Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

An overview of JML tools and applications

Lilian Burdy[1], Yoonsik Cheon[2], David R. Cok[3], Michael D. Ernst[4], Joseph R. Kiniry[5], Gary T. Leavens[6]*, K. Rustan M. Leino[7], Erik Poll[5]

[1] INRIA, Sophia-Antipolis, France
[2] Dept. of Computer Science, University of Texas at El Paso, El Paso, Texas, USA
[3] Eastman Kodak Company, R&D Laboratories, Rochester, New York, USA
[4] Computer Science & Artificial Intelligence Lab, MIT, Cambridge, Massachusetts, USA
[5] Dept. of Computer Science, University of Nijmegen, Nijmegen, the Netherlands
[6] Dept. of Computer Science, Iowa State University, Ames, Iowa, USA
[7] Microsoft Research, Redmond, Washington, USA

Received: date / Revised version: date

Abstract. The Java Modeling Language (JML) can be used to specify the detailed design of Java classes and interfaces by adding annotations to Java source files. The aim of JML is to provide a specification language that is easy to use for Java programmers and that is supported by a wide range of tools for specification type-checking, runtime debugging, static analysis, and verification.

This paper gives an overview of the main ideas behind JML, the different groups collaborating to provide tools for JML, and the existing applications of JML. Thus far, most applications have focused on code for programming smartcards written in the Java Card dialect of Java.

1 Introduction

JML [46,47], the Java Modeling Language, is useful for specifying detailed designs of Java classes and interfaces. JML is a behavioral interface specification language for Java; that is, it specifies both the behavior and the syntactic interface of Java code. The syntactic interface of a Java class or interface consists of its method signatures, the names and types of its fields, etc. This is what is commonly meant by an application programming interface (API). The behavior of such an API can be precisely documented in JML annotations; these describe the intended way that programmers should use the API. In terms of behavior, JML can detail, for example, the preconditions and postconditions for methods as well as class invariants, in the *Design by Contract* style.

An important goal for the design of JML is that it should be easily understandable by Java programmers. This is achieved by staying as close as possible to Java syntax and semantics. Another important design goal is that JML *not* impose any particular design methodology on users; instead, JML should be able to document Java programs designed in any manner.

The work on JML was started by Gary Leavens and his colleagues and students at Iowa State University. It has since grown into a cooperative, open effort. Several groups worldwide are now building tools that support the JML notation and are involved with the ongoing design of JML. For an up-to-date list, see the JML website, `www.jmlspecs.org`. The open, cooperative nature of the JML effort is important both for tool developers and users, and we welcome participation by others. For potential users, the fact that there are several tools supporting the same notation is clearly an advantage. For tool developers, using a common syntax and semantics can make it much easier to get users interested. After all, one of the biggest hurdles to using a new specification-centric tool is often the lack of familiarity with the associated specification language.

The next section introduces the JML notation. Sections 3 through 6 then discuss the tools currently available for JML in more detail. Section 7 discusses the applications of JML in the domain of Java Card, the Java dialect for programming smartcards. Section 8 discusses some related languages and tools, and Section 9 concludes.

2 The JML Notation

JML blends Eiffel's *Design by Contract* approach [54] with the Larch tradition [34,16,45] (both of which share features and ideas with VDM [42]).[1] Because JML sup-

* Supported in part by US NSF grants CCR-0097907 and CCR-0113181

[1] JML also has takes some features from the refinement calculus [55], which we do not discuss in this paper.

ports quantifiers such as `\forall` and `\exists`, and because JML allows model (i.e., specification-only) fields and methods, specifications can easily be made more precise and complete than is typical for Eiffel software. However, like Eiffel, JML uses Java's expression syntax in assertions, so that JML's notation is easier for programmers to learn than notations based on a language-independent specification language, such as the Larch Shared Language [47,48] or OCL [69].

Figure 1 gives an example of a JML specification that illustrates its main features. JML assertions are written as special annotation comments in Java code, either after `//@` or between `/*@ ... @*/`, so that they are ignored by Java compilers but can be used by tools that support JML. Within annotation comments JML extends the Java syntax with several keywords—in the example in Figure 1, the JML keywords `invariant`, `requires`, `assignable`, `ensures`, and `signals` are used. It also extends Java's expression syntax with several operators—in the example `\forall`, `\old`, and `\result` are used; these begin with a backslash so they do not clash with existing Java identifiers.

The central ingredients of a JML specification are preconditions (given in `requires` clauses), postconditions (given in `ensures` clauses), and invariants. These are all expressed as boolean expressions in JML's extension to Java's expression syntax.

In addition to *normal* postconditions, the language also supports *exceptional* postconditions, specified using `signals` clauses. These can be used to specify what must be true when a method throws an exception. For example, the `signals` clause in Figure 1 specifies that `debit` may throw a `PurseException` and that the balance will not change in that case (as specified by the use of the `\old` keyword).

The `assignable` clause for the method `debit` specifies a frame condition, namely that `debit` will assign only to the `balance` field. Although not a traditional part of design by contract languages like Eiffel, such frame conditions are essential for verification of code when using some of the tools described later.

There are many additional features of JML that are not used in the example in Figure 1. We briefly discuss the most important of these below.

