Int J Softw Tools Technol Transfer (2004) / Digital Object Identifier (DOI) 10.1007/s10009-004-0167-4

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, TX, USA
[3] Eastman Kodak Company, R&D Laboratories, Rochester, NY, USA
[4] Computer Science & Artificial Intelligence Lab, MIT, Cambridge, MA, USA
[5] Dept. of Computer Science, University of Nijmegen, Nijmegen, The Netherlands (e-mail: erikpoll@cs.ru.nl)
[6] Dept. of Computer Science, Iowa State University, Ames, IA, USA
[7] Microsoft Research, Redmond, WA, USA

Published online: 14 December 2004 –

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 typechecking, runtime debugging, static analysis, and verification.

This paper gives an overview of the main ideas behind JML, details about JML's wide range of tools, and a glimpse into existing applications of JML.

Keywords: Java – Formal specification – Assertion checking – Program verification – Design by Contract

1 Introduction

JML [57, 58], 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 [73].

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 Web site, `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 specificationcentric tool is often the lack of familiarity with the associated specification language.

The next section introduces the JML notation. Sections 3 through 7 then discuss the tools currently available for JML in more detail. Section 8 discusses the applications of JML in the domain of Java Card, the Java dialect for programming smartcards. Section 9 discusses some related languages and tools, and Sect. 10 concludes.

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

2 The JML notation

JML blends Eiffel's *Design by Contract* approach [73] with the Larch tradition [20, 41, 56] (both of which share features and ideas with VDM [52]).[1] Because JML supports 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, following Eiffel's use of its expression syntax in assertions, JML uses Java's expression syntax in assertions; this makes JML's notation easier for programmers to learn than notations based on a language-independent specification language, such as the Larch Shared Language [58, 59] or OCL [91].

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 ex-

[1] JML also takes some features from the refinement calculus [75], which we do not discuss in this paper.

```
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   p != null && p.length >= 4;
      @ assignable \nothing;
      @ ensures    \result <==> (\forall int i; 0 <= i && i < 4;
      @                                         pin[i] == p[i]);
      @*/
    boolean checkPin(byte[] p) {
        boolean res = true;
        for (int i=0; i < 4; i++) { res = res && pin[i] == p[i]; }
        return res;
    }

    /*@ 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

tends the Java syntax with several keywords – in the example in Fig. 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 Fig. 1's `debit` method 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 Fig. 1. We briefly discuss the most important of these below.

– Model variables, which play the role of abstract values for abstract data types [23], 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 forbids 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 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 [58]. 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 the Java modifiers (`private`, `protected`, and `public`) that control visibility of specifications. For example, an invariant can be declared to be `protected` if it is not observable by clients but is intended for use by programmers of subclasses. (Technically the invariants and method specifications in the Purse example of Fig. 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 an advantage over informal comments, as parsing and typechecking 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 typechecking of Java programs and their JML annotations, and 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 running the Java code and testing for violations of JML assertions. Such runtime assertion checks are accomplished by using the JML compiler *jmlc* (Sect. 4.1).

Given that one often wants to do runtime assertion checking in the testing phase, there is also a *jmlunit* tool (Sect. 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 ability to dispense with user interaction) 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* (Sect. 5.1) can automatically detect certain common errors in Java code and check relatively simple assertions.
- *ESC/Java2* (Sect. 5.2) extends *ESC/Java* to support more of the JML syntax and to add other functionality.
- The *LOOP* tool (Sect. 5.3) 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.
- The program checker *JACK* (Sect. 5.4) offers similar functionality to *ESC/Java* but is more ambitious in attempting real program verification.

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 (Sect. 6.1) infers likely invariants by observing the runtime behavior of a program.
- The *Houdini* tool (Sect. 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 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 (Sect. 7.1) produces browsable HTML pages containing both the API and the specifications for Java code, in the style of pages generated by *javadoc* [38].

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. In this section we discuss two tools, *jmlc* and *jmlunit*, that work this way.

4.1 Runtime assertion checking

4.1.1 Overview and goals

The goal of the JML compiler, *jmlc*, also known as the runtime assertion checker, is to find inconsistencies between specifications and code by executing assertions at runtime. The overall approach is to find such inconsistencies dynamically, by executing JML's assertions while the program runs and notifying the user of any assertion violations. As with other runtime assertion checkers, one normally hopes to find that the code is incorrect with respect to the specification. However, it may also be that the specification itself is incorrect (with respect to what the user has in mind), but the code is correct. Finding problems in specifications is important for keeping the specifications accurate and up-to-date; this solves a common problem with informal documentation, which cannot be mechanically checked against the program.

An important requirement for the runtime assertion checker is that it be good at isolating problems, in the sense that users of the tool should be able to quickly pinpoint what in either the code or specifications must be changed to correct an inconsistency. For this purpose, *jmlc* must provide information that is helpful for users. This includes both static information, such as what parts of the specification were violated and where in the program the violation was detected, as well as dynamic information about the values of variables and what method calls led to the violation (a stack backtrace).

It is also helpful, for isolating problems, if the runtime assertion checker can execute as large a subset of the JML language as possible.

The runtime assertion checker must also be trustworthy, in the sense that it must not generate false reports of assertion violations. That is, every assertion violation must be a report of an assertion that is false, according to the JML semantics. In meeting this goal, the runtime assertion checker can fail to report assertions that might be false. For example, JML includes a way to write informal descriptions in assertions; these informal descriptions are merely pieces of English text, and so only a human reader can decide whether they are true or false. If the runtime assertion checker were to assume some particular truth value for these it might report an assertion violation when none actually existed. In such cases it is better for the runtime assertion checker to not report a violation. Similarly, it is also acceptable for the runtime assertion checker to not execute some parts of assertions, especially in postconditions. However, not being able to execute some precondition could cause a method to fail unexpectedly; thus *jmlc* should give a warning for nonexecutable preconditions. In summary, it is better if the runtime assertion checker can execute all assertions and find all assertion violations, but this is a goal that can be incrementally approached during the development of the tool.

