Teaching program specification and verification using JML and ESC/Java2

Erik Poll

Radboud University, Nijmegen, The Netherlands

Abstract. The paper summarises our experiences teaching formal program specification and verification using the specification language JML and the automated program verification tool ESC/Java2. This technology has proven to be mature and simple enough to introduce students to formal methods, even undergraduate students with no prior knowledge of formal methods and even only very basic knowledge of (Java) programming. However, there are some limitations on the kind of examples that can be comfortably tackled.

1 Introduction

Over the past years we have taught formal program specification and verification using the JML specification language for Java and the automated program verification tool ESC/Java2 to a variety of audiences. We have taught this as a small module as part of larger courses. The module consists of a 2 hour lecture to introduce the basic concepts and notations, and an afternoon exercise lab. The set-up of the practical work is that students annotate example code with JML contracts – expressing preconditions, object invariants, and to a lesser extent postconditions – in response to feedback from the tool.

We have given such classes to students taking a course on formal semantics and program logics (so that they know what Hoare triples and weakest preconditions are), but most classes have been given to students without any exposure to formal methods apart from basic propositional logic. We have also taught the module to Information Science[1] students who only have very basic knowledge of programming.

The outline of the rest of the paper is as follows. Section 2 discusses the motivation and aims for the course module. Sections 3 and 4 discuss JML and ESC/Java2, respectively. Section 5 explains the set-up of the exercise classes and Section 6 discusses the pitfalls and limitations in letting students work with ESC/Java2. Section 7 gives pointers to our course material and discusses some related possibilities. Finally, sections 8 and 9 evaluate and conclude.

2 Context and goals

Our original motivation for the module was that in an existing course on semantics and logics students only experienced techniques such as Hoare-logic

[1] In Dutch: Informatiekunde

and weakest-precondition calculus as paper-and-pencil exercises. We thought it would useful if students experienced the possibilities of such techniques in programming tools, not just to show the capabilities of such tools, but also to make the connection with programming as they know it, in normal Java rather than some toy imperative programming language.

As it became apparent that students did not really need much theoretical background to do practical exercises using ESC/Java2, and that students found them interesting, we reused the idea in other settings, for instance to make information science students appreciate the importance of documenting assumptions and constraints as part of specifications.

The aims of the course are

- to make students aware of the hidden assumptions and implicit constraints and design decisions there are in typical programs, or indeed in specifications;
- to teach them how such assumptions and constraints can be documented in contracts, esp. with preconditions and invariants, using JML;
- to let them experience the added value of doing this in a formal language amenable to tool support, namely that they can run the program checker ESC/Java2;
- for students with knowledge of program logics such as Hoare-logic and wp-calculi: to let them experience what using such techniques in practice can be like.

To achieve these aims, the exercises are designed to include implicit assumptions that are so obvious that they are easy to overlook (e.g. in the example in Fig. 2) and properties where the precision of a more formal language than English (or Dutch) is really useful (e.g. in the example in Fig. 1).

3 JML

JML is a specification language tailored to Java. It allows specifications to be added to Java code, as special comments after `//@` or between `/*@ ... @*/`, in the Design-by-Contract style of Eiffel [17]. The core constructs of JML are preconditions, postconditions and object invariants[2]. JML offers a large range of additional constructs, but these are typically best avoided by a novice user. The initiative to develop JML was taken by Gary Leavens [14], but it has grown into a wide collaboration, with many people contributing to the language definition and using it as specification language in tools. Many program analysis tools for Java support JML in one form or another. The original use of JML was for runtime assertion checking [8], but it has also been used by program verification tools, for instance ESC/Java(2) [11,12], JACK [6], KeY [1], Krakatoa [16], and LOOP [4], and the Java model checker Bogor [19]. For an – already somewhat outdated – overview of JML and JML tool support see [5]. More on ESC/Java(2) below in Section 4.

[2] Object invariants are sometimes called class invariants, but in our opinion this is confusing terminology.

Experiences with JML One of the main design goals of JML is that it should be easy to understand and use for any Java programmer. Although the more complicated constructs of the language are certainly not suitable for the average Java programmer (their precise semantics can still lead to heated debates between experts on the JML mailing list), for the basic JML constructs, such as pre- and postconditions and object invariants, this is in our experience certainly the case. After a short explanation of these notions in a lecture all students can cope with this. Only a minimal amount of syntax needs to be learned, namely just the keywords `requires`, `ensures`, and `invariant`, and the syntax for implication `==>`, bi-implication `<==>`, and universal quantification `\forall`. (To prove the point, we will use JML syntax in the remainder of this paper without any further introduction.)

