An overview of JML tools and applications

www.jmlspecs.org

Lilian Burdy [a], Yoonsik Cheon [c,1], David Cok [b],
Michael D. Ernst [d], Joe Kiniry [e], Gary T. Leavens [c,1],
K. Rustan M. Leino [f], Erik Poll [e,2]

[a] *GEMPLUS Research Lab, Gémenos, France*

[b] *Eastman Kodak Company, R&D Laboratories, Rochester, New York, USA*

[c] *Dept. of Computer Science, Iowa State University, Ames, Iowa, USA*

[d] *MIT Lab for Computer Science, Cambridge, Massachusetts, USA*

[e] *Dept. of Computer Science, University of Nijmegen, Nijmegen, the Netherlands*

[f] *Microsoft Research, Redmond, WA, USA*

Abstract

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

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

Key words: formal methods, formal specification, Java, runtime assertion checking, static checking, program verification

1 Introduction

JML [25,26], which stands for "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 the behavior and the syntactic interface of Java code. The syntactic interface of Java code, a class or interface's method signatures, attribute types, etc., is augmented with JML annotations that more precisely indicate the correct usage of the API so

[1] Supported in part by US NSF grants CCR-0097907 and CCR-0113181.

[2] Partially supported by EU-IST project VerifiCard `www.verificard.org`

that programmers can use it as documentation. In terms of behavior, JML can detail, for example, the preconditions and postconditions for methods as well as class invariants.

An important design goal is that JML be easy to understand for any Java programmer. 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 method on users; instead, JML should be able to document existing Java programs designed in any manner.

The work on JML was started by Gary Leavens and his colleagues and students at Iowa State University, but has 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. The open, cooperative nature of the JML effort is important both for tool developers and for potential 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 tool is often the lack of familiarity with the associated specification language.

The next section introduces the JML notation. Section 3 then discusses the tools for JML in more detail. Section 4 discusses the applications of JML in the domain of Java Card, the Java dialect for programming smartcards. Section 5 discusses some related languages and tools, such as OCL and other runtime assertion checkers, and Section 6 concludes.

2 The JML notation

JML blends Eiffel's design-by-contract approach [33] with the Larch [19] tradition (and others that space precludes mentioning). Because JML supports quantifiers such as `\forall` and `\exists`, and because JML allows "model" (i.e., specification-only) fields, specifications can more easily be made more precise and complete than those typically given in Eiffel. JML uses Java's expression syntax in assertions, thus JML's notation is easier for programmers to learn than one based on a language-independent specification language like the Larch Shared Language [26,27] or OCL [41].

