in Java programs. Our results are described in Section 3.

2 The *CodeStatistics* Tool

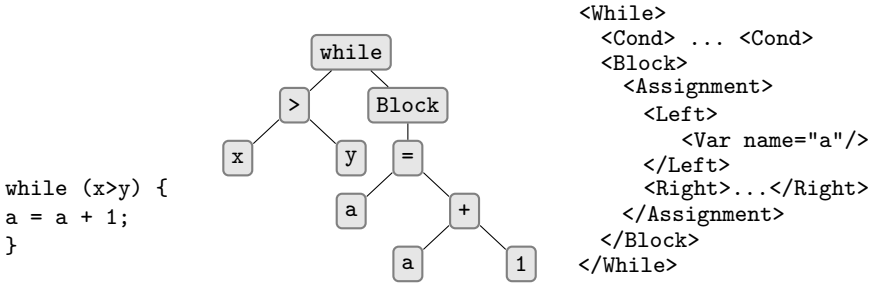
To make the above described approach applicable in practice, a tool that helps to rate the effectiveness of a verification method for different programming languages, is necessary. In our work we have developed *CodeStatistics*[1], a tool for Java. It can recognize patterns specified by the user, count their frequencies in given Java projects and output all matching code fragments, together with their

[1] Available at `http://www.mimuw.edu.pl/~fulara/CodeStatistics`

locations in the project. Patterns are defined in XPath and correspond to nodes from the Abstract Syntax Tree of the processed project.

2.1 Source Code Representation

Source code can be represented as an AST [1]. Each node in AST denotes a construct occurring in the source code. Figure 2 contains an example of AST representation for a small code fragment. Patterns that *CodeStatistics* finds, can be



(a) Source code (b) Abstract Syntax Tree (c) XML representation

Fig. 2: Sample AST representation

expressed as structural conditions for the AST. Our tool is generic. The patterns are not hardcoded, but supplied by the user as a part of the system configuration. They should be written in a simple and expressive language. We have decided to adapt XML as an AST text representation format and XPath as a query language (described in Section 2.2).

2.2 XML and XPath

The Extensible Markup Language (XML) is a general-purpose specification for creating custom markup languages recommended by the World Wide Web Consortium (W3C). It is used in our tool to represent the AST.

In *CodeStatistics* user defines queries in XPath. XPath is a language for finding information in an XML document and provides the ability to navigate around the tree, selecting nodes by a variety of criteria. A path notation is used to navigate over the hierarchical XML structure. A simple path consists of steps separated by "/" operator, for example `a/b`, which returns all `<b>` elements that are children of any `<a>`, which is a child of the current node. It is possible to test attribute values, invoke different functions (string handling, arithmetic, date and time etc.). XPath supports also logical operators, allows to specify branches (for example `a/b[c]` will return only these `<b>` elements that have `<c>` child. It is also possible to easily obtain ancestors, descendants, siblings etc. for a given

node. XPath was defined by W3C and the version 2.0 is now recommended as a standard.

An XPath query that finds all `while` loops that assign something to variable `a` (such as the loop in Figure 2) is presented in Figure 3. It finds all `<While>` elements that have `<Block>` child that has `Assignment` descendant (but maybe not direct child) that has `<Left>` child that has `<Var>` child which has attribute `name` set to `a`.

```
//While[Block//Assignment/Left/Var/attribute::name="a"]
```

Fig. 3: XPath query example.

2.3 *CodeStatistics* in Eclipse

CodeStatistics works as an Eclipse plugin. One can generate statistics for any part of a Java project (for any subtree of the project tree in Eclipse) or print out the XML representation of the AST for a selected class.

User specifies path to his configuration file in *CodeStatistics* submenu available from *Preferences* in *Window* menu. One can choose there also output file location and the desired logging level.