Only the notion of object invariant requires some attention. Courses on program verification typically include *loop* invariants, but not *object* invariants. (In the practice of writing modern OO code, the notion of object invariant may well be more relevant for students to know!) Intuitively, the notion of object invariant can be explained as being implicitly included in the pre- and postconditions of all methods, and in the postconditions of all constructors. However, one should be aware that this is oversimplifying things, and cutting some corners! Precisely defining the semantics of the apparently simple notion of object invariant is notoriously complicated in the presence of call-backs, dynamic binding, subclassing, and aliasing. This might be an interesting topic to explore in a more advanced course on formal methods, but is best avoided in a first introduction to the notion of Design-by-Contract. More about potential hassle with invariants later.

During exercise classes we notice that many students need a hint before they realise that a precondition that they keep repeating needs to be turned into an invariant. This is in part caused by the fact that the work is tool-driven, and ESC/Java2 will complain about missing preconditions, but not about missing invariants. It is good to point out that for nearly every field in a class there is an associated object invariant, even if it is just saying that some reference field `t` is never null,

```
//@ invariant t != null;
```

or some integer field `i` that is always non-negative,

```
//@ invariant i >= 0;
```

JML includes the concept of *exceptional* postconditions, aka `signals`-clauses, which express the postcondition that holds in case an exception (or an exception of a certain type) is thrown. In our experience, this notion is best avoided. It is very easy to get confused between specifying when an exception *may* be thrown and when it *must* be thrown. Our exercises, and indeed the basic setup of ESC/Java2, are geared to proving the absence of all runtime exceptions as a first step (and possibly only step!) in the verification. In our experience just proving this can expose plenty of implicit design decisions.

JML includes the possibility to express frame conditions by so-called *assignable clauses* (aka modifies clauses). While frame conditions are an interesting concept, and crucial to the verification of imperative programs, it is best omitted for a first introduction to formal program verification. (By the way, the notions of object invariant and frame condition are the most important notions missing in traditional approaches to program verification, which just consider pre- and postconditions and loop invariants.)

Finally, we noticed that some students would include some superfluous universal quantifications in object invariants. For example, the invariant about the field `i` above might be written as

```
//@ invariant (\forall SomeClass s; s.i >= 0);
```

where `SomeClass` is the class where the invariant occurs. This is superfluous because object invariants specified for `this` are already implicitly quantified over all objects of the current class. An invariant with such a universal quantification is not only bad style, but it also causes complications in automated verification, as the use of universal quantification is a major bottleneck for automated theorem provers. More on this issue below in our discussion of the experiences with ESC/Java2 exercises in Section 5.

4 ESC/Java(2)

ESC/Java is a program verification tool developed at Compaq (formerly DEC, and subsequently HP) by Rustan Leino and his co-workers [11]. After the disbanding of that research group at Compaq, David Cok and Joe Kiniry have led valiant efforts to keep ESC/Java alive and further improve it, resulting in what is now called ESC/Java2 [12]. In the meantime, Rustan Leino has gone on to develop the Spec# specification language for C# and the associated Boogie verification tool at Microsoft research labs [3][3].

ESC stand for Extended Static Checker. The name was chosen to stress that using the tool is intended to be similar in experience to using an automated, push-button static analysis tool, or a type checker. The tool is geared to a 'lightweight' form of program verification, i.e. verifying relatively simple properties of code rather than detailed functional specifications. (Indeed, this limitation is one of the possible pitfalls we discuss later.) Still, the tool does program verification in the classical way, using weakest precondition generation[4] to produce verification conditions that are fed to an automated theorem prover, Simplify [10]. The users do not see the back-end theorem prover or the verification conditions that are generated, but get feedback about violated invariants, violated pre- or postconditions or unexpected runtime exceptions in specific execution paths.