Figure 1 gives an example of a JML specification that illustrates its main features. JML assertions are written as special 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 such comments JML extends the Java syntax with several keywords—in the example in Figure 1, `invariant`, `requires`, `assignable`, `ensures`, and `signals`. It also extends Java's expression syntax with several operators—in the example `\forall`,

```
public class Purse {

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

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

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

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

Figure 1. Example JML specification

`\old`, and `\result`; 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 (class and interface) invariants. These are all expressed as boolean expressions in JML's extension to Java's expression syntax.

In addition to "normal" postconditions, JML also supports "exceptional" postconditions, specified in `signals` clauses. These can be used to specify what must be true when a method throws an exception. For example, the `signals` clause in Figure 1 specifies that `debit` may throw a `PurseException`, and, in that case, the balance will not change (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 to only the `balance` field. Such frame conditions are essential for verification of code when using some of the tools described later.

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

- To allow specifications to be abstractions of implementation details, JML provides model variables, which play the role of abstract values for abstract data types [12].

  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.
- To support specifications that need mathematical concepts, such as sets or sequences, JML comes with an extensible Java library that provides these notions. Thus, basic mathematical notions can be used in assertions as if they were Java objects.
- A method can be used in assertions only if it is declared as `pure`, meaning the method does not have any visible side-effects For example, if there is a method `getBalance()` that is declared as `pure`,

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

  then this method can be used in the specification instead of the field `balance`.
- Finally, JML supports all the Java modifiers (`private`, `protected`, and `public`) for restricting visibility. For example, invariants can be declared as `private` invariants if they are only concerned with the private implementation details of a class and are not observable for clients.

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. [3]

[3] Most of these tools rely upon an underlying tool that typechecks a Java program and its JML specification.

- **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. Runtime assertion checking is accomplished using the JML compiler `jmlc` (Section 3.2).
  - Given that one often wants to do runtime assertion checking in the testing phase, there is also a `jmlunit` tool (Section 3.3) which combines runtime assertion checking with unit testing.
- **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. Verifying a specification gives more assurance in the correctness of code as it establishes the correctness for all possible code paths, whereas runtime assertion checking is limited by the code 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. A verification tool must trade off the level of automation it offers (i.e., the amount of user interaction it requires) and the complexity of the properties and code that it can handle. There are several tools for statically checking or verifying JML assertions providing different levels of automation and supporting different levels of expressivity in specifications.
  - The program checker ESC/Java (Section 3.5) [17] can automatically detect certain common errors in Java code and check relatively simple assertions.
  - The program checker JACK (Section 3.8) offers a similar functionality to ESC/Java, but is more ambitious in attempting real program verification.
  - The LOOP tool (Section 3.7) translates JML-annotated code to proof obligations that one can then try to prove using the theorem prover PVS. The LOOP tool can handle more complicated specifications and code, but at the price of more user interaction.
  - The CHASE tool (Section 3.6) checks some aspects of frame conditions.
- **Tools to help generate 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 of producing JML specifications.
  - The Daikon tool (Section 3.4, [14]) infers likely invariants by observing the runtime behavior of a program.
  - The Houdini tool [16] uses ESC/Java to infer annotations for code.
  - The `jmlspec` tool can produce a skeleton of a specification file from Java source and can compare the interfaces of two different files for consistency.
- **Documentation**: Finally, in spite of all the tools mentioned above, the most important 'tool' of all is still the human who reads and writes JML specifications. The `jmldoc` tool (Section 3.1) produces browsable HTML from JML specifications, in the style of Javadoc. The resulting HTML is a

convenient format for reading larger bodies of JML specifications.

3.1 Documentation Generation

Since JML specifications are 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. Consequently, the `jmldoc` tool (authored by David Cok) produces browsable HTML pages containing both the API and the specifications for Java code, in the style of pages generated by Javadoc [18].

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

3.2 Runtime Assertion Checking

The JML compiler (`jmlc`), developed at Iowa State University, is an extension to a Java compiler and compiles Java programs annotated with JML specifications into Java bytecode [10]. The compiled bytecode includes runtime assertion checking instructions that check JML specifications such as preconditions, normal and exceptional postconditions, invariants, and history constraints. The execution of such assertion checks is transparent in that, unless an assertion is violated, and except for performance measures (time and space), the behavior of the original program is unchanged. The transparency of runtime assertion checking is guaranteed, as JML assertions are not allowed to have any side-effects [27].