An important goal of the runtime assertion checker is that its work should be transparent when no assertions are violated. That is, except for time and space measurements, a correct program compiled with *jmlc* should behave just as if compiled with a normal Java compiler. The transparency of runtime assertion checking is aided by JML's design, as assertions are not allowed to have any side effects [59].

Although *jmlc* does not have to be used with any particular methodology, there are some general ideas for using such tools that are helpful for beginners [73]. A basic technique for using the runtime assertion checker is to first specify preconditions for the normal behavior of methods. This is easily done and helps ensure that all methods are called in expected states. For debugging purposes, it is also important to add `toString` methods to all types involved, so that *jmlc* can display object values in violation messages. Following this, one could define invariants that describe the legal states of objects of each class (see Sect. 6.1 for more on this topic). To help debug implementations, one can then advance to describing normal postconditions for methods. If one is describing a library for untrusted clients, it may also be useful to document when various exceptions are thrown by writing exceptional postconditions.

4.1.2 Design of the tool

The JML compiler was developed at Iowa State University as an extension to the MultiJava compiler [24]. It compiles Java programs annotated with JML specifications into Java bytecode [19, 21]. The compiled bytecode includes instructions that check JML specifications such as preconditions, normal and exceptional postconditions, invariants, and history constraints.

Because the JML language provides such a rich set of specification facilities, it presents new challenges in runtime assertion checking. One of these challenges that the current tool meets is supporting abstract specifications written in terms of specification-only declarations such as model fields, ghost fields, and model methods. This aspect of 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 [73] or by Java tools such as iContract [55] or Jass [9]. Other advances over such tools include (stateful) interface specifications, multiple inheritance of specifications from interfaces, various forms of quantifiers and set comprehension notation, support for strong and weak behavioral subtyping [28, 68], and a contextual interpretation of undefinedness [21].

4.1.3 Example

The specifications and code in Fig. 1 were debugged using the runtime assertion checker in combination with the unit testing tool described in Sect. 4.2.3. Using *jmlc* on the example is straightforward; the user simply tells the tool to compile the `Purse.java` file and then runs a test driver using *jmlrac* as the virtual machine. The *jmlrac* command is a version of the `java` command that knows about the necessary runtime libraries for runtime assertion checking. Assertion violations are printed as messages on the console. We discuss details of this kind of testing in Sect. 4.2.3.

4.1.4 Experience

The runtime assertion checker is one of the most widely used JML tools. It has been used on several case studies. One of the most demanding of these case studies is the checking of the built-in model types for JML itself, which have very rich and complete specifications. It has been used in several undergraduate classes, but in those cases it has also been used for simple, Design by Contract style, specifications. It has also been used in several of the other case studies mentioned in the remaining part of this paper. It seems to be helpful to use the runtime assertion checker before doing serious program verification, to make sure that the easily found bugs are removed before spending the effort to do verification.

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.

4.1.5 Future work

One of the main issues in future work on *jmlc* is improving the speed of both compilation and of runtime assertion check execution. For the latter, there seem to be several simple things that can be done to improve execution speed. For example, caching the values of model fields instead of recomputing them in several places within an assertion would be helpful.

Another direction for future work is being pursued at Virginia Tech by Stephen Edwards and his student Roy Tan. They are building a version of the JML compiler that produces separate bytecode files for the normal code and for a runtime assertion checking wrapper. Separating the runtime assertion checking code into this wrapper has several advantages. In particular, decisions about what classes should be checked can be made while the program executes. It will also enable the addition of runtime checks to code for which the source code is not available.

4.1.6 Availability

The runtime assertion checker is part of the main JML toolset available via `www.jmlspecs.org`, which is developed as an open source project hosted at `SourceForge.net`.

4.2 Unit testing

4.2.1 Overview and goals

A formal specification can be viewed as a test oracle [3, 84], and JML's runtime assertion checker can be used as the decision procedure for the test oracle [22]. This idea has been implemented as a unit testing tool for Java, *jmlunit*, by combining JML with the popular unit testing tool JUnit [10].

The main goal of the *jmlunit* tool is to significantly automate unit testing of Java code. More specifically, the goal is to free the programmer from writing the code that decides whether unit tests pass or fail.

4.2.2 Design of the tool

The *jmlunit* tool, developed at Iowa State University, 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 detect a failure [22]. 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 generated test classes make it easy for the user to supply these data. The tool comes with a framework that includes sample test data for the built-in Java value types. This framework allows one to combine, filter, and compose test data in several different ways to create a variety of tests. In addition, the user can supply handwritten JUnit test methods if desired. Such handwritten tests are useful for exploring combinations of method calls that the automatic testing ignores.

4.2.3 Example

In this subsection we discuss runtime assertion checking and unit testing with *jmlunit*, based on Fig. 1. To do unit testing with *jmlunit*, one first runs the *jmlunit* tool on the `Purse.java` file (technically, one has to use an option to tell the tool to test methods and constructors with package visibility). This produces a file, `Purse_JML_TestData.java`, into which test data are placed, and another file, `Purse_JML_Test.java`, that contains a driver to run the tests. In the first file we supplied data of the various types used as arguments to the methods being tested; this consists of integers (0, 1, −1, −22, etc.), `Purse` objects (such as `null` and `new Purse(1,1,p)`, where `p` is a 4-element array of bytes), and fresh byte arrays (such as `null`, `new byte[] {}`, `new byte[] {0,0,0}`, and `new byte[] {0, 0, 0, 0}`). To run the tests, one first compiles the classes being tested with *jmlc* (using a special option to flag unhandled and unspecified exceptions as errors); then the classes produced by *jmlunit* are compiled with a normal Java compiler; finally, one executes the automatically generated driver class, `Purse_JML_Test`, using *jmlrac*.

If all the annotations are removed from Fig. 1, then the unit testing process described in the previous paragraph does not detect any errors. This is because the unit testing tool is only testing for violations of assertions, and if there are no assertions, then no violations are detected. This illustrates the important observation that the quality of the testing that *jmlunit* provides is only as good as the specifications.

Consider a version of Fig. 1 that only includes the preconditions of the methods and the constructor but omits the invariants, the frame axioms, and all the normal and exceptional postconditions. Testing of `Purse` produces 11 failures, all of which are similar to that shown in Fig. 2. (Printing of `Purse` objects is handled by adding the obvious `toString` method to the code in Fig. 1.)

This error is the result of not specifying (i.e., deleting) the exceptional postconditions of the `debit` method. It shows that the condition in an exceptional postcondition can be alternatively considered as the negation of a precondition for normal behavior, which makes sense, because throwing an exception is not normal behavior. If the precondition of the `debit` method is changed to the following:

