JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML

Patrice Chalin, Perry R. James, George Karabotsos

Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University, Montréal, Canada
{chalin, perry, g_karab}@dsrg.org

Abstract. Tool support for the Java Modeling Language (JML) is a very pressing problem. A main issue with current tools is their architecture: the cost of keeping up with the evolution of Java is prohibitively high: e.g., Java 5 has yet to be fully supported. This paper presents JML4, our proposal for an Integrated Verification Environment (IVE) for JML that builds upon Eclipse's support for Java, enhancing it with Runtime Assertion Checking (RAC), Extended Static Checking (ESC) and Full Static Program Verification (FSPV). Though it currently only supports a subset of JML, we believe that JML4 is the first IVE to support such a full range of verification techniques for a mainstream programming language.

Introduction

The Java Modeling Language (JML) is the most popular Behavioral Interface Specification Language (BISL) for Java. JML is recognized by a dozen tools and used by over two dozen institutions for teaching and/or research, mainly in the context of program verification [20]. Tools exist to support the full range of verification from Runtime Assertion Checking (RAC) to Full Static Program Verification (FSPV) with Extended Static Checking (ESC) in between [6]. In fact, JML is the only BISL supported by all three of these verification technologies.

Unfortunately, JML tools have been aging very quickly. Researchers have been unable to keep up with the rapid pace of evolution of both Java and JML. A prime example of this is the lack of support for Java 5, despite the fact that it was released in 2004. Keeping up with changes in Java is very labor intensive and from an academic researcher's point of view unrewarding.

In this paper we present JML4, an Eclipse-based Integrated development and Verification Environment (IVE) for Java and JML. Being built on top of the Eclipse Java Development Tooling (JDT), JML4 gets up-to-date support for Java almost "for free". Our contributions are as follows:

- We summarize the JML tooling state-of-affairs, reflecting upon lessons learned from the development of the first generation of tools, projecting successes into our statement of **goals** for any next generation tooling infrastructure (Section 2).
- With the purpose of illustrating progress made in achieving these goals, we describe the capabilities of JML4 (Section 3), with a particular focus on recent additions (Section 3.2) which now bring to JML4 support for the full range of verification techniques (from RAC to FSPV). This makes JML4 the first IVE for a mainstream language supporting such a full range of verification techniques.
- We describe the architecture of JML4 (Section 4), allowing the reader to determine, at an architectural level, the extent to which JML4 achieves its goals.
- An assessment of the current state of JML4 is provided (Section 5). In particular we reassess the goals and identifying risk items.

The capabilities of verification tools supporting JML as well as other languages are reviewed in the section on related work (Section 6). We conclude in Section 7.

2 Problem

First Generation Tools: duplication of effort & high (collective) maintenance overhead. JML can be seen as an extension to Java that adds support for Design by Contract (DBC) though it has more advanced features—such as specification-only class attributes, support for frame properties (indicating which parts of the system state a method must leave unchanged), and behavioral subtyping—that are essential to writing complete interface specifications [12]. The main first generation JML tools essentially consist of:

- Common JML tool suite[1] also known as JML2, which includes the JML RAC compiler and JmlUnit [6],
- ESC/Java2, an extended static checker [15], and
- LOOP and PVS tool pair which supports full static program verification [25].

Of these, JML2 is the original JML tool set. Although ESC/Java2 and LOOP initially used their own annotation languages, they rapidly switched to JML.

Being independent development efforts, each of the tools mentioned above has its own front-end (e.g. scanner, parser, abstract syntax tree (AST) hierarchy and static analysis code) essentially for *all* of Java and JML. This amounts to substantial duplication of effort and code. Recent evolution in the definition of Java (e.g. Java 5, especially generics) and of JML made it painfully evident that the limited resources of the JML community could not cope with the workload that it engendered.

As a result, for example, none of the current first generation tools can yet fully support Java 5 features. With respect to the evolution of JML, only JML2 supports the new non-null by default semantics [9].

[1] Formerly the Iowa State University (ISU) JML tool suite.