- Model variables, which play the role of abstract values for abstract data types [19], allow specifications that hide implementation details. For example, if instead of a class `Purse`, we were specifying an interface `PurseInterface`, we could introduce the balance as such a model variable. A class implementing this interface could then specify how this model field is related to the class's particular representation of balance.
- JML comes with an extensive library that provides Java types that can be used for describing behavior mathematically. This library includes such concepts as sets, sequences, and relations. It is similar to libraries of mathematical concepts found in VDM, Z, LSL, or OCL, but allows such concepts to be used directly in assertions, since they are embodied as Java objects.
- The semantics of JML prevents side-effects in assertions. This both allows assertion checks to be used safely during debugging, and supports mathematical reasoning about assertions. This semantics works conservatively, by allowing a method to only be used in assertions only if it is declared as `pure`, meaning the method does not have any side-effects and does not perform any input or output [47]. For example, if there is a method `getBalance()` that is declared as `pure`,

  ```
  /*@ pure @*/ int getBalance() { ... }
  ```

  then this method can be used in the specification instead of the field `balance`.
- Finally, JML supports all the Java modifiers (`public`, `protected`, and `private`) for expressing visibility. For example, invariants can be declared `protected` if they are not observable by clients but intended for use by programmers of subclasses. (Technically, the invariants and method specifications in the Purse example of Figure 1 have default or package visibility, and thus would only be visible to code in the same package.)

3 Tools for JML

For a specification language, just as for a programming language, a range of tools is necessary to address the various needs of the specification language's users such as reading, writing, and checking JML annotations.

The most basic tool support for JML is parsing and typechecking. This already provides the first benefit of JML annotations over informal comments, as it will catch any typos, type incompatibilities, references to names that no longer exist, etc. The JML checker (`jml`) developed at Iowa State University performs parsing and and typechecking of Java programs and their JML annotations, but in fact most of the other tools mentioned below incorporate this functionality.

The rest of this paper describes the various tools that are currently available for JML. The following categorization serves also as an organization for the immediately following sections of this paper. We distinguish tools for checking of assertions at runtime, tools for statically checking of assertions (at or before compile-time), tools for generating specifications, and tools for documentation.

3.1 Runtime assertion checking and testing

One way of checking the correctness of JML specifications is by runtime assertion checking, i.e., simply run-

```
public class Purse {

    final int MAX_BALANCE;
    int balance;
    //@ invariant 0 <= balance && balance <= MAX_BALANCE;

    byte[] pin;
    /*@ invariant pin != null && pin.length == 4
      @          && (\forall int i; 0 <= i && i < 4;
      @                                0 <= pin[i] && pin[i] <= 9);
      @*/

    /*@ requires   amount >= 0;
      @ assignable balance;
      @ ensures    balance == \old(balance) - amount
      @            && \result == balance;
      @ signals (PurseException) balance == \old(balance);
      @*/
    int debit(int amount) throws PurseException {
        if (amount <= balance) { balance -= amount; return balance; }
        else { throw new PurseException("overdrawn by " + amount); }
    }

    /*@ requires   0 < mb && 0 <= b && b <= mb
      @            && p != null && p.length == 4
      @            && (\forall int i; 0 <= i && i < 4;
      @                                 0 <= p[i] && p[i] <= 9);
      @ assignable MAX_BALANCE, balance, pin;
      @ ensures    MAX_BALANCE == mb && balance == b
      @            && (\forall int i; 0 <= i && i < 4; p[i]==pin[i]);
      @*/
    Purse(int mb, int b, byte[] p) {
        MAX_BALANCE = mb; balance = b; pin = (byte[])p.clone();
    }
}
```

Fig. 1. Example JML specification

ning the Java code and testing for violations of JML assertions. Such runtime assertion checks are accomplished by using the JML compiler `jmlc` (Section 4.1).

Given that one often wants to do runtime assertion checking in the testing phase, there is also a `jmlunit` tool (Section 4.2) which combines runtime assertion checking with unit testing.

3.2 Static checking and verification

More ambitious than testing if the code satisfies the specifications at runtime, is verifying that the code satisfies its specification statically. This can give more assurance in the correctness of code as it establishes the correctness for all possible execution paths, whereas runtime assertion checking is limited by the execution paths exercised by the test suite being used. Of course, correctness of a program with respect to a given specification is not decidable in general. Any verification tool must trade off the level of automation it offers (i.e., the amount of user interaction it requires) and the complexity of the properties and code that it can handle. There are several tools for statically checking or verifying JML assertions providing different levels of automation and supporting different levels of expressivity in specifications:

– The program checker ESC/Java (Section 5.1) can automatically detect certain common errors in Java code and check relatively simple assertions.
– ESC/Java2 (Section 5.2) extends ESC/Java to support more of the JML syntax and to add other functionality.
– The CHASE tool (Section 5.3) automatically checks some aspects of frame conditions.
– The program checker JACK (Section 5.5) offers similar functionality to ESC/Java, but is more ambitious in attempting real program verification.
– The LOOP tool (Section 5.4) translates code annotated with JML specifications to proof obligations that one can then try to prove using the theorem prover PVS. The LOOP tool can handle more complex specifications and code than automatic checkers

like ESC/Java can, but at the price of more user interaction.
Krakatoa [53] is similar to LOOP, but integrates with the Coq theorem prover. It is currently under development and handles a subset of Java.

3.3 Generating specifications

In addition to these tools for checking specifications, there are also tools that help a developer write JML specifications, with the aim of reducing the cost and effort of producing JML specifications:

– The Daikon tool (Section 6.1) infers likely invariants by observing the runtime behavior of a program.
– The Houdini tool (Section 6.2) postulates annotations for code, then uses ESC/Java to check them.
– The `jmlspec` tool can produce a skeleton of a specification file from Java source and can compare the interfaces of two different files for consistency.

3.4 Documentation

Finally, in spite of all the tools mentioned above, ultimately human beings must read and understand JML specifications. Since JML specifications are also meant to be read and written by ordinary Java programmers, it is important to support the conventional ways that these programmers create and use documentation. The `jmldoc` tool developed by David Cok produces browsable HTML pages containing both the API and the specifications for Java code, in the style of pages generated by Javadoc [31].

