Integrating Tools for Automatic Program Verification

Engelbert Hubbers

Nijmeegs Instituut voor Informatica en Informatiekunde,
University of Nijmegen, PO Box 9010, 6500 GL Nijmegen, The Netherlands
hubbers@cs.kun.nl

Abstract. In this paper we describe our findings after integrating several tools based upon the Java Modeling Language (JML) [1], a specification language used to annotate Java programs. The tools we consider are Daikon [2], ESC/Java [3], JML runtime assertion checker [1], and Loop/PVS tool [4]. The first one generates specifications; the others are used to verify them. We find that for the first three it is worthwhile to combine them because this is relatively easy and it improves the specifications. Combining Daikon and the Loop/PVS tool directly works in theory, but in practice it only works if the test suite is very good and hence it is not advisable.

1 Introduction

Specifying Java programs can be done by adding JML-assertions expressing preconditions, postconditions, invariants and lists of modified variables to the code. Verifying Java programs involves proving the source code correct with respect to the given specification. In this paper we present the results of an experiment where we have combined tools for both these tasks.

Note that this experiment describes some small steps toward the so-called *verifying compiler*, a compiler that guarantees program correctness before running the program, listed as one of the grand challenges of computer science [5].

1.1 Tools for Specification Generation

It is common knowledge that annotating programs with specifications or contracts increases their quality. It is also common knowledge that most programmers are not very fond of spending time on writing these specifications. Therefore some tools have been developed in order to assist programmers in writing these contracts. Houdini [6] and Daikon [7] are examples of such tools. In this paper we only use Daikon.

Daikon performs dynamic analysis. It starts with a large standard set of likely program invariants [2]. By executing test suites it deletes those invariants from the set that are falsified during the program run, leaving a set of possibly valid specifications. Some examples generated by Daikon:

```
/*@ invariant this.theArray != null; */
/*@ requires this.topOfStack < this.theArray.length-1; */
/*@ modifies this.theArray[*], this.topOfStack; */
/*@ ensures (\old(this.topOfStack) >= 0)  ==>
          (\old(this.topOfStack) == this.topOfStack + 1); */
```

Obviously this dependence on a specific test suite makes the tool unsound. Therefore we need a different tool in order to check the outcome generated by Daikon.

1.2 Tools for Proving Contracts

Several tools are available for checking Daikon's outcome when it is applied to Java programs: ESC/Java, JML , JACK [8], CHASE [9] and the Loop/PVS tool. Each of them has its special characteristics. In our experiment we use ESC/Java, JML and Loop/PVS. To avoid confusion between the language and the tool JML, we will use JMLRAC to refer to the tool.

1.3 The Specification Language JML

The binding factor between the tools we use is the specification language supported by all of them: JML, the Java Modeling Language. It can be used to specify the behavior of Java modules. It includes class invariants, constraints, method preconditions, and both normal and exceptional postconditions. The clauses are written between the code using special comment tags. Hence normal Java compilers do not notice these JML annotations. Goal of the JML project is to provide a language that resembles the Java programming language very closely, hereby making it easier to use for the actual Java programmers. Since JML is still under development it is almost impossible for the tools to keep up with full JML. Hence most tools use their own dialect of JML. The Java/JML combination has a great resemblance to the Eiffel 'design by contract language' [10]. However, because JML supports quantifiers such as `\forall` and `\exists`, and because JML allows so-called model fields, JML specifications can be more precise and complete than those typically given in Eiffel.

1.4 Related Work

This is not the first paper on integrating tools for specification generation and tools for specification verification. The papers [11,12] describe results of the integration of Daikon and ESC/Java. And the paper [13] describes an experiment of combining Daikon with the Larch Prover [14]. However, the combination of Daikon with the Loop/PVS tool and JMLRAC has not been described before.

In this paper we only look at Java and JML tools. However, tools for other languages also exist. See for instance Splint [15]. Using static analysis this tool checks whether a C program operates according to its user written annotations. And the SLAM/SLIC combination [16] resembles our experiment even better. It is a Microsoft tool that checks safety properties of C programs without the need for users to write annotations by hand because it generates them automatically.

In Sect. 2 we describe the tools used in more detail. In Sect. 3 we describe the experiment itself. In Sect. 4 we discuss the relevance of our findings.

2 Tools

2.1 Daikon

Daikon is developed at MIT. The tool dynamically generates specifications for input source code. We use Java source, but Daikon also supports C and Perl. Given this source file and a test suite it generates a file containing the original code annotated with JML assertions. These assertions are written in ESC-JML, the dialect of JML that is used by ESC/Java.