3 Experiment

In our experiment we have developed a generator of termination conditions of `for` loops in Java programs. According to the approach described in this paper, we have analysed structural properties of real, productional code. We have categorized `for` loops in the following open source applications:

- **Apache Hadoop**[2] is a software platform that lets one easily write and run applications that process vast amounts of data. It implements MapReduce [13], using the Hadoop Distributed File System. Hadoop was tested on clusters with 2000 nodes.
- **Google App Engine**[3] lets you run your web applications on Google's infrastructure. App Engine applications are easy to build, easy to maintain, and easy to scale as your traffic and data storage needs grow. With App Engine, there are no servers to maintain: You just upload your application, and it's ready to serve your users.
- **JEdit**[4] is a cross platform programmer's text editor that is customizable with plugins. JEdit provides syntax highlighting for more than 130 programming languages and supports the most important character encodings.

[2] `http://hadoop.apache.org/`
[3] `http://code.google.com/appengine/`
[4] `http://www.jedit.org/`

– **Hibernate**[5] is a powerful, high performance object/relational persistence and query service. Hibernate lets you develop persistent classes following object-oriented idiom — including polymorphism, association, inheritance, composition and collections. Hibernate allows you to express queries in its own portable SQL extension (HQL), as well as in native SQL, or with an object-oriented Criteria and Example API.
– **Oracle Berkeley DB**[6] is an open source, fast, embeddable database that eliminates the overhead of SQL. It stores arbitrary key/value pairs as byte arrays. Berkeley DB can handle multiple threads or concurrent processes accessing the database.
– **Tomcat**[7] is a servlet container developed by the Apache Software Foundation. It implements the Java Servlet and Java Server Pages specifications and it provides a web server for Java code to run.

Details of these projects are presented in Figure 4.

Project	Size	No. of `for` loops
Apache Hadoop	221	1817
Google App Engine	184	1048
JEdit	108	974
Hibernate	247	720
Oracle Berkeley DB	154	1191
Tomcat	181	1452

Fig. 4: Details of analysed projects (sizes are given in Kilo Lines of Code).

Our goal was to prove automatically termination property of at least 80% of `for` loops in the code. The ideas used in this research are very basic. The main advantage of our approach is that we are showing the effectiveness of such methods in practical, large-scale applications.

Literal Let us start with the simplest type of `for` loop (Figure 5a).

```
int j = 0;
//@ decreases 5 - i;
for (int i = 0; i < 5; i += 1){
  j += i;
}
```

(a) For to a literal.

```
public final static int CONST = 5;
...
//@ decreases i - CONST;
for (int i = 15; i > CONST; i--){
  ...
}
```

(b) For to a constant.

[5] http://www.hibernate.org/
[6] http://www.oracle.com/technology/products/berkeley-db/index.html
[7] http://tomcat.apache.org/

In this case, the control variable `i` is compared to a literal (5). If `i` is not modified in the loop body, we can automatically generate the appropriate JML *decreases* formula. Using *CodeStatistics* one can find out that this kind of loops cover 2% of all `for` loops (in the set of projects used in this experiment).

Constant In the next step we focus on loops that use in the comparison a constant defined in the code (Figure 5b). As soon as the control variable `i` is not modified in the loop body, the *decreases* formula can be generated automatically. These two cases cover together 8,4% of `for` loops.

Local expression Another simple case that can be easily solved automatically is when the control variable is compared to an arithmetic expression composed of local variables that are not modified in the loop body (Figure 5).

```
int j = 0, m = 9, n = 12;
//@ decreases n + m - 7 - i;
for (int i = 0; i < n + m - 7; i++)
  j += i;
```

Fig. 5: For to a non-modified local variable.

The *decreases* formula would be $n + m - 7 - i$ (if only `i, n` and `m` were not modified in the loop body). This case covers 18,6% of all `for` loops. Together 27% are solved now.

```
int[] tab = new int[17];
//@ decreases tab.length - i;
for (int i = 0; i < tab.length; i++)
  tab[i] = i * i;
```

Fig. 6: For to a final field.