This tool reuses the parsing and checking performed by the JML runtime assertion checker and connects it to the doclet API underlying Javadoc. In this way, `jmldoc` remains consistent with the definition of JML and creates HTML pages that are very familiar to a user of Javadoc. The `jmldoc` tool combines and displays in one place all of the specifications that pertain to a given class, interface, method, or field; it combines annotations across a series of source files that constitute successive refinements of a given class or interface, as well as the relevant specifications of overridden methods.

4 Runtime Assertion Checking and Testing

The most obvious way to use JML annotations is to test them at runtime and report any detected violations. There are two tools that can be used to do this.

4.1 Runtime Assertion Checking

The JML compiler (`jmlc`), developed at Iowa State University, is an extension to a Java compiler and compiles Java programs annotated with JML specifications into Java bytecode [15,17]. The compiled bytecode includes runtime assertion checking instructions that check JML specifications such as preconditions, normal and exceptional postconditions, invariants, and history constraints. The execution of such assertion checks is transparent in that, unless an assertion is violated, and except for performance measures (time and space), the behavior of the original program is unchanged. The transparency of runtime assertion checking is guaranteed, as JML assertions are not allowed to have any side-effects [48].

The JML language provides a rich set of specification facilities to write abstract, complete behavioral specifications of Java program modules [48]. It opens a new possibility in runtime assertion checking by supporting abstract specifications written in terms of specification-only declarations such as model fields, ghost fields, and model methods. Thus the JML compiler represents a significant advance over the state of the art in runtime assertion checking as represented by Design by Contract tools such as Eiffel [54], or by Java tools such as iContract [44] or Jass [6]. The `jmlc` tool also supports advances such as (stateful) interface specifications, multiple inheritance of specifications, various forms of quantifiers and set comprehension notation, support for strong and weak behavioral subtyping [52,21], and a contextual interpretation of undefinedness [48].

In sum, the JML compiler brings programming benefits to formal interface specifications by allowing Java programmers to use JML specifications as practical and effective tools for debugging, testing, and design by contract.

4.2 Unit Testing

A formal specification can be viewed as a test oracle [63, 3], and a runtime assertion checker can be used as the decision procedure for the test oracle [18]. This idea has been implemented as a unit testing tool for Java (`jmlunit`), by combining JML with the popular unit testing tool JUnit [7]. The `jmlunit` tool, developed at Iowa State University, frees the programmer from writing most unit test code and significantly automates unit testing of Java classes and interfaces.

The tool generates JUnit test classes that rely on the JML runtime assertion checker. The test classes send messages to objects of the Java classes under test. The testing code catches assertion violation errors from such method calls to decide if the test data violate the precondition of the method under test; such assertion violation errors do not constitute test failures. When the method under test satisfies its precondition, but otherwise has an assertion violation, then the implementation failed to meet its specification, and hence the test data detects a failure [18]. In other words, the generated test code serves as a test oracle whose behavior is derived from the specified behavior of the class being tested. The user is still responsible for generating test data; however the test classes make it easy for the user to supply such test data. In addition, the user can supply hand-written JUnit test methods if desired.

Our experience shows that the tool allows us to perform unit testing with minimal coding effort and detects many kinds of errors. Ironically, about half of our test failures were caused by specification errors, which shows that the approach is also useful for debugging specifications. In addition, the tool can report assertion coverage information, identifying assertions that are always true or always false, and thus indicating deficiencies in the set of test cases. However, the approach requires specifications to be fairly complete descriptions of the desired behavior, as the quality of the generated test oracles depends on the quality of the specifications. Thus, the approach trades the effort one might spend in writing test cases for effort spent in writing formal specifications.

5 Static Checking and Verification

As mentioned before, there are several tools for statically checking – or verifying – JML annotations, providing different degrees of rigour and different degrees of automation.

5.1 Extended Static Checking with ESC/Java

ESC/Java tool [29], developed at Compaq Research, performs what is called *extended static checking* [20,49], compile-time checking that goes well beyond type checking. It can check relatively simple assertions and can check for certain kinds of common errors in Java code, such as dereferencing `null`, indexing an array outside its bounds, or casting a reference to an impermissible type. ESC/Java supports a subset of JML and also checks the consistency between the code and the given JML annotations. The user's interaction with ESC/Java is quite similar to the interaction with the compiler's type checker: the user includes JML annotations in the code and runs the tool, and the tool responds with a list of possible errors in the program.

JML annotations affect ESC/Java in two ways. First, the given JML annotations help ESC/Java suppress spurious warning messages. For example, in Figure 1, the constructor's precondition `p != null` lets ESC/Java determine that the dereference of `p` in the constructor's body is valid, and thus ESC/Java produces no `null`-dereference warning. Second, annotations make ESC/-Java do additional checks. For example, when checking a caller of the `Purse` constructor, the precondition `p != null` causes ESC/Java to emit a warning if the actual parameter for `p` may be passed in as `null`. In these two ways, the use of JML annotations enables ESC/Java to produce warnings not at the source locations where errors manifest themselves at runtime, but at the source locations where the errors are committed.

An interesting property of ESC/Java is that it is neither sound nor complete; that is, it neither warns about all errors, nor does it warn only about actual errors. This is a deliberate design choice: the aim is to increase the cost-effectiveness of the tool. In some situations, convincing a mechanical checker of the absence of some particular error may require a large number of JML annotations (consider, for example, a hypothetical program that dereferences `null` if four of the program's large-valued integer variables satisfy the equation in Fermat's Last Theorem). To make the tool more cost-effective, it may therefore be prudent to ignore the possibility of certain errors, which is what ESC/Java has been designed to do. The ESC/Java User's Manual [50] contains a list of all cases of unsoundness and incompleteness in ESC/Java.