2.2 ESC/Java

ESC/Java stands for Extended Static Checker for Java. It is developed at Compaq Research. ESC/Java tries to prove complete programs correct with respect to their specification in ESC-JML using the theorem prover Simplify as engine. It is neither sound nor complete. However, this has been a design issue: one rather has a tool that is very easy to use and good at detecting certain type of errors than a sound and complete tool which is difficult to use. In particular ESC/Java is good in finding programming errors at compile time which are usually not detected before run time; for example null dereference errors, array bounds errors, type cast errors, and race conditions. Note that ESC/Java does not require any user interaction: once started it runs completely automatic.

2.3 JMLRAC

JMLRAC is being developed primarily at Iowa State University. It compiles JML assertions into an executable with runtime checks. If during execution these assertions do not hold, the program stops and reports a violation of the corresponding assertion. Like ESC/Java, JMLRAC doesn't require user interaction.

We have seen before that ESC/Java uses the so-called ESC-JML dialect of JML. JMLRAC uses the full JML syntax. Unfortunately there are some differences[1] between ESC-JML and JML as it is currently supported by the tools. However, JML is nearly a superset of ESC-JML, hence a program annotated in ESC-JML should almost always run without problems in JMLRAC.

It may seem strange to use a runtime assertion checker like JMLRAC to test whether specifications generated by Daikon are correct or not. If this checker is used on the same test suite as the specification generator Daikon, it is to be

[1] See [17, Ref. Manual, Sect. 16]. At the moment Kiniry and Cok have almost finished their project to adapt ESC/Java to full JML syntax. Their results will be made available on [4].

expected that no line generated by Daikon will lead to a violation of the runtime checks in JMLRAC. So if we do find these violations this might indicate that there is a problem with one of the two tools.

2.4 Loop/PVS

Loop stands for Logic of Object-Oriented Programming. It is the name of a research project carried out by our own Security of Systems group at the University of Nijmegen [4], and also the name of the tool that plays a central role in this project. The topic is formal methods for object-oriented languages. The aim is to specify and verify properties of classes in object-oriented languages, using proof tools like PVS [18].

The Loop project is centered around the Loop tool. Basically this is a compiler. The input for this compiler consists of Java source code annotated with JML specifications. The output are logical theories which can be loaded into the theorem prover PVS. If one succeeds in proving the corresponding theorems, one knows for sure that the JML specifications really match the code. Intrinsically, this process of proving requires a lot of user interaction and is very tedious work, which is a big contrast with ESC/Java and JMLRAC. Therefore an important part of our research involves writing sophisticated proof strategies that can automate the process as much as possible.

The Loop tool is not publicly available. Furthermore, it uses its own JML dialect. We will refer to it as Loop-JML. Although the Loop compiler and PVS are separate tools, we will use the term Loop/PVS as if it is one tool.

Note also that the goal of Loop/PVS is more ambitious than the goals of ESC/Java and JMLRAC. These two mainly serve the purpose to reduce the number of errors very easily, whereas Loop/PVS wants to show the absence of all errors at the cost of hard work.

3 Experiment

3.1 Test Suites

In order to stay close to the experiments already done by Nimmer and Ernst, we have used the same examples, originating from a textbook by Weiss [19]: `DisjSets`, `QueueAr` and `StackAr`. We have also added an example that comes from the tutorial for Loop/PVS [20]: `Tutorial`.

Weiss's examples come with minor test programs. The `QueueAr` and `StackAr` also come with the Daikon distribution where they have larger test suites. We used both of these suites. In the table of results we have made this distinction visible with a (W) for Weiss and (D) for Daikon. The `Tutorial` didn't have any test suite at all, so we made one ourselves.

3.2 Setup

The setup of this experiment is shown in Fig. 1. The dashed arrows indicate the desired situation where the code is proved correct with respect to the given specification. Both ESC/Java and JMLRAC report exactly which assertion is being

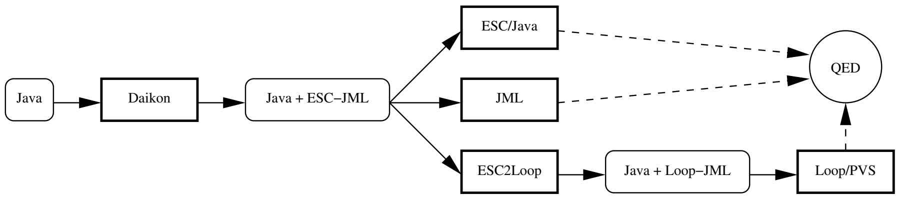

Fig. 1. From Java source file to a proof

violated. Loop/PVS combines all assertions and tries to prove them together. Hence it is more difficult to point out which annotation caused the problem if the whole set of assertions couldn't be proved in Loop/PVS. And therefore we split the experiment into two sub-experiments.