Moving Forward with Lessons Learned. What lessons can be learned from the development of the first generation of tools, especially JML2, which (since early 2000) has been the reference implementation of JML? JML2 was essentially developed as an extension to the MultiJava (MJ) compiler. By "extension", we mean that: for the most part, MJ remains independent of JML; many JML features are naturally implemented by subclassing MJ features and overriding methods; in other situations, extension points (calls to methods with empty bodies) were added to MJ classes so that it was possible to override behavior in JML2. We believe that this approach has allowed JML2 to be successfully maintained as the JML reference implementation until recently. Then what went wrong? We believe it was a combination of factors including the advent of a relatively big step in the evolution of Java (including Java 5 generics) and the difficulty in finding developers to upgrade MJ.

Goals for Next Generation Tool Bases. Keeping in mind that we are targeting mainstream industrial software developers as our primary user base, our goals for a next generation research vehicle for the JML community can be summarized as follows: the new tooling infrastructure should be [22]:

1) based on, at least a Java compiler, ideally a modern IDE, whose maintenance is *outside* the JML community;
2) built, to the extent practicable, as a decoupled "extension" of the base so as to *minimize the integration* effort required when new versions of the base compiler/IDE are released;
3) capable of supporting *at least* the integrated capabilities of RAC, ESC, and FSPV

As will be discussed in the section on related work, a few recent JML projects have attempted to satisfy these goals. In the sections that follow, we describe how we have attempted to satisfy them in our design of JML4.

3 JML4

3.1 Early Prototype

After much discussion, both within our own research group and with other members of the JML community, we decided that basing a next generation JML tooling framework on the Eclipse JDT seemed like the most promising approach. While the JDT is large—approximately 1 MLOC for 5000 files—and the learning curve is steep (partly due to lack of documentation) we nonetheless chose to take the plunge and began prototyping JML4 in 2006.

In our first feature set, JML4 enhanced Eclipse 3.3 with: scanning and parsing of nullity modifiers (`nullable` and `non_null`), enforcement of JML's non-null type system (both statically and at runtime) [9] and the ability to read and make use of the extensive JML API library specifications. This architecturally significant subset of features was chosen so as to exercise some of the basic capabilities that any JML extension to Eclipse would need to support. These include

- recognizing and processing JML syntax inside specially marked comments, both in `*.java` files as well as `*.jml` files;
- storing JML-specific nodes in an extended AST hierarchy,
- statically enforcing a modified type system, and
- generating runtime assertion checking (RAC) code.

The chosen subset of features was also seen as useful in its own right [9], somewhat independent of other JML features. In particular, the capabilities formed a natural extension to the existing embryonic Eclipse support for nullity analysis.

This early prototype served as a basis for analysis by members of the JML Reloaded "subcommittee" of the JML Consortium. In conclusion, the decision was to move forward with development of JML4 [22].