The builders of ESC/Java have been keen to point out that their tool is neither sound nor complete, but aims to spot as many potential bugs with the

[3] See `http://research.microsoft.com/specsharp`

[4] Or strongest postcondition generation, but the user does not even see the difference.

minimum of effort. While this has proven to be a successful design decision, and it is a nice bold statement to challenge some fundamentalist doctrine about formal methods, the disclaimer that the tool is not sound can give the wrong impression. In our experience, ESC/Java is a lot "more sound" than some verification tools that do not make such explicit disclaimers. In particular, ESC/Java takes a rigorous (albeit not completely sound) approach to ensuring that object invariants are not broken, where it makes worst case assumptions about potential aliasing. Note that traditional Hoare logics or weakest precondition calculi do not take into account the notion of object invariant (or indeed aliasing), thus ducking this major complication in program verification.

5 Simple exercises using JML and ESC/Java2

For practical exercises we provide students with example programs that they have to annotate with JML and for which they possibly have to correct bugs in the Java code. They have to add specifications, either in response to warnings by ESC/Java2 (initially about unexpected runtime exceptions, later also about broken invariants and preconditions), or to formally express informal specifications that are given to them.

We do these exercises in a terminal room where there is some help around to answer questions and sometimes point students in the right direction. We have typically had 20 to 30 students doing the exercises, with initially three people around to help, but once the initial peak of questions and technical hassle in getting things to work has passed two people can cope easily. Plenty of students will manage to do the exercises on their own machine without any help.

Experiences using ESC/Java2

It is nice to see students realising the added-value of formal specifications, in that tools can help them to spot bugs, including some subtle bugs that are very easy to overlook. Also, it brings home the message about the importance of making implicit assumptions and design decisions explicit, not just for tools to make sense of code, but also for humans to understand the code. Of course, in standard programming courses students will be taught to document design decisions, as informal comments in code, but then there is typically no tool actually using these comments, so that writing the documentation is only extra work without any immediate benefit.

Of course, the exercises are designed to illustrate that knowing which object invariants hold is useful, if not crucial, to de-bug code. In all honesty, some parts of our sample programs have been very carefully crafted to contain subtle bugs that formal specifications will reveal.

There are some simple practical tips that can help the students. Splitting large invariants with conjunctions into smaller ones will improve feedback from the tool. Adding `assert` annotations to the code – to find out what holds or

```
/* Objects of this class represent euro amounts. For example, an Amount
   object with euros == 1 cents == 55 represents 1.55 euro.

   1) We do not want to represent 1.55 euro as an object with
       euros == 0, cents == 155
      Specify an invariant that rules this out.

   2) We do not want to represent 1.55 euro as an object with
         euros == 2, cents == -45
      Specify one (or more) invariant(s) that rule this out.
  */

public class Amount{

 private int cents, euros;

 public Amount(int euros, int cents){
   this.euros = euros;
   this.cents = cents;
 }

 public Amount negate(){
   return new Amount(-cents,-euros);
 }

 public Amount add(Amount a){
   int new_euros = euros + a.euros;
   int new_cents = cents + a.cents;
     if (new_cents < -100) {
      new_cents = new_cents + 100;
      new_euros = new_euros - 1;
   }
   if (new_cents > 100) {
      new_cents = new_cents - 100;
      new_euros = new_euros - 1;
   }
   if (new_cents < 0 && new_euros > 0) {
       new_cents = new_cents + 100;
       new_euros = new_euros - 1;
   }
   if (new_cents >= 0 && new_euros <= 0) {
       new_cents = new_cents - 100;
       new_euros = new_euros + 1;
   }
   return new Amount(new_euros,new_cents);
 }
}
```

Fig. 1. Example exercise, with a non-trivial invariant to specify

```
public class Taxpayer {
  boolean isFemale, isMale, isMarried;
  Taxpayer father, mother, spouse;
  //@ invariant isMarried ==> spouse.spouse == this;
  int age, tax_allowance;

  //@ requires newSpouse != null;
  public void marry(Taxpayer newSpouse) {
    spouse = newSpouse;
   isMarried = true;
  }

  public void divorce() {
    spouse.spouse = null;
    spouse = null;
    isMarried = false;
  }
```

Fig. 2. Fragment of an example exercise, with an initial attempt at capturing some of the (many!) implicit invariants involved and preconditions needed to ensure that none of these are broken.

does not hold at a particular program point – is a useful way to figure out why something fails to verify.