3.3 Daikon, ESC/Java, and JMLRAC Results

The results are related to the number of Daikon's annotations in Fig. 2. The figures in the Daikon columns reflect the number of `invariant`, `requires` and `ensures` annotations created by Daikon. The figures in the ESC/Java columns are the numbers of annotations that could not be verified by ESC/Java. And the figures in the JMLRAC columns are the numbers of annotations for which JMLRAC detected an assertion violation.

	Daikon (generated)			ESC/Java (rejected)			JMLRAC (rejected)		
	`inv`	`req`	`ens`	`inv`	`req`	`ens`	`inv`	`req`	`ens`
`DisjSets`	7	16	29	$2^{\star}$	1	0	0	0	0
`QueueAr` (W)	9	39	74	$2^{\star}$	$6^{\star}$	$8^{\star}$	0	1	2
`QueueAr` (D)	9	25	49	0	1	4	0	0	1
`StackAr` (W)	8	3	20	$4^{\star}$	0	$4^{\star}$	0	0	1
`StackAr` (D)	7	4	36	1	1	1	0	0	1
`Tutorial`	0	20	26	0	1	7	0	0	1

Fig. 2. Daikon, ESC/Java and JMLRAC results. Entries with a* include unverified clauses because of type check errors

Although in principle ESC/Java is able to read Daikon's output, we did encounter 'type check errors' running ESC/Java on all three examples with Weiss's test suites. By removing the corresponding assertions and counting them as unverified and running ESC/Java again, we came up with the figures in Fig. 2.

A similar strategy is used for JMLRAC. The checker typically only mentions the first assertion violation. By removing this assertion and running JMLRAC again, we filled the whole table.

The problems we found here can be seen as interpretation differences between the tools or as bugs. We would like to classify the first four problems as interpretation differences, the last one as a bug.

1. ESC/Java gives a type check error on Daikon's `this != null` invariant.
2. ESC/Java gives a type check error on Daikon's `requires` and `ensures` clauses with the construct `\elemtype(this.theArray)`.
3. JMLRAC detects postcondition violations on Daikon's postconditions with the construct `\typeof(\result) == \typeof(null)`. JML's `\typeof` refers to the dynamical type. But what is the dynamical type of `null`? And what is Daikon's interpretation of `\typeof`?
4. ESC/Java complains about the fact that Daikon uses `\old` on variables that are not mentioned in the `modifies` clause in its `ensures` clauses. In JML `\old(v)` refers to the value of `v` in the pre state. If `v` is not modified this implies that `\old(v)==v` in the post state and therefore it is usually written as `v`. However `\old(v)` is correct JML, so why is ESC/Java cautious about this? And what is Daikon's interpretation of `\old`?
5. JMLRAC reports a postcondition violation on `\result == this.i` executing the `Tutorial` example. The code causing this problem is:

```
int tryfinally() {
    try { switch (i) { case 0:  throw (new Exception());
                         default: return i; }
        }
    catch(MyException e) { return --i; }
    catch(Exception e) { return ++i; }
    finally { i += 10;
              j += 100; }
}
```

   Apparently Daikon gets confused because of the tricky `try-catch-finally` construction. It seems that Daikon only regards the `return` statements as exit points for the method and doesn't take into account that the finally part modifies some values afterwards.

3.4 Daikon and Loop Results

Because Loop-JML is not a superset of ESC-JML we use the script ESC2Loop to transform Daikon's output to a format that can be parsed by Loop/PVS. This transformation involves both pure syntax modifications as well as deletions of constructs that are not supported in Loop-JML.

The real work starts in PVS. Every method in the Loop-JML file has been translated to a lemma. If such a lemma is proved, one knows that the complete specification for this method is correct. If one doesn't succeed in proving this, it takes some experience to see which line of the specification is really causing the problem. Hence there is no easy reference back to the original line in the specification generated by Daikon.

We managed to prove the specifications for some of the methods. Unfortunately most method specifications lead to dead ends: situations where we don't know how to complete the proof. Sometimes this was because we could see that the specifications were incorrect. ESC/Java can help here: if it cannot establish the condition, it sometimes gives enough hints to construct a counterexample. However, more often this was because we got stuck in PVS technicalities and the complexity of the underlying formalism for the memory model as it is being used by Loop/PVS.