The JML language provides a rich set of specification facilities to write abstract, complete behavioral specifications of Java program modules [27]. It opens a new possibility in runtime assertion checking by supporting abstract specifications written in terms of specification-only declarations such as model fields, ghost fields, and model methods. Thus the JML compiler represents a significant advance over the state of the art in runtime assertion checking as represented by design by contract tools such as Eiffel [33] or by Java tools such as iContract [24] or Jass [3]. The `jmlc` tool also supports advances such as (stateful) interface specifications, multiple inheritance of specifications, various forms of quantifiers and set comprehension notation, support for strong and weak behavioral subtyping [30,13], and a contextual interpretation of undefinedness [27].

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.

3.3 Unit Testing

A formal specification can be viewed as a test oracle [37,2], and a runtime assertion checker can be used as the decision procedure for the test oracle [11]. This idea has been implemented as a unit testing tool for Java (`jmlunit`), by combining JML with the popular unit testing tool JUnit for Java [4]. The `jmlunit` tool, developed at Iowa State University, frees the programmer from writing most unit test code and significantly automates unit testing of Java classes and interfaces.

The tool generates JUnit test classes that rely on the JML runtime assertion checker. The test classes send messages to objects of the Java classes under test. The testing code catches assertion violation errors from such method calls to decide if the test data violate the precondition of the method under test; such assertion violation errors do not constitute test failures. When the method under test satisfies its precondition, but otherwise has an assertion violation, then the implementation failed to meet its specification, and hence the test data detects a failure [11]. In other words, the generated test code serves as a test oracle whose behavior is derived from the specified behavior of the class being tested. The user is still responsible for generating test data; however the test classes make it easy for the user to supply such test data. In addition, the user can supply hand-written JUnit test methods if desired.

Our experience shows that the tool allows us to perform unit testing with minimal coding effort and detects many kinds of errors. Ironically, about half of our test failures were caused by specification errors, which shows that the approach is also useful for debugging specifications. In addition, the tool can report 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.

3.4 Invariant Detection with Daikon

Most tools that work with JML assume the existence of a JML specification, then verify code against the specification. Writing the JML specification is left to a programmer. Because this task can be time-consuming, tedious, and error-prone, the Daikon invariant detector [14] provides assistance in creating a specification. Daikon outputs observed program properties in JML form and automatically inserts them into a target program.

The Daikon tool dynamically detects likely program invariants. In other words, given program executions, it reports properties that were true over those executions. It operates by observing values that a program computes at runtime, generalizing over those values, and reporting the resulting properties. Like any dynamic analysis, the technique is not sound: 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 suppress false positives [15]. 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.

Even with modest test suites, Daikon's output is highly accurate. In one set of experiments [34], 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 [35], 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.

3.5 Extended Static Checking with ESC/Java

The ESC/Java tool [17], developed at Compaq Research, performs what is called "extended static checking", static checking that goes well beyond type checking. It can check relatively simple assertions and can check for certain kinds of common errors in Java code, such as dereferencing `null` or indexing an array outside its bounds. The tool is fully automated, and the user interacts with it only by supplying JML annotations in the code. Thus, the user never sees the theorem prover that is doing the actual work behind the scenes.

An interesting property of ESC/Java is that it is neither sound nor complete. This is a deliberate design choice: the aim is to maximize the useful feedback the tool can give—i.e., the number of potential bugs in the code it will point out—fully automatically, without requiring full functional specifications.

In a sense, ESC/Java uses JML annotations to suppress spurious warning messages. For example, in Figure 1, the conjunct `p != null` in constructor's precondition causes ESC/Java not to warn about the possibility of null dereferences on the formal parameter `p` in the body of the constructor. Such annotations also cause ESC/Java to perform additional checks. For example, the precondition on the constructor of `Purse` causes ESC/Java to emit warnings when the actual parameter passed as `p` to the constructor may be null. Thus, the addition of JML annotations helps give better quality warnings, the use of ESC/Java fosters more annotations, and in turn these annotations help the tool do a better job of checking code for potential errors.

ESC/Java works by translating a given JML-annotated Java program into verification conditions [29], which are then passed to an automatic theorem prover. The ESC/Java User's Manual [28] provides a detailed description of the semantics of JML annotations, as they pertain to ESC/Java.

As of this writing, there are some small syntactic and semantic differences between JML and the subset of JML supported by ESC/Java. Kiniry and Cok are working on patches to fix this problem. The next version of ESC/Java will accept all notation that has been introduced since ESC/Java's last release, and it will ignore all non-critical annotations it does not understand.