```
amount >= 0 && amount <= balance
```

```
1) debit(Purse_JML_Test$TestDebit)junit.framework.AssertionFailedError:
        Method 'debit' applied to
        Receiver: Purse(max=1, bal=0, pin={0123})
        Argument amount: 1
Caused by: org.jmlspecs.jmlrac.runtime.JMLExceptionalPostconditionError:
 by method Purse.debit regarding specifications at
        File "Purse.java", line 9, character 17, when
                'jml$e' is PurseException: overdrawn by 1
        at Purse.checkXPost$debit$Purse(Purse.java:256)
        at Purse.debit(Purse.java:347)
```

Fig. 2. Example output from testing with *jmlunit*

then all of these failures go away. This also happens if the `debit` method has the exceptional postcondition restored from Fig. 1, which tells the runtime assertion checker that such exceptions are expected.

This kind of testing is also effective at finding various omissions in preconditions. For example, if the precondition in the `checkPin` method or the constructor that specifies that the array must be of an appropriate length is omitted, then the tests will encounter failures.

Checking preconditions will not show places where the code is wrong, unless one method in the code calls another incorrectly. For the most part, errors in code are revealed by adding either invariants or postconditions. If we add the invariants back into the version of `Purse`, but still leave out the postconditions, then testing can detect omitted initialization of the `MAX_BALANCE` field in the constructor (although Java itself detects missing initializations of final fields, so for JML to detect this error, one also has to omit the `final` attribute from that field). Similarly, with the invariants the constructor's precondition must have the first line shown in Fig. 1, or many violations of the first invariant in the figure occur.

Adding postconditions from Fig. 1 allows many other errors in coding to be detected. For example, with all the postconditions restored, omissions of initializations of the `balance` and `pin` fields are detected. The postconditions can also detect incorrect coding in the loop of the `checkPin` method, but doing so requires test data for byte arrays that differ in only the positions not checked by the code; we had to add such data to our initial set of test data, since the original test data did not detect these errors. Figuring out the right test data to add in this case was subtle and could easily have been missed.

4.2.4 Experience

Our experience shows that the tool allows one 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.

4.2.5 Future work

JML/JUnit testing is limited in that it only detects problems that are the result of single method or constructor calls. Thus test data have to be carefully crafted so that the method is applied to objects in states that will fully exercise it. This process would be easier if the test drivers applied several methods in sequence to various pieces of data. One alternative for doing this would be to generate such sequences of method calls automatically. (An experimental version of Daikon can do this.) Another alternative is to augment JML with facilities to write specifications for blocks of example code to be used in testing.

4.2.6 Availability

jmlunit is part of the main JML toolset. This toolset is available at `www.jmlspecs.org`. It has been developed as an open source project hosted at `SourceForge.net`.

5 Static checking and verification

In this section, we describe several tools for statically checking – or verifying – JML annotations, providing different degrees of rigor and automation.

5.1 Extended static checking with ESC/Java

5.1.1 Overview and goals

The *ESC/Java* tool [36], originally developed at Compaq Research, performs what is called *extended static checking* [27, 60], compile-time checking that goes well beyond typechecking. 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. For this subset it 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 a 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.

5.1.2 Design of the tool

JML annotations affect *ESC/Java* in two ways. First, the given JML annotations help *ESC/Java* suppress spurious warning messages. For example, in Fig. 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 no `null`-dereference warning is produced. 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 [64] 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 (noninteractive) theorem prover, Simplify [26]. *ESC/Java* translates a given JML-annotated program into verification conditions [37, 61, 65]. Verification conditions are 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 [62]. The user's manual for *ESC/Java* [64] also provides a detailed description of the semantics of JML annotations, as they pertain to *ESC/Java*.

5.1.3 Example

We refrain from giving details of an *ESC/Java* example here. Instead, we describe an example in the context of *ESC/Java*'s successor, *ESC/Java2*, in Sect. 5.2.3.

5.1.4 Experience

The first major experience with *ESC/Java* was to apply the tool to the sources of its own front end, over 40 KLOC of Java. This source was "fully annotated," meaning that enough specifications were given for *ESC/Java* to check the front end for runtime errors (like `null` dereferences and array-index bounds errors) and specification violations (like precondition violations) without producing any warnings. This and some other early experiences are described in the *ESC/Java* overview paper [36].

Applications to Java Card are discussed in Sect. 8. The experience applying *ESC/Java* to Java Card was one of the motivations for the work on *ESC/Java2*, as maintaining different versions of the API specification, one using *ESC/Java*'s dialect of JML and one using the full JML language, was becoming a lot of work.

5.1.5 Availability

The final binary release (version 1.2.4) of *ESC/Java* is available from Compaq/HP's Web site: `www.research.compaq.com/downloads.html`. The source code (including that of related tools, e.g., Houdini, Calvin, and Simplify) is available as well. This source code release is obscurely named the "Java Programming Toolkit Source Release." *ESC/Java* only runs on x86 machines with Linux and Microsoft Windows, Sun's SPARC with Solaris, and Alpha processors with Hewlett-Packard's Tru64 Unix.

5.2 ESC/Java2

5.2.1 Overview and goals

Development of version 1 of *ESC/Java* had ceased by the time the Compaq Systems Research Center became part of HP Labs, where it was later dissolved. 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 the current version of JML;
- To increase the amount of JML that is checked, while remaining true to the original engineering goals of *ESC/Java*.

5.2.2 Design of the tool

ESC/Java2 follows the design of *ESC/Java*. In addition, *ESC/Java2*, like *ESC/Java*, recognizes that the state of the art of static checking is such that not all mismatches between code and specifications are reported by static checking tools; that is, there are aspects that are unsound, typically because some of the Java semantics are not yet fully modeled. Similarly, some generated warnings are not actually errors in the program; that is, there are aspects that are incomplete, typically because current theorem provers are insufficiently powerful. It is a goal of all such tools, including *ESC/Java2*, to be as sound and complete as is possible within reasonable engineering limits, but since no existing tools fully model or fully prove full multithreaded Java (indeed, portions of the semantics of the language are still being debated), the authors of both *ESC/Java* and *ESC/Java2* believe that it is in the interests of users to be explicit about the known sources of unsoundness and incompleteness.

ESC/Java2 does include improvements to *ESC/Java* in the following areas, while retaining backwards compatibility in all but a few features:

- It parses Java 1.4 (*ESC/Java* only parsed Java 1.3). In particular *ESC/Java2* handles the Java `assert` statement. A tool option allows the user to choose whether Java `assert` statements are treated as statements that may throw exceptions (per the Java semantics) or whether they are treated like `assert` statements in JML, which are checked by the static checker.
- It handles the current binary format for Java classes.

– It parses all of current JML. This is a somewhat moving target, since JML is the subject of ongoing 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 and are ignored for purposes of static checking.
– It 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.
– It follows the JML semantics for specification inheritance. The constructs specific to *ESC/Java* version 1 (`also_requires`, etc.) were dropped.
– It enlarges the set of JML features that are statically checked, most importantly:
  – Pure methods, which may be included in annotations;
  – Most aspects of `assignable` clauses;
  – Model fields, with the associated `represents`, `in`, and `maps` annotations.

5.2.3 Example

As an example, if the second invariant in Fig. 1 is omitted and the current *ESC/Java2* tool is applied to the source code, the warnings shown in Fig. 3 are produced. The warning messages indicate the likely problem and the source code location that violates the implicit or explicit specification, namely, in this case, the implicit specification that the left-hand operand of the dereference operation must not be a null reference and that the index of an array reference must be less than the array length.

If *ESC/Java2* is applied to `Purse.java` as it stands (using a current version of JML's specifications for Java system classes), a warning will be produced reflecting the fact that the specifications of the behavior of `clone` are not yet completed.

A source of unsoundness in *ESC/Java(2)* that is relevant in the Purse example is its handling of loops: by default, it will not attempt verification of the loop in `checkPin`, but simply unroll it once. This makes it easy for the programmer, who does not have to supply a loop invariant, but it may also miss errors. In contrast, *LOOP* and *JACK* (and *ESC/Java2* with the `-loopSafe` switch) handle loops soundly but then require users to supply loop invariants. For this case, the loop invariant as illustrated in Fig. 4 would have to be given.

5.2.4 Experience

The first major partial verification using *ESC/Java2* was done in early 2004 when the Dutch parliament decided in 2003 to construct an Internet-based remote voting system for use by Dutch expatriates. The SoS group at the University of Nijmegen was part of an expert review panel for the system and also performed a black-box network and system security evaluation of this system in late 2003. They also were responsible for designing, implementing, and verifying the vote tally subsystem of this system in early 2004. This implementation used JML and *ESC/Java2* extensively.

ESC/Java2 made a very positive impression on the SoS developers. Its increased capabilities as compared to Compaq *ESC/Java*, particularly with regards to handling the full JML language, the ability to reason with models, and specifications with pure methods, are very impressive. And, while the tool is still classified as an "alpha" release, we found it to be quite robust (perhaps unsurprising given its history, the use of JML and *ESC/Java2* in and on its own source code, and the fact that it is passed through seven alpha releases thus far). But there are still a number of issues with *ESC/Java2* and JML that were highlighted by this verification effort and are discussed in another paper [54].

5.2.5 Future work

There are a number of major areas of development of ESC/Java2 that will improve overall usability of the tool, besides performance improvements.

– The use of model variables and method calls in annotation expressions. Model variables are an important abstraction mechanism in writing specifications, and

```
Purse.java:31: Warning: Possible null dereference (Null)
        for (int i=0; i < 4; i++) { res = res && pin[i] == p[i]; }
                                                 ^
---------------------------------------------------------------------
Purse.java:31: Warning: Array index possibly too large (IndexTooBig)
        for (int i=0; i < 4; i++) { res = res && pin[i] == p[i]; }
                                                    ^
```

Fig. 3. Example *ESC/Java2* warnings