Something that we did not anticipate was that for some students ESC/Java2 is their first experience of using an automated theorem prover. Simplify, the back-end theorem prover of ESC/Java2, is quite good at propositional logic, so it is an eye-opener for some students that they can use the tool to spot some less obvious consequences of their specifications, for instance when they are struggling with subtly different formulations of object invariants for the example in Figure 1. For example, after specifying (incorrectly, by the way)

```
//@ invariant euros > 0 ==> cents > 0;
//@ invariant euros < 0 ==> cents < 0;
```

ESC/Java2 will point out errors in claims such as

```
somemethod(){
 //@ assert !(euros <= 0 & cents => 0);
 ..
}
```

at particular program points.

A danger when using an automated program verification tool like ESC/Java2 is that students end up 'mindlessly' trying out specifications to stop the tool from complaining, without really thinking about the meaning of the specifications they write. It is useful to ask them questions to reflect on what they are doing and

why. Having them work in pairs and discuss with others helps here. It is also useful to let students examine the code, and make them think about possible object invariants, *before* letting them use the tool.

One cause for confusion for students is that they initially do not realise that program verification is done in a modular fashion, per-method or per-constructor. If method `m()` calls method `n()`, then when the tool verifies `m()` it will only use the contract for method `n()`, and not look at its code. This means that the tool will complain about programs that are – in the eyes of the student – obviously correct, because they know the code of `n()`.

There is a deeper reason for this modularity, namely that method n may be overridden in a subclass. JML enforces the notion of behavioural subtyping, which says that any methods overridden in a subclass have to satisfy any contracts written in the parent class.

The same issue of modularity can also cause some confusion with constructors and invariants. For example, students are typically surprised that the tool complains about an integer field `n` potentially being negative, even if all constructors obviously initialise `n` to a non-negative value, and no code anywhere assigns negative values to `n`. In such cases the implicit invariant needs to be made explicit, by adding

```
//@ invariant n >= 0;
```

to the code.

The same deeper reason for this applies, of course: there could be constructors in subclasses that fail to establish the property, or methods that break it. Or, simpler still, someone making changes to the code of the original class could unwittingly break such representation invariants. Of course, the whole point of explicitly documenting design decisions – such as which invariants are supposed to hold – is to avoid such problems.

6 Limitations and pitfalls in the use of ESC/Java2

It would of course be nice to move to more ambitious programs for students to specify and verify, and also programs they write themselves, rather than programs that are given to them. However, there are some practical limitations to be aware of:

1. Firstly, there is the limited power of the back-end automated theorem prover. If specifications become too expressive, the verification conditions may be too complicated for the theorem prover.
   This problem typically surfaces when people try to write detailed function specifications that involve universal quantifiers. For example, attempts to verify the full functional correctness of some sorting algorithm – one of the standard examples in traditional course material on program verification –

are unlikely to be successful[5]. Here the fact that the user doesn't get to see the back-end theorem prover becomes something of a disadvantage. The user notices that the tool cannot prove a specification, but cannot see where it goes wrong or where it misses some additional information. (By the way, exercises where there are some relatively basic properties to specify, such as object invariants, are in our opinion more realistic than the examples involving full-blown functional specifications that traditionally feature in course material on program verification! For larger programs functional specification quickly becomes infeasible, but specifying more basic properties, such as object invariants, remains feasible and interesting.)

2. A second limitation is the need for API specifications. To verify a piece of code one will need formal specifications of any API methods it uses. There have been some collective efforts to write JML specifications for parts of the Java API (see `http://www.jmlspecs.org`), but these only cover small parts of the API. Moreover, not only the *absence* of API specifications can be a problem, but also their *presence* can be: if the specifications are too expressive, one runs into the first limitation mentioned above.
3. A third limitation is a more fundamental complication for program verification of object-oriented programs, or indeed any imperative programs: *aliasing*, especially the way it interacts with the meaning of object invariants.
   Intuitively, during the execution of a method the invariant of the current object may temporarily be broken, as long as it is re-established at the end. However, an additional complication is that invoking a method on one object may break the invariant of another object. This may happen if the invariant of the one object refers to field of the other object, or if the object have fields that could potentially be aliased. ESC/Java2 is quite good (or, a less positive way of phrasing it, extremely paranoid) when it come to spotting potential trouble caused by aliasing. If two different objects have fields of compatible types, ESC/Java2 will consider the (worst case) scenario that these objects may be aliased, unless specifications explicitly rule this out.
   This is probably the major source of confusing error messages that ESC/Java2 may report to the unsuspecting user. There are ways to solve these issues, but simply spotting the source of the problem can be tricky.
4. Finally, there are limitations to the Java features that ESC/Java2 can handle. Most importantly, it does not support generics. The ongoing evolution of a programming language such as Java poses a serious challenge to the development of tool support, despite ongoing initiatives such as JML4 [7].