3.6 CHASE

One source of unsoundness of ESC/Java is that it does not check `assignable` clauses. The semantics of these frame axioms are also not checked by the JML compiler. The CHASE tool [9] tries to remedy these problems. It performs a syntactic check on `assignable` clauses, which, in the spirit of ESC/Java, is neither sound nor complete, but which spots many mistakes made in `assignable` clauses. This is another example of the utility of a common language; developers can reap the benefits of complementary tools.

3.7 Program Verification with LOOP

The University of Nijmegen's LOOP tool [23,20] translates JML-annotated Java code into proof obligations for the theorem prover PVS [36]. One can then try to prove these obligations, interactively, in PVS. The translation from JML to formal proof obligations builds on a formal semantics for sequential Java that has has been formalized in PVS, and which has been extended to provide a formal semantics for a large part of JML. The verification of the proof obligations is accomplished using a Hoare Logic [22] and a weakest-precondition calculus [21] for Java and JML. Interactive theorem proving is very labor-intensive, but allows verification of more complicated properties than can be handled by extended static checking with ESC/Java. A recent paper describing a case study with the LOOP tool, giving the best impression of the state of the art, is now available [5].

A similar program verification tool for JML-annotated code under development is the Krakatoa tool [31]; it produces proof obligations for the theorem prover Coq, but currently covers only a subset of Java.

3.8 Static Verification with JACK

The JACK [7] tool has been developed at the research lab of Gemplus, a manufacturer of smartcards and smartcard software. JACK aims to provide an environment for Java and Java Card program verification using JML annotations. It implements a fully automated weakest precondition calculus in order to generate proof obligations from JML-annotated Java sources. Those proof obligations can then be discharged using a theorem prover. Currently the proof obligations are generated for the B [1] method's prover.

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

An important design goal of the JACK tool is to be usable by normal Java developers, allowing them to validate their own code. Thus, care has been taken to hide the mathematical complexity of the underlying concepts, and JACK provides a dedicated proof obligation viewer. This viewer presents the proof obligations as execution paths within the program, highlighting the source code relevant to the proof obligations. Moreover, goals and hypotheses are displayed in a Java/JML-like notation. To allow developers to work in a familiar environment, JACK is integrated as a plug-in in the Eclipse[4] IDE.

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

Despite using formal techniques, the goal of JACK is not only to allow formal methods experts to prove Java applet correctness, but also to allow Java programmers to obtain high assurance of code correctness. This may be a way to let non-experts venture into the formal world.

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

4 Applications of JML to Java Card

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

Java Card is a well-suited target for the application of formal methods. It is a relatively simple language with a restricted API. Moreover, Java Card programs, called "applets", are small, typically on the order of several KBytes of bytecode. Additionally, correctness of Java Card programs is of crucial importance, since they are used in sensitive applications such as bank cards and mobile phone SIMs. (An interesting overview of security properties that are relevant for Java Card applications is available [32].)

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 [39]. This experience has shown that JML is expressive enough to specify a non-trivial existing API. The runtime assertion checker has been used to specify and verify a component of a smartcard operating system [38].

ESC/Java has been used with great success to verify a realistic example of an electronic purse implementation in Java Card [8]. 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. This 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 mentioned above has provided the most serious case study to date with the LOOP tool [5].

5 Related Work

5.1 Other runtime assertion checkers

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

5.2 JML vs. OCL

Despite the similarity in the acronyms, JML is *very* different in its aims from UML [40]. Unlike UML, which attempts to have notations for all phases of the analysis and design process, JML has the much more modest aim of describing the behavior of Java classes and interfaces and recording detailed design and implementation decisions.

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

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

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

rather than the OCL:

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

JML supports all the Java modifiers such as `static`, `private`, `public`, etc., and these can be used to record detailed design decisions. 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. In fact, a semantics for OCL was only recently proposed [6].

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

6 Conclusions

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

(i) JML is *easy to learn* for any Java programmer, since its syntax and semantics are very close to Java.

We believe this a crucial advantage, as the biggest hurdle to introducing formal methods in industry is often that people are not willing, or do not have the time, to learn yet another language.

(ii) 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 two 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 (e.g., [5,8,39]) have involved existing APIs and code.

(iii) There is a (growing) availability of a wide range of tool support for JML.

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

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

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