Under the hood, ESC/Java is powered by detailed program semantics and an automatic (non-interactive) theorem prover. ESC/Java translates a given JML-annotated Java program into verification conditions [51, 30], logical formulas that are valid if and only if the program is free of the kinds of errors being analyzed. Any verification-condition counterexamples found by the theorem prover are turned into programmer-sensible warning messages, including the kind and source location of each potential error. The ESC/Java User's Manual [50] also provides a detailed description of the semantics of JML annotations, as they pertain to ESC/Java.

5.2 ESC/Java2

Development of version 1 of ESC/Java ceased with the dissolving of the SRC group at Compaq. Consequently Cok and Kiniry have in progress a version 2 of ESC/Java, built on the source code release provided by Compaq and HP. This version has the following goals:

– to migrate the code base of ESC/Java and the code accepted by ESC/Java to Java 1.4;
– to update ESC/Java to accept annotations consistent with current version of JML;
– to increase the general use of the tool by packaging it in an easy-to-use form;
– to increase the amount of JML that ESC/Java statically checks, consistent with the original engineering goals of ESC/Java;
– and, over time, to update the associated tools of the ESC suite (Calvin, Houdini, RCC) in a similar manner.

There is currently an alpha version of ESC/Java2 available (from `http://www.cs.kun.nl/ita/research/projects/sos/projects/escjava.html`). This release includes the following improvements with respect to the original ESC/Java.

– Parses Java 1.4 (the old version only parsed Java 1.3). In particular ESC/Java2 handles the Java `assert` statement, treating it as either a Java assert statement that possibly throws an exception or as a JML assert statement that may provoke a static checker warning.

– Parses all of current JML. This is a somewhat moving target, since JML is the subject of current discussion and research. Nevertheless the core part of JML is stable and that is the portion that ESC/Java2 attempts to statically check. Some of the more esoteric features of JML (e.g. model programs) are only parsed, are not thoroughly type checked, and are ignored for purposes of static checking.
– Allows specifications to be placed in (multiple) files separate from the implementation, using JML's refinement features. ESC/Java2 makes checks by combining all available specifications and implementations. It also checks these specifications for consistency.
– Follows the JML semantics for specification inheritance. The constructs specific to ESC/Java version 1 (`also_requires`, etc.) were dropped.
– Increased the set of JML features that are statically checked, as described in the implementation notes accompanying the release.

There are two major areas of development of ESC/-Java2 that will improve overall usability of the tool, besides performance improvements. The first is the use of model variables and method calls in annotation expressions. Model variables are an important abstraction mechanism in writing specifications and model methods allow much more readable and compact specifications. This is a current topic of research and experimentation and some partial facilities in this direction are a part of the current alpha release of ESC/Java2.

The second major area for development is checking of the frame conditions (i.e., JML's `assignable` clauses, also known as `modifies` clauses). It is an acknowledged unsoundness of ESC/Java that these are not checked and faulty `assignable` clauses can be a subtle source of errors. In particular, the default `assignable` clause in some JML specifications is that everything is potentially modified; this interpretation is not currently implemented.

5.3 Chase

The CHASE tool highlights the complementary nature of the tools available for JML. As previously mentioned, one source of unsoundness of ESC/Java is that it does not check `assignable` clauses. The semantics of these frame axioms are also not checked by the JML compiler. The CHASE tool [14] tries to remedy these problems. It performs a syntactic check on `assignable` clauses, which, in the spirit of ESC/Java, is neither sound nor complete, but which spots many mistakes made in the user's `assignable` clauses. This is another example of the utility of a common language; developers can reap the benefits of complementary tools. (The latest JML tools from Iowa State are also incorporating some of this functionality.)

5.4 Program Verification with LOOP

The LOOP project at the University of Nijmegen started out as an exploration of the semantics of object-oriented languages in general, and Java in particular. A denotational semantics of sequential Java was defined in the language of the theorem prover PVS [62], and an associated compiler, called the LOOP tool [8], was developed, which translates any given sequential Java class into PVS theories describing its semantics. In order to conveniently use this as a basis for the specification and verification of Java code, the LOOP tool was then extended to also provide a formal semantics of JML, so that the tool now translates JML-annotated Java code into proof obligations for PVS, which one can try to prove interactively, in PVS. These proof obligations are expressed as a special kind of Hoare statements about methods, and they be proved using an associated Hoare logic [41] and weakest-precondition calculus [40] for Java and JML, both of which have been formalized in PVS. The LOOP tool generates a single proof obligation for every method and constructor, expressed as a Hoare statement. It does not, as commonly done in verification condition generators, split this up into smaller verification conditions. Instead, this splitting up is done inside the theorem prover PVS, using dedicated proof strategies.

A more detailed overview of the LOOP project is given in [39]. Case studies with the LOOP tool are discussed in [10, 38].

A difference between LOOP and both ESC/Java and JACK is that it provides a so-called shallow embedding of Java and JML in PVS, defining a formal (denotational) semantics of both Java and JML in PVS. This semantics is still executable to a degree, and it has been extensively tested against real Java implementations. The Hoare logic and wp-calculi that are used have also been completely formalized and proved sound with respect to these semantics in PVS. Both ESC/Java and JACK directly rely on an axiomatic semantics.

Verification of JML-annotated code with the LOOP tool (more in particular, the interactive theorem proving with PVS that this involves) can be very labor-intensive, but allows verification of more complicated properties than can be handled by extended static checking with ESC/Java. Because of this labor-intensive nature, one will typically first want to use other, less labor-intensive, approaches, such as runtime assertion checking or extended static checking, to remove some of the errors in the code or specifications before turning to the LOOP tool. Experiences with such a combined approach are described in [9]. The possibility to do this is an important —if not crucial— advantage of using a specification language that is supported by a range of tools.