```
//@ loop_invariant 0 <= i <= 5;
//@ loop_invariant res == (\forall int j; 0 <= j && j < i; pin[j] == p[j]);
```

Fig. 4. Loop invariant

model methods allow much more readable and compact specifications [23]. This is a current topic of research and experimentation; most of what is needed to support these features is a part of the current alpha release of *ESC/Java2* [25].
– Checking of the frame conditions specified by JML's `assignable` clause (also known as `modifies`). It is an acknowledged unsoundness of *ESC/Java* that these are not checked, and faulty `assignable` clauses can be a subtle source of errors. *ESC/Java2* checks most aspects of `assignable` clauses. However, the default `assignable` clause in JML specifications is that everything is potentially modified; this interpretation is not currently implemented.
– Arithmetic. JML needs to have available for specifications both mathematical integers and reals as well as the finite-precision approximations that are used in computer programs. There is some initial work [18] incorporating these into JML but as yet no axiomatization that enables reasoning with *ESC/Java2*.

The most significant aspect of future work, however, is experimentation with specification and static checking of larger, more varied, and real-world bodies of source code. Such experimentation is needed to verify that JML has the facilities that are needed for realistic specifications and that static checking tools such as *ESC/Java2* are capable of providing a benefit to working programmers.

5.2.6 Availability

An alpha version of *ESC/Java2* is available at `http://www.cs.kun.nl/sos/research/escjava`. The tool is a Java program that is fairly platform independent, but it uses the Simplify prover, which is only available on Linux, Windows, Solaris, and MacOSX platforms.

5.3 Program verification with LOOP

5.3.1 Overview and goals

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. Only later did it evolve to investigate verification of JML-annotated Java. For a detailed overview of the *LOOP* project we refer the reader to [50].

5.3.2 Design of the tool

The project began with the formalization of a denotational semantics of sequential Java [51] in the language of the theorem prover PVS [82]. An associated compiler, called the *LOOP* tool [11], was developed that 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 are proved using an associated Hoare logic [49] and weakest-precondition calculus [47] for Java and JML, both of which have been formalized in PVS.

A difference between *LOOP* and both *ESC/Java(2)* and *JACK* (see Sect. 5.4 for the *JACK* tool) 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 has its advantages. The Hoare logic and wp-calculi that are used have been completely formalized and proven sound with respect to these semantics in PVS, whereas both *ESC/Java(2)* and *JACK* directly rely on an axiomatic semantics. Also, our semantics of Java in PVS is still (symbolically) executable to a degree as it lets PVS evaluate the denotation of a program. This has been very useful in the extensive testing and debugging of our formal semantics, where we compared the results of the normal execution of a Java program, i.e., the result of executing its bytecode on a Java VM, and the symbolic execution of its semantics in PVS.

5.3.3 Example

Using the *LOOP* tool to verify the example in Fig. 1 fails for the constructor, as it did for *ESC/Java*, because the specifications of the behavior of `clone` are incomplete. The verification of the methods is fully automatic using *LOOP*, using its weakest precondition calculus, except that the verification of `checkPin` needs manual interaction in PVS to supply the loop invariant, as the tool does not yet handle JML's `loop_invariant`.

5.3.4 Experience

Case studies with the *LOOP* tool are discussed in [12, 46, 48]. Verification of JML-annotated code with the *LOOP* tool (especially the required interactive theorem proving with PVS) can be very labor intensive, but it allows verification of more complicated properties than can be handled by fully automated extended static checking using *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 [13]. 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.

The *LOOP* tool generates a single proof obligation for each method and constructor, expressed as a Hoare statement. It does not, as is 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 disadvantage of this is that the size of proof obligations that can be comfortably handled in PVS has become a bottleneck.

5.3.5 Future work

Ongoing work on the *LOOP* tool includes support for the different forms of arithmetic as proposed in [18] and investigations into proving information flow properties. The longer-term plans for the *LOOP* tool are currently unclear.

5.3.6 Availability

The *LOOP* tool is not publicly available, simply because it is not easy to use without intensive user support and documentation that we cannot provide. Actually, *LOOP* itself is easy enough to use – it is simply a compiler that outputs PVS – but dealing with the large and numerous PVS theories it outputs requires considerable (PVS) expertise.

5.4 Static verification with JACK

5.4.1 Overview and goals

The *JACK* [15] tool was initially developed at the research lab of Gemplus, a manufacturer of smartcards and smartcard software. Further development is now happening at INRIA. *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 different theorem provers.

The main design goals are an easily accessible user interface, a high degree of automation, a high correctness assurance, and prover independence.

5.4.2 Design of the tool

The main goal of *JACK* is that it should be usable by normal Java developers, allowing them to validate their own code, following, in this way, the JML philosophy. Thus, care has been taken to hide the mathematical formulation of the underlying concepts. To allow developers to work in a familiar environment, *JACK* is integrated as a plug-in to the Eclipse[2] IDE. This plug-in allows users to generate proof obligations, to run the automatic provers, and to inspect the generated lemmas. To facilitate this last task, *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. The user can then work within its current development tool, add the JML annotations, and check partially the correctness of the code in a familiar environment.

JACK's core is an implementation, in Java, of a weakest-precondition calculus. This ensures proof obligation generation without user interaction. Following this step, automatic provers are used to prove the generated lemmas. Users then have to check whether or not any remaining lemmas are valid. To reduce the remaining costly manual task – creating the JML annotation assertions – we have developed and integrated in *JACK* a prototype that annotates source code with assertions by propagation of pre- and postconditions. This is a way to reduce the cost of using JML, since, at the moment, the main issue when using *JACK* is the time spent annotating classes.

JACK is not based on a formalization of Java, as is *LOOP*; thus one cannot easily prove the formal correctness of the tool, and the implementation of the weakest-precondition calculus can contain bugs. Nevertheless, the aim of the tool is to be complete and sound (i.e., to generate all proof obligations that are valid if and only if the application respects its formalization). So users can choose to check partially the correctness of their application by just reviewing the unproved proof obligations, or they can also prove all the proof obligations using an interactive theorem prover, thereby obtaining a complete assurance on the development correctness.

JACK provides an interface to automatic theorem provers. Currently, the prover of the Atelier B toolkit, Simplify (the prover used in *ESC/Java*), and PVS are integrated. These provers are integrated as plug-ins in *JACK*. Since *JACK* is based on an intermediate lemma formulation language, it is quite easy to integrate new provers by implementing a translator from this intermediate language to the prover input. Interfacing several provers increases the automatic proof ratio. This also allows people to prove any remaining lemmas interactively in their preferred prover.

The actually interfaced automatic provers 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, PVS, or the Coq proof assistant. However, *JACK* is meant to be used by Java developers, who cannot be expected to use a proof assistant. Therefore, in addition to the proved and unproved states, *JACK* adds a *checked* state, which 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.

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

5.4.3 Example

The code of the class given in the Fig. 1 was proved using *JACK*. To generate proof obligations automatically, loop invariants have to be given explicitly in the code. Here, the JML annotation of Fig. 4 is added in the body of the method `checkPin` before the `for` statement. When this annotation is added, one can run *JACK*, which then calculates proof obligations automatically and proves them using the automated provers. Here, only three proof obligations remain unproved due to, yet again, the lack of complete specification of the `clone()` method in the constructor.

5.4.4 Experience

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 like to use. We believe that this may be a way to let nonexperts venture into the world of formal verification.