More generally, there are still many open issues involving the specification of object-oriented systems. When exactly should invariants hold? How should concurrency properties be specified? JML's specification inheritance forces subtypes to be behavioral subtypes [13,26], but subtyping in Java is used for implementation inheritance as well; is it practical to always weaken the specifications of supertypes enough so that their subtypes are behavioral subtypes? There are also semantics issues with frame axioms, pure methods, and aliasing. Such subtleties are evidenced by the slightly different ways in which different tools approach these problems.

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.

Acknowledgments

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

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Sergio Antoy and Dick Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.

[3] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass — Java with assertions. In *Workshop on Runtime Verification at CAV'01*, 2001. Published in *ENTCS*, K. Havelund and G. Rosu (eds.), 55(2), 2001.

[4] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[5] Cees-Bart Breunesse, Joachim van den Berg, and Bart Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, *AMAST'02*, number 2422 in LNCS, pages 304–318. Springer, 2002.

[6] Achim D. Brucker and Burkhart Wolff. A proposal for a formal OCL semantics in Isabelle/HOL. In César Muñoz, Sophiène Tahar, and Víctor Carreño, editors, *TPHOL'02*, volume 2410 of *LNCS*, pages 99–114. Springer, 2002.

[7] Lilian Burdy, Jean-Louis Lanet, and Antoine Requet. JACK (Java Applet Correctness Kit). At `http://www.gemplus.com/smart/r_d/trends/jack.html`, 2002.

[8] Néstor Cataño and Marieke Huisman. Formal specification of Gemplus's electronic purse case study. In L. H. Eriksson and P. A. Lindsay, editors, *FME 2002*, volume LNCS 2391, pages 272–289. Springer, 2002.

[9] Néstor Cataño and Marieke Huisman. CHASE: A static checker for JML's assignable clause. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *VMCAI: Verification, Model Checking, and Abstract Interpretation*, volume 2575 of *LNCS*, pages 26–40. Springer, 2003.

[10] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, June 2002.

[11] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *ECOOP 2002*, volume 2374 of *LNCS*, pages 231–255. Springer, June 2002.

[12] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10, Department of Computer Science, Iowa State University, April 2003. Available from `archives.cs.iastate.edu`.

[13] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *18th International Conference on Software Engineering*, pages 258–267. IEEE Computer Society Press, 1996.

[14] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, 2001.

[15] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. Quickly detecting relevant program invariants. In *ICSE 2000, Proceedings of the 22nd International Conference on Software Engineering*, pages 449–458, 2000.

[16] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for esc/java. In J. N. Oliveira and P. Zave, editors, *FME 2001*, volume LNCS 2021, pages 500–517. Springer, 2001.

[17] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, pages 234–245, 2002.

[18] Lisa Friendly. The design of distributed hyperlinked programming documentation. In S. Fraïssè, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *IWHD'95*, pages 151–173. Springer, 1995.

[19] John V. Guttag, James J. Horning, et al. *Larch: Languages and Tools for Formal Specification*. Springer, New York, NY, 1993.

[20] Marieke Huisman. *Reasoning about Java Programs in higher order logic with PVS and Isabelle*. IPA dissertation series, 2001-03, University of Nijmegen, Holland, February 2001.

[21] Bart Jacobs. Weakest precondition reasoning for Java programs with JML annotations. *JLAP*, 2002. To appear.

[22] Bart Jacobs and Erik Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 2029 of *LNCS*, pages 284–299. Springer, 2001.

[23] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java classes (preliminary report). In *OOPSLA'98*, volume 33(10) of *ACM SIGPLAN Notices*, pages 329–340. ACM, October 1998.

[24] Reto Kramer. iContract – the Java design by contract tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems, Los Alamitos, California*, pages 295–307, 1998.

[25] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[26] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u, Iowa State University, Department of Computer Science, April 2003. See `www.jmlspecs.org`.

[27] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. Technical Report 03-04, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, March 2003. To appear in the proceedings of FMCO 2002.

[28] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq SRC, October 2000.

[29] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Note 1999-002, Compaq SRC, May 1999.

[30] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[31] Claude Marché, Christine Paulin, and Xavier Urbain. The Krakatoa tool for JML/Java program certification. Available at `http://krakatoa.lri.fr`, 2003.

[32] Renaud Marlet and Daniel Le Metayer. Security properties and Java Card specificities to be studied in the SecSafe project. Technical Report SECSAFE-TL-006, Trusted Logic, August 2001. Available from `http://www.doc.ic.ac.uk/~siveroni/secsafe/docs.html`.

[33] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[34] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, 2002.

[35] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 11–20, 2002.

[36] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in LNCS, pages 411–414. Springer, 1996.

[37] Dennis K. Peters and David Lorge Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[38] Erik Poll, Pieter Hartel, and Eduard de Jong. A Java reference model of transacted memory for smart cards. In *Smart Card Research and Advanced Application Conference (CARDIS'2002)*, pages 75–86. USENIX, 2002.

[39] Erik Poll, Joachim van den Berg, and Bart Jacobs. Formal specification of the Java Card API in JML: the APDU class. *Computer Networks*, 36(4):407–421, 2001.

[40] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Publishing Company, 1998.

[41] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley Publishing Company, 1999.