Another tool for the interactive verification of JML-annotated Java code is Krakatoa [53]. This tool, which currently covers only a subset of sequential Java, produces proof obligations for the theorem prover Coq [5].

5.5 Static Verification with JACK

The JACK [12] tool has been developed at the research lab of Gemplus, a manufacturer of smartcards and smartcard software. JACK aims to provide an environment for Java and Java Card program verification using JML annotations. It implements a fully automated weakest precondition calculus in order to generate proof obligations from JML-annotated Java sources. Those proof obligations can then be discharged using a theorem prover. Currently the proof obligations are generated for the B-Method's prover [1].

The approach taken in JACK is somewhere between the approaches of ESC/Java and LOOP, but probably closer to LOOP than to ESC/Java; JACK tries to provide the best features of both of these tools. On the one hand, JACK is much more ambitious than ESC/Java, in that it aims at real program verification rather than just extended static checking, and JACK does not make all the assumptions that result in soundness issues in ESC/Java, some of which were made to speed up checking. On the other hand, JACK does not require its users to have expertise in the use of a theorem prover as LOOP does.

An important design goal of the JACK tool is to be usable by normal Java developers, allowing them to validate their own code. Thus, care has been taken to hide the mathematical complexity of the underlying concepts, and JACK provides a dedicated proof obligation viewer. This viewer presents the proof obligations as execution paths within the program, highlighting the source code relevant to the proof obligations. Moreover, goals and hypotheses are displayed in a Java/JML-like notation. To allow developers to work in a familiar environment, JACK is integrated as a plug-in in the popular Eclipse[2] IDE.

As earlier mentioned, JACK provides an interface to the automatic theorem prover of the Atelier B toolkit. The prover can usually automatically prove up to 90% of the proof obligations; the remaining ones have to be proved outside of JACK, using the classical B proof tool. However, JACK is meant to be used by Java developers, who cannot be expected to use the B proof tool. Therefore, in addition to the proved and unproved states, JACK adds a *checked* state, that allows developers to indicate that they have manually checked the proof obligation. In order to better handle those cases, other different approaches could be investigated, such as integration with test tools such as `jmlunit`, integration of other proof assistants, or perhaps support from a proof-expert team.

Like ESC/Java, JACK tries to hide the complications of the underlying theorem prover from the user, by providing a push-button tool that normal Java developers, and not just formal methods experts, can and would would like to use. We believe that this may be a way to let non-experts venture into the formal world.

[2] `http://www.eclipse.org`

6 Generating Specifications

Apart from the whole issue of checking that implementations meet specifications, an important bottleneck in the use of any formal specification language is writing specifications in the first place. The JML tools discussed so far assume the existence of a JML specification, and leave the task of writing it to the programmer. However, in practice this task can be time-consuming, tedious, and error-prone, so tools that can help in this task can be of great benefit.

6.1 Invariant Detection with Daikon

The Daikon invariant detector [24,25] is a tool that provides assistance in creating a specification. Daikon outputs observed program properties in JML syntax (as well as other output formats) and automatically inserts them into a target program.

The Daikon tool dynamically detects likely program invariants. In other words, given program executions, it reports properties that were true over those executions. The set of reported properties is also known as an *operational abstraction*. Invariant detection operates by observing values that a program computes at runtime, generalizing over those values, and reporting the resulting properties. The properties reported by Daikon encompass numbers ($x \leq y$, $y = ax + b$), collections ($x \in$ *mytree*, *mylist is sorted*), pointers ($n = n.child.parent$), and conditionals (*if* $p \neq$ *null then* $p.value > x$); a complete list appears in the Daikon user manual. Daikon is available at `http://pag.lcs.mit.edu/daikon/`. Several other implementations of dynamic invariant detection exist [35,66,37], but they do not presently produce output in JML format.

Like any dynamic analysis, the accuracy of the inferred invariants depends in part on the quality and completeness of the test cases, and other executions may falsify some of the reported properties. (Furthermore, the actual behavior of the program is not necessarily the same as its intended behavior.) However, Daikon uses static analysis, statistical tests, and other mechanisms to reduce the number of false positives [26]. Even if a property is not true in general, Daikon's output provides valuable information about the test suite over which the program was run. Combining invariant detection with a static verifier such as ESC/Java helps to overcome the problems of both techniques: the unsoundness of the dynamic analysis and the static analysis's need for annotations.

Even with modest test suites, Daikon's output is remarkably accurate. In one set of experiments [60], over 90% of the properties that it reported were verifiable by ESC/Java (the other properties were true, but were beyond the capabilities of ESC/Java), and it reported over 90% of the properties that ESC/Java needed in order to complete its verification. For example, if Daikon generated 100 properties, users had only to delete less than 10 properties and to add another 10 properties in order to have a verifiable set of properties. In another experiment [61], users who were provided with Daikon output (even from unrealistically bad test suites) performed statistically significantly better on a program verification task than did users who did not have such assistance.

In addition to aiding the task of static checking as described above, operational abstractions generated by the Daikon invariant detector have been used to generate and improve test suites [36,70,33], to automate theorem-proving [58,59], to identify refactoring opportunities [43], to aid program analysis [22,23], and to to detect anomalies and bugs [67,32,11], among other uses.

6.2 Inferring annotations with Houdini

An obstacle to using program verification tools such as ESC/Java on legacy code is the lack of annotations in such a program. The warnings more likely point out missing annotations than errors in the code. The Houdini tool [28,27] attempts to alleviate this problem by supplying many of the missing annotations.