ESC/Java, *LOOP*, and *JACK* all use (or, in the case of *LOOP*, have the option of using) a weakest-precondition calculus to generate verification conditions. *ESC/Java* and *LOOP* generate one verification condition per method implementation, whereas *JACK* generates roughly one verification condition per syntactic code path through the code. So each of *JACK*'s verification conditions is smaller than those generated by *ESC/Java* and *LOOP*. On the other hand, *JACK* may generate a very large number of verification conditions. Though it generates just one verification condition per method, *ESC/Java* factors its verification conditions differently than the other two tools (see [37, 61]) and therefore is able to keep the one verification condition reasonably small. More important than size, verification conditions generated by *ESC/Java* often let the theorem prover avoid redundant work. *JACK*'s approach has the advantage that it is easy to pass the different verification conditions to different theorem provers.

5.4.5 Future work

To increase the automation of this validation phase, we are currently thinking of interfacing *JACK* with a counterexample detector or runtime test generator. We are also still investigating the annotation generation and propagation techniques since we consider that it can be a way to reduce the cost of using the tool.

5.4.6 Availability

JACK is currently not publicly available.

6 Generating specifications

Apart from checking that implementations meet specifications, a considerable barrier to entry 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. 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

6.1.1 Overview and goals

The *Daikon* invariant detector [31, 32] 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.

6.1.2 Design of the tool

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*. Dynamic 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 ($x <= y$, $y == ax + b$), collections ($mytree.contains(x)$, $mylist.isSorted()$), pointers ($n == n.next.prev$), and implications (p $!= null ==> p.value > x$); a complete list appears in the *Daikon* user manual.

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 [33]. 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.

6.1.3 Example

In order to apply *Daikon* to a program, a user runs an instrumented version of the program to create a data trace file, then runs *Daikon* over the data trace file to produce likely invariants. The instrumented version of the program contains, at program points such as procedure entries and exits, code that writes the values of all variables in scope to a trace file. In some cases (as for *Daikon*'s C front end), the instrumentation is performed automatically on a compiled executable by a special run-time system. In other cases (as for *Daikon*'s Java front end), the user runs a source-to-source translator that instruments the program, then runs the instrumented program in place of the original.

Given a simple test suite that creates 1000 random `Purse` objects and invokes `debit` on each one, the *Daikon* tool automatically generates the annotations of Fig. 1, except that the current version of *Daikon* does not generate JML's `signals` clauses. *Daikon*'s output is correct JML that is parseable by the JML toolset.

6.1.4 Experience

Even with modest test suites, *Daikon*'s output is remarkably accurate. In one set of experiments [80], 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 [81], 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 [40, 44, 93], automate theorem proving [78, 79], identify refactoring opportunities [53], aid program analysis [29, 30], choose modalities [67], predict incompatibilities in component upgrades [71, 72], detect anomalies and bugs [14, 43, 70, 87, 89], and isolate errors [39, 66, 92], among other uses.

6.1.5 Future work

As noted above, *Daikon* does not generate JML `signals` clauses for exceptional method exits. Doing so requires enhancements to the language-specific front ends, but no significant changes to *Daikon* proper. Another instrumentation enhancement that we are pursuing is replacing the current Java instrumenter (which performs a source-to-source translation) by one that is embedded in the Java Virtual machine and works on compiled Java programs. This change will simplify using *Daikon* by reducing the work required of a user. Finally, making *Daikon* work online – taking data from a running program rather than from a trace file – will reduce the number of steps to one, which is the same as currently required to run any Java program (via the *java* command).

Our main research thrust is not to improve *Daikon* itself but to find more uses for the operational abstractions that it produces. Linking it to verification tools from the JML toolset is just one application; some others were noted above in Sect. 6.1.4.

6.1.6 Availability

Daikon is publicly available, in both source and compiled form, at `http://pag.csail.mit.edu/daikon/`. *Daikon* includes front ends for Java, C, Perl, and other languages and input formats.
Several other implementations of dynamic invariant detection exist [43, 45, 87]. However, they do not produce output in JML format, are not publicly available, and check and report only a small fraction of the properties that *Daikon* does [83].

6.2 Inferring annotations with Houdini

6.2.1 Overview and goals

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 [34, 35] attempts to alleviate this problem by supplying many of the missing annotations.

6.2.2 Design of the tool

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 subarrays 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.

6.2.3 Example

If the class in Fig. 1 is given to *Houdini* without any annotations, then *Houdini* will produce a number of candidate annotations, including the invariants `0 <= balance` and `1 <= balance` and the `Purse`-constructor preconditions `0 <= b` and `1 <= b`. If the given program contains a call to the `Purse` constructor that passes in `0` for `b`, then the candidate precondition `1 <= b` is refuted and removed. Since the constructor assigns `b` to `balance`, the candidate invariant `1 <= balance` will then eventually also become refuted.

Houdini will also include `balance <= MAX_BALANCE`, among many other candidate annotations, but will not include, for example, the universal quantifications shown in Fig. 1.

6.2.4 Experience

Houdini has been applied to a number of real application programs, the initial account of which is reported in [35]. For each of the applications, *Houdini* (in concert with *ESC/Java*) was able to find errors. The number of warnings produced was generally larger than the number of warnings inspected by a user. For example, in the 36 KLOC program "Cobalt" [35], only 200 of the 540 warnings were inspected by a user, though this inspection revealed 8 errors. In the largest program to which *Houdini* was applied, a systems administration tool comprising 500 KLOC of Java, the number of warnings produced was too large to be particularly useful, though an inspection of 10 of the warnings still revealed 2 program errors. The experience with *Houdini*, albeit limited, suggests that it is possible for a user to inspect a program for errors at a rate of upwards of 1000 LOC per hour.

6.2.5 Future work

Though *Houdini* has found real errors, some problems make the tool less effective than one would like. We mention three such problems here.

First, *Houdini*'s simple strategy for producing candidate annotations limits the number of *ESC/Java* warnings it can suppress. Future work might consider applying more static analysis or dynamic profiling to improve the initial set of candidate annotations.

Second, to reduce the number of warnings produced, it is important for *Houdini* to infer good class invariants. Even in the cases where *Houdini*'s candidate set includes the necessary invariants, *Houdini* may fail to infer them because the first point at which they hold is unknown. For example, *ESC/Java* generally checks that an object's invariant has been established before the object's constructor invokes any method on the object. But the purpose of such a method invocation is sometimes to help establish the object's invariant in the first place. In an attempt to improve this situation, *Houdini* uses a special mode of *ESC/Java*, where *ESC/Java* inlines any method call from a constructor. This mode allows *Houdini* to infer better invariants but sometimes produces enormous verification conditions in *ESC/Java*. Future work might find a better solution to this problem.

Third, to avoid forcing users to write loop invariants, *ESC/Java* by default analyzes only a fixed number of unrollings of each loop. If the loop is known always to go through more iterations than are unrolled (for example, if a `for` loop iterates exactly 10 times, where 10 is a constant mentioned in the loop head), then the effect is that *ESC/Java*'s analysis never reaches the other side of the loop. This may be acceptable in a manual application of *ESC/Java*, since *ESC/Java* performs modular checking method by method, and therefore the checking of other methods is unaffected. However, for *Houdini*, whose inference is more like that of a whole-program analysis, this situation can have a paralyzing effect on the entire program analysis. *Houdini* sidesteps this situation by using a special mode of *ESC/Java*, where *ESC/Java* in effect introduces a jump from its last unrolling of the loop until after the loop. This is much better for *Houdini*, but it also introduces execution paths that do not exist in the given program, which leads to other problems. Perhaps there are better solutions.

6.2.6 Availability