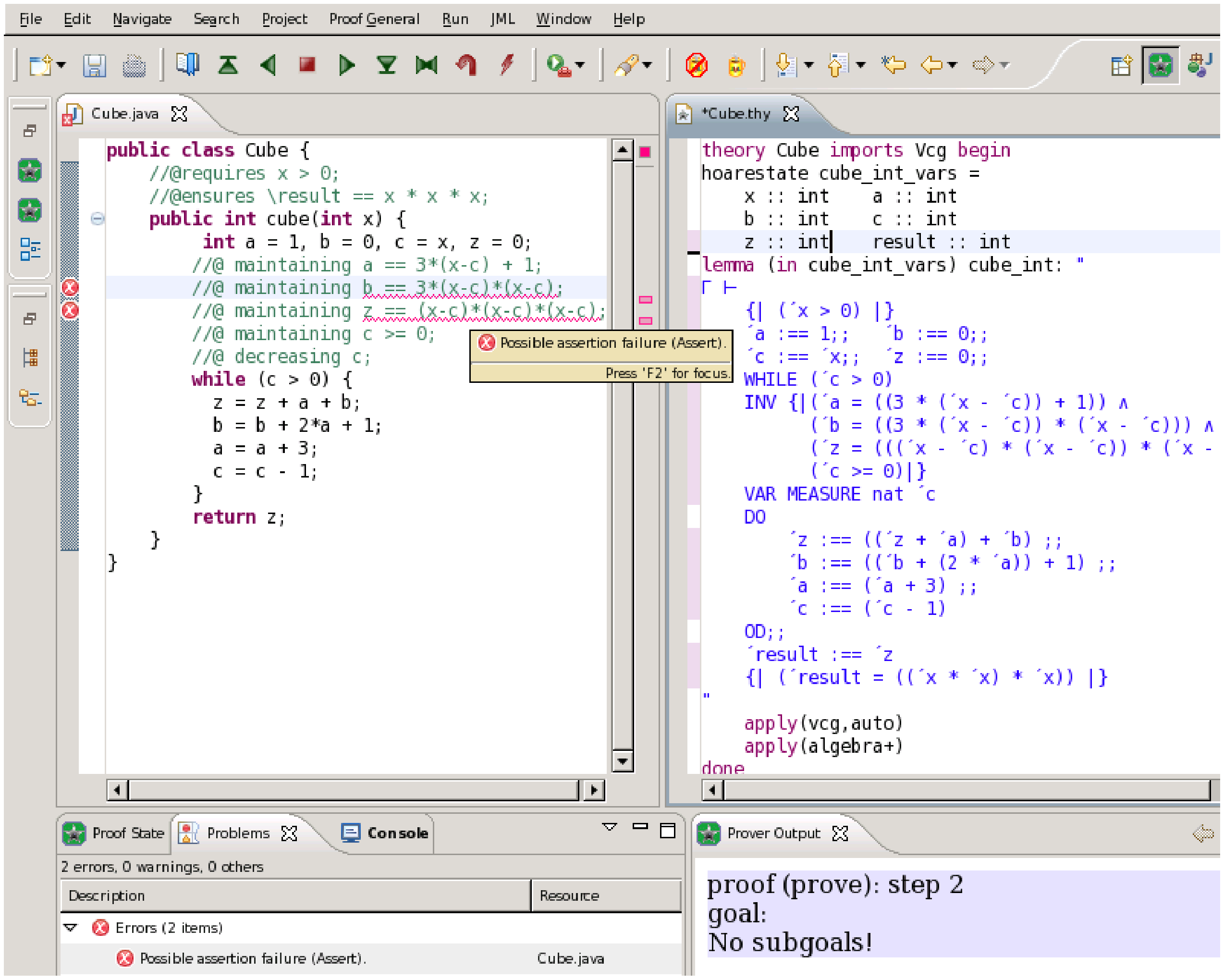
3.2 New Feature Set

In this section we describe the current JML4 feature set as evidence that progress is being made towards the goals stated in Section 2, especially with respect to framework capabilities in support of the full range of verification technologies.

3.2.1 Overview

We mention in passing that in parallel with our work on next generation components we have integrated the two main first-generation JML tools: ESC/Java2 and the JML RAC. Hence, at a minimum, JML users actively developing with first generation tools will be able to continue to do so, but now within the more hospitable environment offered by Eclipse.