Final field Often programmers iterate over a table. This case can be generalized as iterating to a final field in an object (Figure 6). We have chosen a very restrictive "alias safe" approach. If the guard (in our simple example `tab.length`) was of form $o_1.o_2.o_3...o_n$, then

- o_1 must either be declared as final or be a local variable that is not assigned in the loop body,
- all o_2, o_3, ... o_n must be declared as final.

In this case, the well-known problem of modification using aliases is eliminated. Using *CodeStatistics* it occurs that this is a very common use of a `for` loop: it

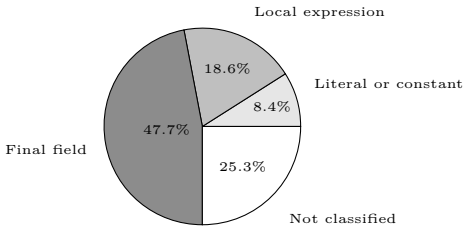


Fig. 7: Contribution of each loop type.

covers 47,7% of all cases. The above described categories cover 74,7% of all `for` in the code.

All the above described categories can be expressed by pure syntactic criteria. In order to verify that the obtained results should also apply to other projects, we have checked our method against a fresh set of applications:

- **AspectJ**[8] is a seamless aspect-oriented extension to the Java programming language that enables clean modularization of these 'crosscutting concerns'.
- **Spring**[9] is an application framework for Java platform. It includes Inversion of Control container that manages object lifecycles. Objects created by the container (Beans) are defined in XML configuration files. They can be obtained using the Dependency Injection pattern. Spring includes also Aspect-Oriented Programming framework.
- **Vuze**[10] is an application to exchange and distribute data over the Internet. In the BitTorrent protocol implemented by Vuze, the amount of downloaded and uploaded data are linked, to eliminate users that do not want to share downloaded data. Vuze is now the most popular BitTorrent client.

On these projects the recognized loop types cover 71% of `for` loops.

3.1 Bad Loops Found

Using *CodeStatistics*, we were also able to find non-trivial loops. They were sometimes written improperly, sometimes could be simplified. Our tool provides them (their code, location in the project etc.) so that a programmer can handle them manually. Using output generated by *CodeStatistics* it was very easy to select examples for this section. We have just taken random examples from the code snippets that were marked as "not classified" and were short enough to be included in this paper.

In the code fragment from Figure 8 it is not obvious why the loop should be terminating. At first it seems correct but one can notice that the inequality direction in termination condition is against intuition. The loop will terminate

[8] `http://www.eclipse.org/aspectj/`
[9] `http://www.springsource.org/`
[10] `http://www.vuze.com`

```
for (int i=nSamples - 1; i < nSamples - nRemove; i--)
  deltas.remove(i);
```

Fig. 8: Vuze: NetworkAdminSpeedTesterBTImpl.java, lines 644-645

only if `nSamples - nRemove` gets smaller than `i`, but this can be done only inside the `deltas.remove` method.

In the next example (Figure 9) the termination depends on data stored in some array (`data`). If it contained only negative numbers, the loop would not terminate (and would probably cause an `ArrayIndexOutOfBoundsException`).

```
for (int shift=0; ; shift+=15) {
  long s=data[index++];
  if (s < 0) {
    val+=(-1 - s) << shift;
  } else {
    val+=s << shift;
    break;
  }
}
```

Fig. 9: BerkeleyDB: PackedOffsets.java, lines 120-128

In the example from Figure 10 the control variable `j` is decreased in the loop body. Since in some iterations `j` does not change, and `count` is not modified in the loop body, it is not clear why should this loop terminate (in fact it terminates, because `count` is decreased inside `removeHeader(j−−)` function call).

```
for (int j=i + 1; j < count; j++)
  if (headers[j].getName().equalsIgnoreCase(name))
    removeHeader(j--);
```

Fig. 10: Tomcat: MimeHeaders.java, lines 268-272

In all examples presented in this section, it is not obvious why the loops are eventually terminating. It should be at least documented why it is working correctly. It should be explained in the loop and in the place which is responsible for the termination (e.g. in example from Figure 8, in the `deltas.remove()` function it should be marked that termination of some loop depends on changes made here to the `nremove` field). Finding such error prone code fragments is crucial also for maintenance.