Work on the *Houdini* tool petered out in 2001 with the transformation of the Compaq Systems Research Center. The sources of the final version of *Houdini* are available in the *ESC/Java* source distribution, named the "Java Programming Toolkit Source Release," at `http://www.research.compaq.com/downloads.html`.

7 Documentation

Generating human-readable Web pages from JML specifications is accomplished by the *jmldoc* tool.

7.1 jmldoc

7.1.1 Overview and goals

The goal of the *jmldoc* tool is to produce HTML pages like those produced by the *javadoc* tool, but including information from JML annotations as well. JML allows specifications to be spread across a number of refinement files. This is essential, for example, in the case that the Java source code may not be modified to include specifications directly in the source code. Even within one file, the specifications relevant to the class may be spread throughout the file, making easy spotting of a relevant invariant difficult. Also, JML enforces behavioral inheritance, in which an overriding method must satisfy the specification of the methods it overrides. Accordingly, *jmldoc* includes in the HTML representation of the specifications of a method the specifications of the methods it overrides. By combining and grouping these specifications appropriately, *jmldoc* makes them more accessible to the programmer. Particularly for those accustomed to browsing the *javadoc* documentation of an API, the inclusion of the additional specifications in a formal notation as provided by *jmldoc* is expected to be a convenience.

7.1.2 Design of the tool

The *jmldoc* tool is designed to leverage as much of both the JML tools and the *javadoc* tool as possible. It uses the classes of the JML checker to parse, typecheck, and provide an AST that includes specifications of each class and method being documented. The *javadoc* tool provides a *doclet API* that allows some reuse of the *javadoc* framework. Many of the contributed doclets use the provided classes to parse valid Java source code with *javadoc* comments and then to do checks or alternative processing on those files, such as producing PDF rather than HTML or checking that all methods do indeed have *javadoc* comments. The *jmldoc* tool instead alters (by derivation) the mechanism that generates the HTML pages so that the output will contain, in addition, information about the JML annotations in the source files, as provided by the JML-generated AST. In this way, the *jmldoc* tool remains consistent with the other JML tools in their handling of the JML language, but it also produces HTML pages consistent with other current *javadoc* documentation and with that produced by the *javadoc* tool itself. Aside from accepting additional command-line options appropriate to JML, *jmldoc* is intended to be a drop-in replacement for *javadoc*.

7.1.3 Example

An example of *jmldoc*'s output is shown in Fig. 5; it shows the current output produced for the method `HashMap.size` as currently specified by JML specifications for Java system classes.

7.1.4 Experience

The main experience we have with *jmldoc* is in documentation of packages that ship with JML, such as JML's built-in types for modeling and its samples, and with documentation of parts of the Java standard libraries. While these are used by JML users on a daily basis, there have been no formal case studies of the usefulness of *jmldoc*. Informal reports, however, have been positive.

7.1.5 Future work

The tool is being maintained as part of the JML toolset but not being extended further, other than to keep pace with changes in the definition of JML itself. Extensive maintenance is also needed to keep pace with changes in the doclet API with each new version of Java. As it happens, the portions of the doclet API that are extended by *jmldoc* have been changing significantly even between minor releases of *javadoc*. If this rate of change continues, the JML project may need to seek an alternative design that is not tied as closely to the current appearance of *javadoc* documentation in order to lessen the maintenance burden.

7.1.6 Availability

The *jmldoc* tool was authored by David Cok along the lines of the goals espoused by Raghavan [88]. It is part of the main JML toolset available via `www.jmlspecs.org`, which is developed as an open source project hosted at `SourceForge.net`.

8 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 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[3], or threads.

[3] The fact that Java Card does not have cloning means that a version of the `Purse` example in Fig. 1 rewritten to Java Card rather than Java does verify using *ESC/Java*, *LOOP*, or *JACK*. Indeed, the absence of `clone` in Java Card is a reason why dealing with `clone` has not been a priority in these tools.

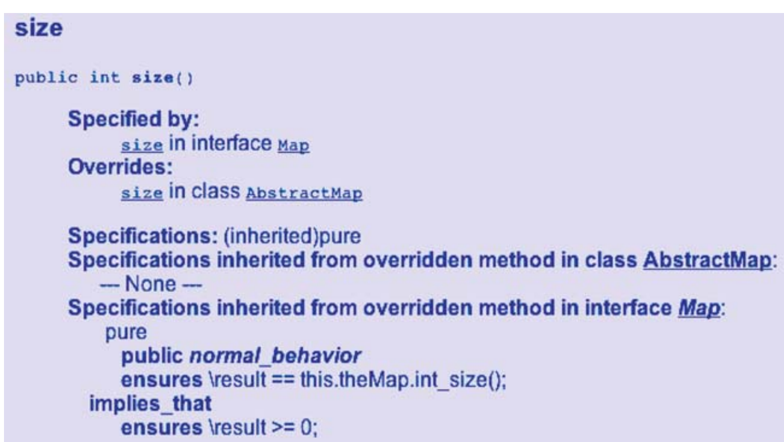

Fig. 5. Example *jmldoc* output

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. Furthermore, once such smartcards are issued, it is difficult, if not impossible, to fix any software errors.

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 [86]. This experience has shown that JML is expressive enough to specify nontrivial existing API classes. The runtime assertion checker has been used to specify and verify a component of a smartcard operating system [85].

ESC/Java has been used with great success to verify a realistic example of an electronic purse implementation in Java Card [16]. 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 [12]. 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 [13].

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.

9 Related work

9.1 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 includes 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.

The *ChAsE* tool [17] is a static checker for JML's `assignable` clauses. It performs a syntactic check on such 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. *ChAsE* was developed to complement the functionality missing in other tools: not checking assignable clauses was one of the sources of unsoundness of *ESC/Java*. Also, assignable clauses are not checked by the runtime assertion checker, making errors in assignable clauses hard to detect. The functionality to check assignable clauses is now incorporated in *ESC/Java2*. Also, the JML runtime assertion checker has started to incorporate some of this functionality.

In addition to *ESC/Java(2)*, *LOOP*, and *JACK*, several other tools exist for the verification of Java code, for instance *Krakatoa* [69], *Jive* [74], and *KeY* [1]. The *Krakatoa* tool also uses JML as specification language; it produces proof obligations for the theorem prover Coq. It is planned that *Jive* will also start supporting JML. The *KeY* tool uses OCL instead as its specification language and is integrated with a commercial CASE tool.

9.2 Other languages

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 (with no object references and subclasses, for example) 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 Systems Ltd., demonstrating that this technology is commercially viable.

A more recent initiative that is very similar to JML is Spec# [6]. The Spec# language extends C# with contract specifications, analogously to the way JML extends Java. The Spec# compiler then introduces runtime checks for the declared specifications (akin to *jmlc*), and the Boogie program verifier tries to prove these specifications statically using an automatic theorem prover (akin to the tools described in Sect. 5). One difference between

Spec# and JML is that Spec# builds in a new methodology for object invariants [5, 7, 63], trading restrictions on the kinds of programs that can be written for a sound modular reasoning technique.

9.3 OCL: UML's constraint language

Despite the similarity in the acronyms, JML is *very* different in its aims from UML [90]. The most basic difference is that the UML aims to cover all phases of analysis and design with many notations, and it tries to be independent of programming languages, 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) [91], 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 cannot 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 under way.

In all, we believe that a language like JML, which is tailored to Java, is better suited for recording the detailed design of 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. There has been work on automatically translating OCL to JML [42].

10 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 a big 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 users 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 weighing 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.

Using any of the tools for static checking or verification requires formal specifications of the APIs of any system libraries used, and the cost of developing such specifications is very high. Indeed, the largest case study to date in using JML for specification is the ongoing work in developing specifications for substantial parts of the Java system libraries. Being able to reuse these same specifications for different tools is an important advantage.

Future work

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.

A major recent extension to JML concerns the support for different forms of arithmetic, providing normal mathematical integers in addition to Java's n-bit 2's-complements integers [18].

One important aspect of future work is experimenting with the use of JML for specification of real-world code and APIs, and using the associated tools. There has been considerable work on producing JML specifications of the Java system libraries (these can be downloaded from `www.jmlspecs.org`), but more work is needed.