With respect to the next generation components proper, at the time of writing, JML4's front-end support is nearing what is called JML Level 1, which includes the most frequently used core JML constructs [21, §2.9]. When completed, this will provide the capabilities of a type checker, similar to that provided by JML2's `jmlc`. While basic support for RAC (e.g., inline assertions and simple contracts) is available, a next generation design inspired from the current JML compiler is being lead by its original author [13], Yoonsik Cheon, and his team at the University of Texas at El Paso. Our research group is leading development in static verification components—details are given in the next section. Yet others are exploring the integration of new tools for JML; e.g. Robby and his team at Kansas State University are looking into the integration of the Bogor/Kiasan symbolic execution system and the associated KUnit test generation framework [16]. We are hopeful that next generation components fully processing JML Level 1 specifications will be ready by the end of 2008.



```
public class Cube {
    //@requires x > 0;
    //@ensures \result == x * x * x;
    public int cube(int x) {
        int a = 1, b = 0, c = x, z = 0;
        //@ maintaining a == 3*(x-c) + 1;
        //@ maintaining b == 3*(x-c)*(x-c);
        //@ maintaining z == (x-c)*(x-c)*(x-c);
        //@ maintaining c >= 0;
        //@ decreasing c;
        while (c > 0) {
          z = z + a + b;
          b = b + 2*a + 1;
          a = a + 3;
          c = c - 1;
        }
        return z;
    }
}
```

```
Possible assertion failure (Assert).
Press 'F2' for focus.
```

```
theory Cube imports Vcg begin
hoarestate cube_int_vars =
    x :: int    a :: int
    b :: int    c :: int
    z :: int    result :: int
lemma (in cube_int_vars) cube_int: "
Γ ⊢
    {| (´x > 0) |}
    ´a :== 1;;   ´b :== 0;;
    ´c :== ´x;;  ´z :== 0;;
    WHILE (´c > 0)
    INV {|(´a = ((3 * (´x - ´c)) + 1)) ∧
          (´b = ((3 * (´x - ´c)) * (´x - ´c))) ∧
          (´z = (((´x - ´c) * (´x - ´c)) * (´x -
          (´c >= 0)|}
    VAR MEASURE nat ´c
    DO
        ´z :== ((´z + ´a) + ´b) ;;
        ´b :== ((´b + (2 * ´a)) + 1) ;;
        ´a :== (´a + 3) ;;
        ´c :== (´c - 1)
    OD;;
    ´result :== ´z
    {| (´result = ((´x * ´x) * ´x)) |}
"
    apply(vcg,auto)
    apply(algebra+)
done
```

```
2 errors, 0 warnings, 0 others
Description                                   Resource
Errors (2 items)
    Possible assertion failure (Assert).      Cube.java
```

```
proof (prove): step 2
goal:
No subgoals!
```

Figure 1. ESC4 reporting that it cannot prove loop invariants in Cube.java (because it is using only first order provers**).** **FSPV TG** theory (Cube.thy) and its proof confirmed valid by Isabelle.

3.2.2 Static Verification (SV): ground-up designs using latest techniques

Besides work on the JML4 infrastructure, our research group has been focusing its efforts on the development of a new component called the JML Static Verifier (JML SV). This new component offers the basic capabilities of ESC and FSPV.

The ESC component of JML4, referred to as ESC4, is a ground-up rewrite of ESC which is based on Barnett and Leino's innovative and improved approach to a weakest precondition semantics for ESC [4]. Our FSPV tool, called the FSPV Theory Generator (TG), is like the JML LOOP compiler [25] in that it generates theories containing lemmas whose proof establish the correctness of the compilation unit in question. The FSPV TG currently generates theories written in the Hoare Logic of SIMPL—an Isabelle/HOL based theory designed for the verification of sequential imperative programs [23].

Lemmas are expressed as Hoare triples. To prove the correctness of such lemmas, a user can interactively explore their proof using the Eclipse version of Proof General (PG) [2]—see Figure 1.

3.2.3 Innovative SV features

In addition to supporting ESC and FSPV, the JML SV currently supports the following features, most of which are novel either in the context of verification tools in general or JML tools in particular:

- Multi Automated Theorem Prover (ATP) support including:
  - First-order ATPs: Simplify and CVC3.
  - Isabelle/HOL, which, we have found can be used quite effectively as an ATP.
- A technique we call 2D Verification Condition (VC) cascading where VCs that are unprovable are broken down into sub-VCs (giving us one axis of this 2D technique) with proofs attempted for each sub-VC using each of the supported ATPs (second axis).
- VC proof status caching. VCs (and sub-VCs) are self-contained, context-independent lemmas (because the lemmas' hypotheses embed their context), and hence they are ideal candidates for proof status caching. I.e., the JML SV keeps track of proven VCs and reuses the proof status on subsequent passes, matching textually VCs and hence avoiding expensive re-verification.
- Offline User Assisted (OUA) ESC, which we explain next.