We have given one course where students did use ESC/Java2 on code that they wrote themselves from scratch. In that course students wrote Java Card

[5] In private communication, Cormac Flanagan has reported good experiences with letting students write an implementation of quicksort for which they have to check a partial specification, which only states that the result array is sorted, not that it is a permutation of the input array; this is still simple enough to avoid running into this problem.

code, for execution on smartcards. For Java Card applications the pitfalls mentioned above can be avoided:

1. By instructing students to specify only very simple properties (basically, absence of runtime exceptions), the first pitfall can be avoided.
2. Because Java Card provides only a very limited API, and there are good specifications for this entire API [18], the second pitfall mentioned above can be avoided.
3. Because Java Card programs are not very object-oriented – most objects are just arrays – problems with aliasing are relative easy to control.
4. Finally, there are no generics in Java Card.

Not surprisingly, many program verification tools for Java have focused on Java Card as an interesting application area, e.g. [6,16,1]. The fact that smartcard code is hard to debug – in the absence of a screen, you cannot debug by adding println's to the code – was a nice additional motivation for students to verify the code. Still, writing Java Card code is something of an obscure specialism, and installing this software on smartcards requires special skills, so this idea is not easy to re-use by others.

7 Pointers and related tools

All material we use is available on-line[6]. Much more teaching material using JML is available via the JML website[7]. There is a mailing list[8] to get help from experienced ESC/Java2 users, should that be necessary.

For years we have used the stand-alone version of ESC/Java2[9] which can be run from the Windows, Linux, UNIX, or MacOS command line, and proved easy to install. There are now also stable versions of ESC/Java2 available as Eclipse plugin, in the form of the Mobius Program Verification Environment[10], which is also integrated with the JML4 initiative [7]. Beware that some of the earlier attempts at Eclipse plug-ins for ESC/Java2 might not be easy to install.

A more ambitious course that aims at a thorough integration of JML (as well as BON) into a software engineering course, has been developed by Joe Kiniry and Daniel Zimmerman [13].

There are other program verification tools that one could use for exercise courses, notably Spec#/Boogie [3] for C#, or KeY [1] or Krakatoa [16] for Java. For Spec# there have been efforts to improve the handling of set comprehensions such as sum, min, and max to make the Spec# program verifier capable of verifying standard textbook examples fully automatically [15]. KeY and Krakatoa can expose more of their internal working, which may be useful as part of

[6] From `http://www.cs.ru.nl/~erikpoll/Teaching/JML`
[7] At `http://www.jmlspecs.org/teaching.shtml`
[8] https://lists.sourceforge.net/lists/listinfo/jmlspecs-escjava
[9] Available from `http://kind.ucd.ie/products/opensource/ESCJava2`
[10] Available from `http://kind.ucd.ie/products/opensource/Mobius`

a course on say Hoare logics, where would want to show the internal workings, and not just use the program verifier as a black box. The KeY tool supports a simple while-language that could be used for this purpose [2]. Teaching material for the KeY tool is available from `http://www.key-project.org/teaching`.

Instead of program verification there is also the possibility to let students use a runtime assertion checker for JML, for instance using the JMLUnit combination of JML runtime assertion checking with JUnit [9]. Beware that the collection of JML tools available of the web, including JML2, JML4, JML5, and OpenJML, can be a bit bewildering.

8 Evaluation

This JML and ESC/Java2 module has only been a small part in larger courses (in the order of one week in a 14 week term). Since course evaluations (via web questionnaires) are done for a course as a whole, these do not provide a lot of detailed information about this particular module. Still, students do regularly mention it as the most interesting part of the course in the section for open comments on the evaluation form. This confirms a lot of positive feedback we get from students in class.

The experience helping out during the exercise classes does confirm that the messages that the course tries to make come across. But apart from doing the exercises, there is no additional exam that would allow a more impartial assessment if the course meets its aims.

The main difference we noticed between information science students and those having some background in formal methods is that the latter are much more at ease with using propositional logic.

9 Conclusions

The good news is that program verification technology in tools such as ESC/Java2 is mature enough for *any* student to use – even first year undergraduates, and even as part of courses which are *not* specifically about formal methods, such as standard programming or software engineering courses. It is straightforward to explain students what they need to know in a two hour lecture and then let them play with the tool in a practical session for a couple of hours. It seems a missed opportunity if not all computer science students experience using such a program verification tool. The theory of program verification might be relegated to more specialised (Master) courses that not all students take, but the use of verification tools should not be.