Using JML to specify real-world code raises many interesting issues. For instance, JML allows pure methods to be used in annotations, where pure methods are defined as those that have no side effects. But this is a very strict definition, which can be impractical when writing specifications, as many methods (including some in core Java libraries) that programmers intuitively assume to be pure are not pure, due to unobservable and benevolent side effects [59]. Work continues on a better and more useful definition of purity, e.g., [8].

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. One thing that has become very clear in the course of developing JML is that precisely defining the semantics of a specification language such as JML is very tricky.

More generally, there are several fundamental issues in the specification of object-oriented systems that are still active topics of investigation. The notion of object invariant is tricky in the presence of callbacks [5, 7, 63, 77]. Another largely open issue is how concurrency properties should be specified.

As always in imperative programming, aliasing is a major source of complications, and an important source of bugs. For example, in the example in Fig. 1 it is probably important that in the constructor the field `pin` is not simply aliased to the argument `p` but that a new array is created. However, the current specification does not demand this. JML should offer practical ways to constrain potential aliasing. A first proposal is given in [76].

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 coauthors, still more people have been involved in developing the tools discussed in this paper, including Joachim van den Berg, Abhay Bhorkar, Kristina Boysen, Cees-Bart Breunesse, Néstor Cataño, Patrice Chalin, Curtis Clifton, Kui Dai, Werner Dietl, Marko van Dooren, Cormac Flanagan, Mark Lillibridge, Marieke Huisman, Bart Jacobs, Jean-Louis Lanet, Todd Millstein, Peter Mueller, Greg Nelson, Jeremy Nimmer, Carlos Pacheco, Arun Raghavan, Antoine Requet, Frederic Rioux, Clyde Ruby, Jim Saxe, Raymie Stata, Roy Tan, 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. Ahrendt W, Baar T, Beckert B, Bubel R, Giese M, Hähnle R, Menzel W, Mostowski W, Roth A, Schlager S, Schmitt PH (2004) The KeY tool. Softw Syst Model (in press)
2. Amey P, Chapman R (2002) Industrial strength exception freedom. In: ACM SigAda 2002, pp 1–9
3. Antoy S, Hamlet D (2000) Automatically checking an implementation against its formal specification. IEEE Trans Softw Eng 26(1):55–69
4. Barnes J (2003) High integrity software: the SPARK approach to safety and security. Addison-Wesley, Reading, MA
5. Barnett M, DeLine R, Fähndrich M, Leino KRM, Schulte W (2004) Verification of object-oriented programs with invariants. J Object Technol 3(6):27–56
6. Barnett M, Leino KRM, Schulte W (2004) The Spec# programming system: An overview. In: Construction and analysis of safe, secure and interoperable smart devices (CASSIS). Lecture notes in computer science, vol . Springer, Berlin Heidelberg New York (in press)
7. Barnett M, Naumann D (2004) Friends need a bit more: maintaining invariants over shared state. In: Kozen D (ed) Mathematics of program construction. Lecture notes in computer science, vol 3125. Springer, Berlin Heidelberg New York, pp 54–84
8. Barnett M, Naumann DA, Schulte W, Sun Q (2004) 99.44% pure: useful abstractions in specifications. In: Formal techniques for Java-like programs. Proceedings of the ECOOP'2004 workshop. Technical Report NIII-R0426, University of Nijmegen, pp 11–18
9. Bartetzko D, Fischer C, Möller M, Wehrheim H (2001) Jass – Java with assertions. In: Havelund K, Rosu G (eds) Workshop on runtime verification at CAV'01. Electronic notes in theoretical computer science, vol 55(2)
10. Beck K, Gamma E (1998) Test infected: programmers love writing tests. Java Rep 3(7):37–50
11. van den Berg J, Jacobs B (2001) The LOOP compiler for Java and JML. In: Margaria T, Yi W (eds) TACAS'01. Lecture notes in computer science, vol 2031. Springer, Berlin Heidelberg New York, pp 299–312

12. Breunesse C-B, van den Berg J, Jacobs B (2002) Specifying and verifying a decimal representation in Java for smart cards. In: Kirchner H, Ringeissen C (eds) AMAST'02. Lecture notes in computer science, vol 2422. Springer, Berlin Heidelberg New York, pp 304–318
13. Breunesse C-B, Cataño N, Huisman M, Jacobs B (2003) Formal methods for smart cards: an experience report. Technical report, University of Nijmegen. NIII Technical Report NIII-R0316.
14. Brun Y, Ernst MD (2004) Finding latent code errors via machine learning over program executions. In: Proceedings of the 26th international conference on software engineering (ICSE'04), Edinburgh, UK, 26–28 May 2004
15. Burdy L, Requet A, Lanet J-L (2003) Java applet correctness: a developer-oriented approach. In: Mandrioli D, Araki K, Gnesi S (ed) FME 2003. Lecture notes in computer science, vol 2805. Springer, Berlin Heidelberg New York, pp 422–439
16. Cataño N, Huisman M (2002) Formal specification of Gemplus's electronic purse case study. In: Eriksson LH, Lindsay PA (eds) FME 2002. Lecture notes in computer science, vol 2391. Springer, Berlin Heidelberg New York, pp 272–289
17. Cataño N, Huisman M (2003) CHASE: A static checker for JML's assignable clause. In: Zuck LD, Attie PC, Cortesi A, Mukhopadhyay S (eds) VMCAI: Verification, model checking, and abstract interpretation. Lecture notes in computer science, vol 2575. Springer, Berlin Heidelberg New York, pp 26–40
18. Chalin P (2004) JML support for primitive arbitrary precision numeric types: definition and semantics. J Object Technol 3(6):57–79
19. Cheon Y (2003) A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April. Author's PhD dissertation. `archives.cs.iastate.edu`
20. Cheon Y, Leavens GT (1994) The Larch/Smalltalk interface specification language. ACM Trans Softw Eng Methodol 3(3):221–253
21. Cheon Y, Leavens GT (2002) A runtime assertion checker for the Java Modeling Language (JML). In: Arabnia HR, Mun Y (eds) International conference on software engineering research and practice (SERP '02). CSREA Press, Las Vegas, pp 322–328
22. Cheon Y, Leavens GT (2002) A simple and practical approach to unit testing: the JML and JUnit way. In: Magnusson B (ed) ECOOP 2002. Lecture notes in computer science, vol 2374. Springer, Berlin Heidelberg New York, pp 231–255
23. Cheon Y, Leavens GT, Sitaraman M, Edwards S (2003) Model variables: cleanly supporting abstraction in design by contract. Technical Report 03-10, Department of Computer Science, Iowa State University, Ames, Iowa, April 2003
24. Clifton C (2001) MultiJava: design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, November 2001. Available from `www.multijava.org`
25. Cok DR (2004) Reasoning with specifications containing method calls in jml. In: Formal techniques for Java-like programs. Proceedings of the ECOOP'2004 Workshop. Technical Report NIII-R0426, University of Nijmegen, The Netherlands, pp 41–48
26. Detlefs D, Nelson G, Saxe JB (2003) Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003
27. Detlefs DL, Leino KRM, Nelson G, Saxe JB (1998) Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998
28. Dhara KK, Leavens GT (1996) Forcing behavioral subtyping through specification inheritance. In: 18th international conference on software engineering. IEEE Press, New York, pp 258–267
29. Dodoo N, Donovan A, Lin L, Ernst MD (2002) Selecting predicates for implications in program analysis, 16 March 2002. Draft. `http://pag.lcs.mit.edu/~mernst/pubs/invariants-implications.ps`
30. Dodoo N, Lin L, Ernst MD (2003) Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, 21 July 2003
31. Ernst MD (2000) Dynamically Discovering Likely Program Invariants. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, WA
32. Ernst MD, Cockrell J, Griswold WG, Notkin D (2001) Dynamically discovering likely program invariants to support program evolution. IEEE Trans Softw Eng 27(2):1–25
33. Ernst MD, Czeisler A, Griswold WG, Notkin D (2000) Quickly detecting relevant program invariants. In: Proceedings of the 22nd international conference on software engineering (ICSE 2000), pp 449–458
34. Flanagan C, Joshi R, Leino KRM (2001) Annotation inference for modular checkers. Inf Process Lett 77(2–4):97–108
35. Flanagan C, Leino KRM (2001) Houdini, an annotation assistant for ESC/Java. In: Oliveira JN, Zave P (eds) FME 2001. Lecture notes in computer science, vol 2021. Springer, Berlin Heidelberg New York, pp 500–517
36. Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R (2002) Extended static checking for Java. In: ACM SIGPLAN 2002 conference on programming language design and implementation (PLDI'2002), pp 234–245
37. Flanagan C, Saxe JB (2001) Avoiding exponential explosion: generating compact verification conditions. In: Conference record of the 28th annual ACM symposium on principles of programming languages, January 2001. ACM Press, New York, pp 193–205,
38. Friendly L (1995) The design of distributed hyperlinked programming documentation. In: Fraïssè S, Garzotto F, Isakowitz T, Nanard J, Nanard M (eds) IWHD'95. Springer, Berlin Heidelberg New York, pp 151–173
39. Groce A, Visser W (2003) What went wrong: explaining counterexamples. In: 10th international SPIN workshop on model checking of software, Portland, OR, 9–10 May 2003, pp 121–135
40. Gupta N, Heidepriem ZV (2003) 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, 8–10 October 2003
41. Guttag JV, Horning JJ (1993) Larch: languages and tools for formal specification. Springer, Berlin Heidelberg New York
42. Hamie A (2004) Translating the Object Constraint Language into the Java Modeling Language. In: Proceedings of the 2004 ACM symposium on applied computing (SAC'2004). ACM Press, New York, pp 1531–1535