4 Summary

We have presented a simple approach for creating verification methods and described the *CodeStatistics* tool that supports this methodology. We have also applied the approach in an experiment — simple loop termination prover for Java `for` loops. The received results show that basic verification techniques can be successfully used in real, productional code. Tools similar to ours can be really helpful in Quality Assurance process. Code reviewers can pay less attention to simple cases, hence they can focus on difficult aspects that are usually more error prone. Such methods can even guarantee correctness of some aspect of the code (in our case that `for` loops terminate). They can be used as a first step in introducing formal methods into business applications.

5 Future Work

We plan to employ the described way of creating verification methods, *CodeStatistics* and Abstract Interpretation techniques [11] to prove that no incorrect array accesses may occur during program execution (or find places where such errors could possibly occur).

6 Acknowledgements

This work was partly supported by Polish government grant 177/6.PR UE/2006/7 and Information Society Technologies programme of the European Commission FET project IST-2005-015905 MOBIUS. This paper reflects only authors' views and the Community is not liable for any use that may be made of the information contained therein.

References

1. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
2. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Dealing with numeric fields in termination analysis of java-like languages. In *FTfJP*, pages 77–87, 2008.
3. M. Andersen, M. Barnett, M. Fähndrich, B. Grunkemeyer, K. King, and F. Logozzo. *Code Contracts User Manual*.
4. K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.
5. J. P. Bowen and M. G. Hinchey. The use of industrial-strength formal methods. In *Proceedings of 21st International Computer Software and Application Conference (COMPSACŐ97*, pages 332–337. IEEE Computer Society Press, 1997.
6. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

7. P. Chalin and F. Rioux. Jml runtime assertion checking: Improved error reporting and efficiency using strong validity. In *FM '08: Proceedings of the 15th international symposium on Formal Methods*, pages 246–261, Berlin, Heidelberg, 2008. Springer-Verlag.
8. M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz. Propagation of JML non-null annotations in Java programs. In *PPPJ*, pages 135–140, 2006.
9. D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *In Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*. Springer–Verlag, 2004.
10. P. Cousot. Avionic software verification by abstract interpretation. In *2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation. Special Workshop Theme: Formal Methods in Avionics, Space and Transport*, Poitiers, France, December 12–14 2007.
11. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
12. D. Craigen, D. Craigen, O. Canada, O. Canada, S. Gerhart, S. Gerhart, T. Ralston, and R. H. Brown. An international survey of industrial applications of formal methods volume 2 case studies, 1993.
13. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.
14. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, Dec. 2007.
15. C. Flanagan, K. Rustan, and M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *In Proceedings of the International Symposium of Formal Methods Europe*, pages 500–517, 2001.
16. D. Harel and R. Marelly. Specifying and executing behavioral requirements: the play-in/play-out approach. *Software and System Modeling*, 2(2):82–107, 2003.
17. D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
18. G. Leavens and Y. Cheon. Design by contract with jml, 2003.
19. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual.
20. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, March 1997.
21. O. Moor, D. Sereni, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, and J. Tibble. .QL: Object-oriented queries made easy. *Generative and Transformational Techniques in Software Engineering II: International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers*, pages 78–133, 2008.
22. L. Walton. Eclipse metrics plugin: `http://metrics.sourceforge.net/`.
23. J. Yoo, S. Cha, and E. Jee. A verification framework for fbd based software in nuclear power plants. In *APSEC '08: Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*, pages 385–392, Washington, DC, USA, 2008. IEEE Computer Society.