The bad news is that the use of the tool is best limited to controlled experiments, where the students work with (essentially toy) programs that are supplied, rather than code they develop themselves, to avoid running into the problems mentioned in Section 6. Moreover, traditional program verification exercises, that involve detailed and complete functional specifications are best

avoided, as explained in Section 6, though these might be feasible using Spec#, as discussed in Section 7.

An important positive aspect of using JML is that students see that formal program specification and verification can be applied to a real programming language, Java, rather than some toy while-language or guarded command language. Another positive aspect of getting students to use tools is that they experience the potential added value of writing formal specifications, namely that tools can them help them to identify bugs and expose implicit hidden assumptions. We do not think it is a good idea to let students write formal specifications *without* ever letting them experience using a tool that shows them what the potential benefits might be. Most importantly perhaps, most students seem to enjoy playing with ESC/Java2.

Acknowledgements Credit goes to the many people have contributed to the development of ESC/Java(2) over the years. ESC/Java was designed by Rustan Leino and Jim Saxe, and developed with help from Cormac Flanagan, Rajeev Joshi, Mark Lillibridge, Todd Millstein, Greg Nelson, Raymie Stata, and Caroline Tice. The ESC/Java2 initiative has been led by Joe Kiniry and David Cok, and includes contributions from Patrice Chalin, Julien Charles, Dermot Cochran, Matthew Dwyer, Arnout van Engelen, Alexander Fuchs, Connor Gallagher, Robin Green, Radu Grigore, George Hagen, Clément Hurlin, Perry James, Mikoláš Janota, George Karabotsos, Hermann Lehner, Alan Morkan, Michal Mosal, Carl Pulley, Frédéric Rioux, Will Sargent, and Aleksy Schubert.

References

1. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4(1):32–54, 2005.
2. Wolfgang Ahrendt, Richard Bubel, and Reiner Hähnle. Integrated and tool-supported teaching of testing, debugging, and verification. In *2nd Int. Conference on Teaching Formal Methods (TFM'2009)*, 2009. To appear.
3. Michael Barnett, Robert DeLine, Manuel Fähndrich, Bart Jacobs, K. Rustan M. Leino, Wolfram Schulte, and Herman Venter. The Spec# programming system: Challenges and directions. In Bertrand Meyer and Jim Woodcock, editors, *VSTTE*, volume 4171 of *LNCS*, pages 144–152, 2008.
4. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS*, number 2031 in LNCS, pages 299–312. Springer, Berlin, 2001.
5. L. Burdy, Y. Cheon, D.C. Cok, M.R. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.
6. Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003*, volume 2805 of *LNCS*, pages 422–439. Springer, 2003.

7. Patrice Chalin, Perry R. James, and George Karabotsos. JML4: Towards an industrial grade IVE for Java and next generation research platform for JML. In Natarajan Shankar and Jim Woodcock, editors, *VSTTE*, volume 5295 of *LNCS*, pages 70–83. Springer, 2008.
8. Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, June 2002.
9. Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002*, volume 2374 of *LNCS*, pages 231–255. Springer, 2002.
10. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
11. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.
12. Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS'2004)*, volume 3362 of *LNCS*, pages 108–128. Springer, 2005.
13. Joseph R. Kiniry and Daniel M. Zimmerman. Secret ninja formal methods. In *FM '08: Proceedings of the 15th international symposium on Formal Methods*, pages 214–228. Springer, 2008.
14. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06t, Iowa State University, Department of Computer Science, June 2002.
15. K. Rustan M. Leino and Rosemary Monahan. Automatic verification of textbook programs that use comprehensions. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP'2007)*, 2007.
16. Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML. *J. Log. Algebr. Program.*, 58(1-2):89–106, 2004.
17. Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, second edition, 1997.
18. Wojciech Mostowski. Fully verified Java Card API reference implementation. In Bernhard Beckert, editor, *Verify'07: 4th International Verification Workshop*, volume 259 of *CEUR WS*, July 2007.
19. Robby, Edwin Rodríguez, Matthew B. Dwyer, and John Hatcliff. Checking JML specifications using an extensible software model checking framework. *STTT*, 8(3):280–299, 2006.