4 Conclusion

The original goal of this experiment was to test whether Daikon's results could be verified by other tools. We noticed that both ESC/Java and JMLRAC are well suited for this task. Mainly because they run without user interaction. The combination of Daikon and Loop/PVS is less suitable for this task. PVS is good at noticing that a proof has been completed. However, it is not good at detecting branches of proofs that will lead to contradictions. If one also takes into account that most proofs require a lot of work, it seems best not to feed Daikon's output directly to Loop/PVS. Note that this is in line with the earlier claim that Loop/PVS is more ambitious but requires more effort: if you decide to do the hard work with Loop/PVS then you should also invest more time in writing the specifications and not rely on a specification generation tool only. But if you do want to use a tool like Daikon, make sure that you also use a tool like ESC/Java to filter out the 'obvious' problems before feeding the specifications to Loop/PVS to formally prove their correctness.

During the experiment we found that combining tools also serves a second goal: to detect problems with the tools. As we have seen before, these problems can be real bugs or just differences of interpretation of the specification language JML. The fact that we got type check errors for Weiss's test suites is a good indication that there is a problem between Daikon's ESC-JML and ESC/Java's. And the fact that JMLRAC reports a postcondition violation although it is run on the same test suite as Daikon indicates that there must be something wrong with one of the programs. Furthermore, running Loop/PVS on Daikon's output revealed quite a few bugs in the parser of the Loop compiler. These bugs showed up because Daikon sometimes uses constructs that are valid JML, but would never have been written like this by a human. And hence Daikon's automatic output is interesting test input.

Finally, a tool for automatic derivation of test suites would be great. However, creating it might be an even greater challenge than Hoare's verifying compiler.

References

1. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06t, Iowa State University, Dep. of Computer Science (2002) See `www.jmlspecs.org`.
2. Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically Discovering Likely Program Invariants to Support Program Evolution. IEEE Trans. on Software Eng. **27** (2001) 99–123
3. Compaq Systems Research Center: Extended Static Checker for Java (2001) version 1.2.4, `http://research.compaq.com/SRC/esc/`.
4. Nijmeegs Instituut voor Informatica en Informatiekunde, University of Nijmegen, The Netherlands: Security of Systems group (2003) `http://www.cs.kun.nl/ita/research/projects/loop/`.
5. Hoare, T.: The verifying compiler: a grand challenge for computing research. Invited talk for PSI'03 conference (2003)
6. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for ESC/Java. In: Formal Methods Europe. Volume 2021 of LNCS., Berlin, Germany (2001) 500–517
7. Lab. for Computer Science, MIT: Daikon Invariant Detector (2003) version 2.4.2, `http://pag.lcs.mit.edu/daikon/`.
8. Lilian Burdy and Jean-Louis Lanet and Antoine Requet: JACK (Java Applet Correctness Kit) (2002) `http://www.gemplus.com/smart/r_d/trends/jack.html`.
9. Cataño, N., Huisman, M.: Chase: A Static checker for JML's Assignable Clause. In: Verification, Model Checking and Abstract Interpretation (VMCAI'03). Number 2575 in LNCS, Springer-Verlag (2003) 26–40
10. Meyer, B.: Eiffel: the language. Prentice Hall, New York, NY (1992)
11. Nimmer, J.W., Ernst, M.D.: Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In Havelund, K., Rosu, G., eds.: Electronic Notes in Theoretical Computer Science. Volume 55., Elsevier Science Publishers (2001)
12. Nimmer, J.W., Ernst, M.D.: Automatic generation of program specifications. In: ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis, Rome, Italy (2002) 232–242
13. Win, T.N., Ernst, M.D.: Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, MIT Lab. for Computer Science (2002)
14. Garland, S.J., Guttag, J.V.: A guide to LP, the Larch Prover. Technical Report 82, Digital Equipment Corporation, Systems Research Center (1991)
15. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. IEEE Software **19** (2002) 42–51
16. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. Lecture Notes in Computer Science **2057** (2001) 103–122
17. Gary Leavens et al.: Java Modeling Language (JML) (2003) `http://www.jmlspecs.org/`.
18. Computer Science Lab., SRI: The PVS Specification and Verification System (2002) version 2.4.1, `http://pvs.csl.sri.com`.
19. Weiss, M.A.: Data Structures and Algorithm Analysis in Java. Addison Wesley (1999) ISBN: 0-201-35754-2.
20. Jacobs, B.P.: A Tutorial for the Logic of JML in PVS. under development (2002)