Houdini works by making up *candidate annotations* for the given program. Such candidate annotations compare fields and array lengths to -1, 0, 1, constants used in array constructors, `null`, `true`, and `false` (depending on the type of the field), and indicate that arrays and sub-arrays contain no null elements. To find which of the candidate annotations hold for the program, Houdini repeatedly invokes ESC/Java, removing those candidate annotations that ESC/Java finds to be inconsistent with the code. When all remaining candidate annotations are consistent with the code, Houdini invokes ESC/Java a final time to produce warnings that are then presented to the user. Houdini thus retains the precision of ESC/Java, trading quick turnaround for a reduced annotation effort.

Note that any user-supplied JML annotations in the program still get used by Houdini, since they become part of each invocation of ESC/Java. Thus, the benefits of using JML annotations are the same for Houdini as for ESC/Java, but Houdini can find program errors from a smaller set of user-supplied JML annotations.

7 Applications of JML to Java Card

Although JML is able to specify arbitrary sequential Java programs, most of the serious applications of JML and JML tools up to now have targeted Java Card. Java Card^{TM} is a dialect of Java specifically designed for the programming of the latest generation of smartcards. Java Card is adapted to the hardware limitations of smartcards; for instance, it does not support floating point numbers, strings, object cloning, or threads.

Java Card is a well-suited target for the application of formal methods. It is a relatively simple language with a restricted API. Moreover, Java Card programs, called *applets* are small, typically on the order of several KBytes of bytecode. Additionally, correctness of Java Card programs is of crucial importance, since they are used in sensitive applications, e.g. as bank cards, identity cards, and in mobile phones.

JML, and several tools for JML, have been used for Java Card, especially in the context of the EU-supported project VerifiCard (`www.verificard.org`).

JML has been used to write a formal specification of almost the entire Java Card API [65]. This experience has shown that JML is expressive enough to specify non-trivial existing API classes. The runtime assertion checker has been used to specify and verify a component of a smartcard operating system [64].

ESC/Java has been used with great success to verify a realistic example of an electronic purse implementation in Java Card [13]. This case study was instrumental in convincing industrial users of the usefulness of JML and feasibility of automated program checking by ESC/Java for Java Card applets. In fact, this case study provided the motivation for the development of the JACK tool discussed earlier, which is specifically designed for Java Card programs. One of the classes of the electronic purse has also been verified using the LOOP tool [10]. An overview of the work on this electronic purse, and the way in which ESC/Java and LOOP can be used to complement each other, is given in [9].

As witnessed by the development of the JACK tool by Gemplus, Java Card smartcard programs may be one of the niche markets where formal methods have a promising future. Here, the cost that companies are willing to pay to ensure the absence of certain kinds of bugs is quite high. It seems that, given the current state of the art, using static checking techniques to ensure relatively simple properties (e.g., that no runtime exception ever reaches the top-level without being caught) seems to provide an acceptable return-on-investment. It should be noted that the very simplicity of Java Card is not without its drawbacks. In particular, the details of its very primitive communication with smartcards (via a byte array buffer) is not easily abstracted away from. It will be interesting to investigate if J2ME (Java 2 Micro Edition), which targets a wider range of electronic consumer products, such as mobile phones and PDAs, is also an interesting application domain for JML.

8 Related Work

8.1 Other runtime assertion checkers for Java

Many runtime assertion checkers for Java exist, for example Jass, iContract, and Parasoft's jContract, to name just a few. Each of these tools has its own specification language, thus specifications written for one tool do not work in any other tool. And while some of these tools support higher-level constructs such as quantifiers, all are quite primitive when compared to JML. For example, none include support for purity specification and checking, model methods, refinements, or unit test integration. The developers of Jass have expressed interest in moving to JML as their specification language.

8.2 SparkAda

SPARK (`www.sparkada.com`, [4]) is an initiative similar to JML in many respects, but much more mature, and targeting Ada rather than Java. SPARK (which stands for Spade Ada Kernel) is a language designed for programming high-integrity systems. It is a subset of Ada95 enriched with annotations to enable tool support. This includes tools for data- and information-flow analysis, and for code verification, in particular to ensure the absence of runtime exceptions [2]. Spark has been successfully used to construct high-integrity systems that have been certified using the Common Criteria, the ISO standard for the certification of information technology security. SPARK and the associated tools are marketed by Praxis Critical System Ltd., demonstrating that this technology is commercially viable.

8.3 JML vs. OCL

Despite the similarity in the acronyms, JML is *very* different in its aims from UML [68]. The most basic difference is that the UML aims to cover all phases of analysis and design with many notations, and tries to be independent of programming language, while JML only deals with detailed designs (for APIs) and is tied to Java. The *model* in JML refers to abstract, specification-only fields that can be used to describe the behavior of various types. By contrast, the *model* of UML refers to the general modeling process (analysis and design) and is not limited to abstractions of individual types.

JML does have some things in common with the Object Constraint Language (OCL) [69], which is part of the UML standard. Like JML, OCL can be used to specify invariants and pre- and postconditions. An important difference is that JML explicitly targets Java, whereas OCL is not specific to any one programming language. One could say that JML is related to Java in the same way that OCL is related to UML.

JML clearly has the disadvantage that it can not be used for, say, C++ programs, whereas OCL can. But it also has obvious advantages when it comes to syntax, semantics, and expressivity. Because JML sticks to the Java syntax and typing rules, a typical Java programmer will prefer JML notation over OCL notation, and, for instance, prefer to write (in JML):

```
invariant pin != null && pin.length == 5;
```

rather than the OCL:

```
inv: pin <> null and pin->size() = 5
```

JML supports all the Java modifiers such as `static`, `private`, `public`, etc., and these can be used to record detailed design decisions for different readers. Furthermore, there are legal Java expressions that can be used in JML specifications but that cannot be expressed in OCL.