43. Hangal S, Lam MS (2002) Tracking down software bugs using automatic anomaly detection. In: Proceedings of the 24th international conference on software engineering (ICSE'02), Orlando, FL, 22–24 May 2002, pp 291–301
44. Harder M, Mellen J, Ernst MD (2003) Improving test suites via operational abstraction. In: Proceedings of the 25th international conference on software engineering (ICSE'03), Portland, OR, 6–8 May 2003, pp 60–71
45. Henkel J, Diwan A (2003) Discovering algebraic specifications from Java classes. In: 15th European conference on object-oriented programming (ECOOP 2003), Darmstadt, Germany, 23–22 July 2003
46. Jacobs B, Kiniry J, Warnier M (2003) Java program verification challenges. In: FMCO 2002. Lecture notes in computer science, vol 2852. Springer, Berlin Heidelberg New York, pp 202–219
47. Jacobs B (2004) Weakest precondition reasoning for Java programs with JML annotations. J Logic Algebr Programm 58(1–2):61–88
48. Jacobs B, Oostdijk M, Warnier M (2004) Source code verification of a secure payment applet. J Logic Algebr Programm 58(1–2):107–120
49. Jacobs B, Poll E (2001) A logic for the Java Modeling Language JML. In: Hussmann H (ed) Fundamental approaches to software engineering (FASE). Lecture notes in computer science, vol 2029. Springer, Berlin Heidelberg New York, pp 284–299

50. Jacobs B, Poll E (2004) Java program verification at Nijmegen: developments and perspective. In: International symposium on software security (ISSS'2003). Lecture notes in computer science, vol 3233. Springer, Berlin Heidelberg New York, pp 134–153
51. Jacobs B, van den Berg J, Huisman M, van Berkum M, Hensel U, Tews H (1998) Reasoning about Java classes (preliminary report). In: OOPSLA'98, ACM SIGPLAN Notices. ACM Press, New York, 33(10):329–340
52. Jones CB (1990) Systematic Software Development Using VDM. International series in computer science, 2nd edn. Prentice-Hall, Englewood Cliffs, NJ
53. Kataoka Y, Ernst MD, Griswold WG, Notkin D (2001) Automated support for program refactoring using invariants. In: Proceedings of the international conference on software maintenance (ICSM 2001), Florence, Italy, 6–10 November 2001, pp 736–743
54. Kiniry JR, Cok DR (2004) ESC/Java2: Uniting ESC/Java and JML: progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In: Construction and analysis of safe, secure and interoperable smart devices (CASSIS). Lecture notes in computer science, vol . Springer, Berlin Heidelberg New York (in press)
55. Kramer R (1998) iContract – the Java design by contract tool. TOOLS 26: Technology of object-oriented languages and systems, Los Alamitos, CA, pp 295–307
56. Leavens GT (1996) An overview of Larch/C++: behavioral specifications for C++ modules. In: Kilov H, Harvey W (eds) Specification of behavioral semantics in object-oriented information modeling, Chap 8. Kluwer, Boston, pp 121–142. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa
57. Leavens GT, Baker AL, Ruby C (1999) JML: A notation for detailed design. In: Kilov H, Rumpe B, Simmonds I (eds) Behavioral specifications of businesses and systems. Kluwer, Boston, pp 175–188
58. Leavens GT, Baker AL, Ruby C (2003) Preliminary design of JML: a behavioral interface specification language for Java. Technical Report 98-06u, Department of Computer Science, Iowa State University, Ames, IA, April 2003
59. Leavens GT, Cheon Y, Clifton C, Ruby C, Cok DR (2003) How the design of JML accommodates both runtime assertion checking and formal verification. In: FMCO 2002. Lecture notes in computer science, vol 2852. Springer, Berlin Heidelberg New York, pp 262–284. Also appears as technical report TR03-04, Department of Computer Science, Iowa State University, Ames, IA
60. Leino KRM (2000) Extended static checking: A ten-year perspective. In: Wilhelm R (ed) Informatics – 10 years back, 10 years ahead. Lecture notes in computer science, vol 2000. Springer, Berlin Heidelberg New York
61. Leino KRM (2004) Efficient weakest preconditions. Technical Report MSR-TR-2004-34, Microsoft Research, Redmond, WA, April 2004
62. Leino KRM, Millstein T, Saxe JB (2004) Generating error traces from verification-condition counterexamples. Sci Comput Programm (in press)
63. Leino KRM, Müller P (2004) Object invariants in dynamic contexts. In: 18th European conference object-oriented programming, (ECOOP 2004), Olso, Norway, 16–18 June 2004, pp 491–516
64. Leino KRM, Nelson G, Saxe JB (2000) ESC/Java user's manual. Technical Note 2000-002, Compaq SRC, October
65. Leino KRM, Saxe JB, Stata R (1999) Checking Java programs via guarded commands. Technical Note 1999-002, Compaq SRC, May
66. Liblit B, Aiken A, Zheng AX, Jordan MI (2003) Bug isolation via remote program sampling. In: Proceedings of the ACM SIGPLAN 2003 conference on programming language design and implementation, San Diego, 9–11 June 2003, pp 141–154
67. Lin L, Ernst MD (2004) Improving adaptability via program steering. In: Proceedings of the 2004 international symposium on software testing and analysis (ISSTA 2004), Boston, 12–14 July 2004
68. Liskov B, Wing J (1994) A behavioral notion of subtyping. ACM Trans Programm Lang Syst 16(6):1811–1841
69. Marché C, Paulin-Mohring C, Urbain X (2004) The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. J Logic Algebr Programm 58(1–2):89–106
70. Mariani L, Pezzè M (2004) A technique for verifying component-based software. In: International workshop on test and analysis of component based systems, Barcelona, Spain, 27–28 March 2004
71. McCamant S, Ernst MD (2003) Predicting problems caused by component upgrades. In: Proceedings of the 10th European conference on software engineering and the 11th ACM SIGSOFT symposium on the foundations of software engineering, Helsinki, Finland, 3–5 September 2003, pp 287–296
72. McCamant S, Ernst MD (2004) Early identification of incompatibilities in multi-component upgrades. In: 18th European conference on object-oriented programming, (ECOOP 2004), Olso, Norway, 16–18 June 2004
73. Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice-Hall, Englewood Cliffs, NJ
74. Meyer J, Poetzsch-Heffter A (2000) An architecture for interactive program provers. In: Graf S, Schwartzbach M (eds) TACAS'00. Lecture notes in computer science, vol 1785. Springer, Berlin Heidelberg New York, pp 63–77
75. Morgan C (1994) Programming from specifications, 2nd edn. Prentice-Hall International, Hempstead, UK
76. Müller P, Poetzsch-Heffter A, Leavens GT (2003) Modular specification of frame properties in JML. Concurrency Comput Pract Experience 15(2):117–154
77. Müller P, Poetzsch-Heffter A, Leavens GT (2003) Modular invariants for object structures. Technical Report 424, ETH Zurich, October
78. Ne Win T, Ernst MD (2002) Verifying distributed algorithms via dynamic analysis and theorem proving. Technical Report 841, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, 25 May 2002
79. Ne Win T, Ernst MD, Garland SJ, Kırlı D, Lynch N (2004) Using simulated execution in verifying distributed algorithms. Int J Softw Tools Technol Transfer 6(1):67–76
80. Nimmer JW, Ernst MD (2002) Automatic generation of program specifications. In: International symposium on software testing and analysis (ISSTA 2002), Rome, Italy, pp 232–242
81. Nimmer JW, Ernst MD (2002) Invariant inference for static checking: an empirical evaluation. In: ACM SIGSOFT 10th international symposium on the foundations of software engineering (FSE 2002), pp 11–20
82. Owre S, Rajan S, Rushby JM, Shankar N, Srivas M (1996) PVS: Combining specification, proof checking, and model checking. In: Alur R, Henzinger TA (eds) Computer aided verification. Lecture notes in computer science, vol 1102. Springer, Berlin Heidelberg New York, pp 411–414
83. Perkins JH, Ernst MD (2004) Efficient incremental algorithms for dynamic detection of likely invariants. In: ACM SIGSOFT 12th international symposium on the foundations of software engineering (FSE 2004), Newport Beach, CA, November 2004
84. Peters DK, Lorge Parnas D (1998) Using test oracles generated from program documentation. IEEE Trans Softw Eng 24(3):161–173
85. Poll E, Hartel P, de Jong E (2002) A Java reference model of transacted memory for smart cards. In: Conference on smart card research and advanced application (CARDIS'2002). USENIX, pp 75–86
86. Poll E, van den Berg J, Jacobs B (2001) Formal specification of the Java Card API in JML: the APDU class. Comput Netw 36(4):407–421
87. Pytlik B, Renieris M, Krishnamurthi S, Reiss SP (2003) Automated fault localization using potential invariants. In: 5th international workshop on automated and algorithmic debugging (AADEBUG'2003), Ghent, Belgium, 8–10 September 2003
88. Raghavan AD (2000) Design of a JML documentation generator. Technical Report 00-12, Department of Computer Science, Iowa State University, Ames, IA, July

89. Raz O, Koopman P, Shaw M (2002) Semantic anomaly detection in online data sources. In: Proceedings of the 24th international conference on software engineering (ICSE'02), Orlando, FL, 22–24 May 2002, pp 302–312
90. Rumbaugh J, Jacobson I, Booch G (1998) The Unified Modeling Language reference manual. Addison-Wesley, Reading, MA
91. Warmer J, Kleppe A (1999) The Object Constraint Language: precise modeling with UML. Addison-Wesley, Reading, MA
92. Xie T, Notkin D (2002) Checking inside the black box: regression fault exposure and localization based on value spectra differences. Technical Report UW-CSE-02-12-04, University of Washington Department of Computer Science and Engineering, Seattle, WA, December
93. Xie T, Notkin D (2003) Tool-assisted unit test selection based on operational violations. In: Proceedings of the 13th annual international conference on automated software engineering (ASE 2003), Montreal, 8–10 October 2003