More significant than these limitations, or differences in syntax, are differences in semantics. JML builds on the (well-defined) semantics of Java. So, for instance, `equals` has the same meaning in JML and Java, as does `==`, and the same rules for overriding, overloading, and hiding apply. One cannot expect this for OCL, although efforts to define a semantics for OCL are underway.

In all, we believe that a language like JML, which is tailored to Java, is better suited for recording the detailed design of a Java programs than a generic language like OCL. Even if one uses UML in the development of a Java application, it may be better to use JML rather than OCL for the specification of object constraints, especially in the later stages of the development.

9 Conclusions

We believe that JML presents a promising opportunity to gently introduce formal specification into industrial practice. It has the following strong points:

1. JML is *easy to learn* for any Java programmer, since its syntax and semantics are very close to Java.
   We believe this a crucial advantage, as the biggest hurdle to introducing formal methods in industry is often that people are not willing, or do not have the time, to learn yet another language.
2. There is no need to invest in the construction of a formal model before one can use JML. Or rather: the source code *is* the formal model. This brings further advantages:

  - It is easy to introduce the use of JML *gradually*, simply by adding the odd assertion to some Java code.
  - JML can be used for existing (legacy) code and APIs. Indeed, most applications of JML and its tools to date have involved existing APIs and code.
  - There is no discrepancy between the actual code and the formal model. In traditional applications of formal methods there is often a gap between the formal model and the actual implementation, which means that some bugs in the implementation cannot be found, because they are not part of the formal model, and, conversely, some problems discovered in the formal model may not be relevant for the implementation.
3. There is a growing availability of a wide range of tool support for JML.

Unlike B, JML does not impose a particular design methodology on its users. Unlike UML, VDM, and Z, JML is tailored to specifying both the syntactic interface of Java code and its behavior. Therefore, JML is better suited than these alternative languages for documenting the detailed design of existing Java programs.

As a common notation shared by many tools, JML offers users multiple tools supporting the same notation. This frees them from having to learn a whole new language before they can start using a new tool. The shared notation also helps the economics both for users and tool builders. Any industrial use of formal methods will have to be economically justified, by comparing the costs (the extra time and effort spent) against the benefits (improvements in quality, number of bugs found). Having a range of tools, offering different levels of assurance at different costs, makes it much easier to start using JML. One can begin with a technique that requires the least time and effort (perhaps runtime assertion checking) and then move to more labor-intensive techniques if and when that seems worthwhile, until one has reached a combination of tools and techniques that is cost-effective for a particular situation.

There are still many opportunities for further development of both the JML language and its tools. For instance, we would also like to see support for JML in integrated development environments (such as Eclipse) and integration with other kinds of static checkers. We believe that, as a common language, JML can provide an important vehicle to transfer more tools and techniques from academia to industry.

Of course, with more tools supporting JML, and the specification language JML growing in complexity due to the different features that are useful for the different tools, one important challenge is maintaining agreement on the semantics of the language between the different tools.

More generally, there are still many open research issues involving the specification of object-oriented systems that the JML effort is investigating. For instance, when exactly should invariants hold [57]? How should concurrency properties be specified? JML's specification inheritance forces behavioral subtyping [21,47], but subtyping in Java is used for implementation inheritance as well; is it practical to always weaken the specifications of supertypes enough so that their subtypes are behavioral subtypes? There are also semantics issues with frame axioms, pure methods, and aliasing, and practical ways to constrain the potential of aliasing, e.g. as proposed in [56].

The subtleties involved in such open problems are evidenced by the slightly different ways in which different tools approach these problems. This reflects the research (as opposed to industrial development) focus of most of those involved in JML and its tools. Nevertheless, JML seems to be successful in providing a common notation and a semantics that is, at least for a growing core subset, shared by many tools, and as a common notation, JML is already proving to be useful to both tool developers and users.

Acknowledgements. Despite our long list of co-authors, still more people have been involved in developing the tools discussed in this paper, including Joachim van den Berg, Cees-Bart Breunesse, Néstor Cataño, Cormac Flanagan, Mark Lillibridge, Marieke Huisman, Bart Jacobs, Jean-Louis Lanet, Todd Millstein, Greg Nelson, Jeremy Nimmer, Antoine Requet, Clyde Ruby, James B. Saxe, and Martijn Warnier. Thanks to Raymie Stata for his initiative in getting the JML and ESC/Java projects to agree on a common syntax, and to Michael Möller for the logo. Work on the JML tools at Iowa State builds on the MultiJava compiler written by Curtis Clifton as an adaptation of the Kopi Java compiler.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. Peter Amey and Roderick Chapman. Industrial strength exception freedom. In *ACM SigAda 2002*, pages 1–9. ACM, 2002.
3. Sergio Antoy and Dick Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.
4. John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison Wesley, 2003.
5. B. Barras, S. Boutin, C. Cornes, J. Courant, J.-Chr. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant User's Guide Version 6.1. Technical Report 203, INRIA Rocquencourt, France, May 1997.
6. D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass — Java with assertions. In *Workshop on Runtime*

Verification at CAV'01, 2001. Published in *ENTCS*, K. Havelund and G. Rosu (eds.), 55(2), 2001.

7. Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.
8. Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS'01*, number 2031 in LNCS, pages 299–312. Springer, 2001.
9. C.-B. Breunesse, N. Cataño, M. Huisman, and B.P.F. Jacobs. Formal methods for smart cards: an experience report. Technical report, University of Nijmegen, 2003. NIII Technical Report NIII-R0316.
10. Cees-Bart Breunesse, Joachim van den Berg, and Bart Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, *AMAST'02*, number 2422 in LNCS, pages 304–318. Springer, 2002.
11. Yuriy Brun. Software fault identification via dynamic analysis and machine learning. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 16, 2003.
12. Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In D. Mandrioli K. Araki, S. Gnesi, editor, *FME 2003*, volume 2805 of *LNCS*, pages 422–439. Springer-Verlag, 2003.
13. Néstor Cataño and Marieke Huisman. Formal specification of Gemplus's electronic purse case study. In L. H. Eriksson and P. A. Lindsay, editors, *FME 2002*, volume LNCS 2391, pages 272–289. Springer, 2002.
14. Néstor Cataño and Marieke Huisman. CHASE: A static checker for JML's assignable clause. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *VMCAI: Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *LNCS*, pages 26–40. Springer, 2003.
15. Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation. Available from `archives.cs.iastate.edu`.
16. Yoonsik Cheon and Gary T. Leavens. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, July 1994.
17. 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.
18. 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, June 2002.
19. Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10, Department of Computer Science, Iowa State University, April 2003. Available from `archives.cs.iastate.edu`.
20. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.
21. Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996.
22. Nii Dodoo, Alan Donovan, Lee Lin, and Michael D. Ernst. Selecting predicates for implications in program analysis, March 16, 2002. Draft. `http://pag.lcs.mit.edu/~mernst/pubs/invariants-implications.ps`.
23. Nii Dodoo, Lee Lin, and Michael D. Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, July 21, 2003.
24. Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.
25. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.
26. Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, 2000.
27. Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 77(2–4):97–108, February 2001.
28. Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME 2001*, volume LNCS 2021, pages 500–517. Springer, 2001.
29. 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.
30. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages*, pages 193–205. ACM, January 2001.
31. Lisa Friendly. The design of distributed hyperlinked programming documentation. In S. Fraïssè, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *IWHD'95*, pages 151–173. Springer, 1995.
32. Alex Groce and Willem Visser. What went wrong: Explaining counterexamples. In *10th International SPIN Workshop on Model Checking of Software*, pages 121–135, Portland, Oregon, May 9–10, 2003.
33. Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *Proceedings of the 13th Annual International Conference on Automated Software Engineering (ASE 2003)*, Montreal, Canada, October 8–10, 2003.

34. John V. Guttag, James J. Horning, et al. *Larch: Languages and Tools for Formal Specification*. Springer, New York, NY, 1993.
35. Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, Orlando, Florida, May 22–24, 2002.
36. Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *ICSE'03, Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, Portland, Oregon, May 6–8, 2003.
37. Johannes Henkel and Amer Diwan. Discovering algebraic specifications from Java classes. In *ECOOP 2003 — Object-Oriented Programming, 15th European Conference*, Darmstadt, Germany, July 23–25, 2003.
38. B. Jacobs, J. Kiniry, and M. Warnier. Java program verification challenges. In *FMCO 2002*, volume 2852 of *LNCS*, pages 202–219. Springer, 2003.
39. B. Jacobs and E. Poll. Java program verification at Nijmegen: Developments and perspectiv e. Technical report, University of Nijmegen, 2003. NIII Technical Report NIII-R0316.
40. Bart Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *JLAP*, 2002. To appear.
41. Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *LNCS*, pages 284–299. Springer, 2001.
42. Cliff B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.
43. Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *ICSM 2001, Proceedings of the International Conference on Software Maintenance*, pages 736–743, Florence, Italy, November 6–10, 2001.
44. Reto Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California*, pages 295–307, 1998.
45. Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.
46. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
47. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, April 2003. See `www.jmlspecs.org`.
48. Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In *FMCO 2002*, volume 2852 of *LNCS*, pages 262–284. Springer, 2003. Also appears as technical report TR03-04, Dept. of Computer Science, Iowa State University.
49. K. Rustan M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*. Springer, 2000.
50. K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq SRC, October 2000.
51. K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq SRC, May 1999.
52. Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
53. Claude Marché, Christine Paulin, and Xavier Urbain. The Krakatoa tool for JML/Java program certification. Available at `http://krakatoa.lri.fr`, 2003.
54. Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, NY, second edition, 1997.
55. Carroll Morgan. *Programming from Specifications: Second Edition.* Prentice Hall International, Hempstead, UK, 1994.
56. P. Müller, A. Poetzsch-Heffter, and G.T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15(2):117–154, 2003.
57. Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for object structures. Technical Report 424, ETH Zurich, October 2003.
58. Toh Ne Win and Michael D. Ernst. Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, May 25, 2002.
59. Toh Ne Win, Michael D. Ernst, Stephen J. Garland, Dilsun Kırlı, and Nancy Lynch. Using simulated execution in verifying distributed algorithms. *Software Tools for Technology Transfer*, 2004.
60. Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, 2002.
61. Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, 2002.
62. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in LNCS, pages 411–414. Springer, 1996.
63. Dennis K. Peters and David Lorge Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

64. Erik Poll, Pieter Hartel, and Eduard de Jong. A Java reference model of transacted memory for smart cards. In *Smart Card Research and Advanced Application Conference (CARDIS'2002)*, pages 75–86. USENIX, 2002.
65. Erik Poll, Joachim van den Berg, and Bart Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.
66. Brock Pytlik, Manos Renieris, Shriram Krishnamurthi, and Steven P. Reiss. Automated fault localization using potential invariants. In *AADEBUG'2003, Fifth International Workshop on Automated and Algorithmic Debugging*, Ghent, Belgium, September8–10, 2003.
67. Orna Raz, Philip Koopman, and Mary Shaw. Semantic anomaly detection in online data sources. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 302–312, Orlando, Florida, May 22–24, 2002.
68. Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Publishing Company, 1998.
69. Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Publishing Company, 1999.
70. Tao Xie and David Notkin. Tool-assisted unit test selection based on operational violations. In *Proceedings of the 13th Annual International Conference on Automated Software Engineering (ASE 2003)*, Montreal, Canada, October 8–10, 2003.