By definition, ESC is fully automatic [18] whereas FSPV requires interaction with the developer. OUA ESC offers a compromise: a user is given an opportunity to provide (offline) proofs of sub-VCs which ESC4 is unable to prove automatically. Currently, ESC4 writes unprovable lemmas to an Isabelle/HOL theory file (one per compilation unit). The user can then interactively prove the lemmas using Proof General. Once this is done, ESC4 will make use of the proof script on subsequent invocations. We have found OUA ESC to be quite useful because ESC4 is generally able to automatically prove most sub-VCs, hence only asking the user to prove the ones beyond ATP abilities greatly reduces the proof burden on users.

Figure 2 sketches the relationship between the effort required to make use of each of the JML SV verification techniques vs. the level of completeness that can be achieved. Notice how ESC4, while requiring no more effort to use than its predecessor ESC/Java2, is able to achieve a higher level of completeness. This is because ESC4 makes use of multiple prover back-ends including the first order provers Simplify and CVC3 as well as Isabelle/HOL. As was mentioned earlier, Isabelle/HOL can be used quite effectively as an automated theorem prover; in fact, Isabelle is able to (automatically) prove the validity of assertions that are beyond the capabilities of the first order provers—e.g., assertions making use of quantifiers. An example of a method which JML SV can prove correct using Isabelle/HOL as an ATP is Cube.java given in Figure 1 (the reason ESC4 shows that it is unable to prove the loop invariants is because we disabled use of Isabelle/HOL as an ATP for illustrative purposes—to contrast with what can be proven using Cube.thy).

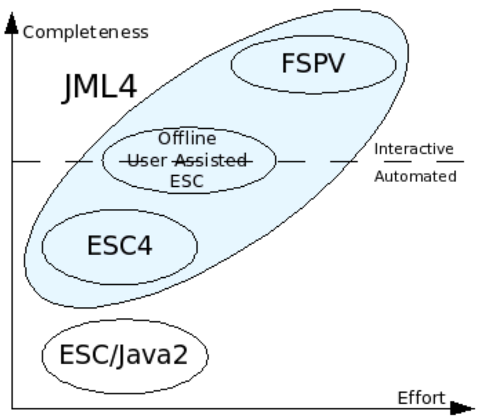


Figure 2. Static verification in JML4

In summary, the latest features added to JML4 make it the first IVE for a mainstream programming language to support the full range of verification technologies (from RAC to FSPV). Its innovative features make it easier to achieve complete verification of JML annotated Java code and this more quickly: preliminary results show that ESC4 will be at least 5 times faster than ESC/Java2. Furthermore, features like proof caching, and other forms of VC proof optimization, offer a further 50% decrease in verification time. Of course, until JML SV supports the full JML language, these results are to be taken as preliminary, but we believe that they are indicative of the kinds of efficiency improvements that can be expected.

In the next section, we explore the architecture of JML4 in general, and the JML SV in particular, allowing us to see how the features described above have been realized.

4 Architecture

In this section we present an architectural overview of JML4 with a particular focus on the compiler (rather than other aspects of the IDE) which is referred to as the JML4 *core*.

4.1 Overview

At the heart of JML4 is the JML4 core, whose processing phases are illustrated in Figure 3. Most phases are conventional. In the Eclipse JDT (and hence in JML4), there are two types of parsing: in addition to the usual full parse, there is also a diet parse, which only gathers class signature information and ignores method bodies. JML4-specific phases are shown in bold and include the merge of external specifications and static verification. Code instrumentation for the purpose of Runtime Assertion Checking (RAC) is done during the JDT's code generation phase.

A top-level module view of Eclipse and JML4 is given in Figure 4. Eclipse is a plug-in based application platform and hence an Eclipse application consists of the Eclipse plug-in loader (Platform Runtime component), certain common plug-ins (such as those in the Eclipse Platform package) along with application specific plug-ins. While Eclipse is written in Java, it does not have built-in support for Java. Like all other Eclipse features, Java support is provided by a collection of plug-ins referred to as the Eclipse Java Development Tooling (JDT).

The JML JDT extends the Eclipse JDT to offer basic support for JML. In particular, the JML JDT contains a modified scanner, parser and Abstract Syntax Tree (AST) hierarchy. The JML Static Verifier (SV) component design is described next.

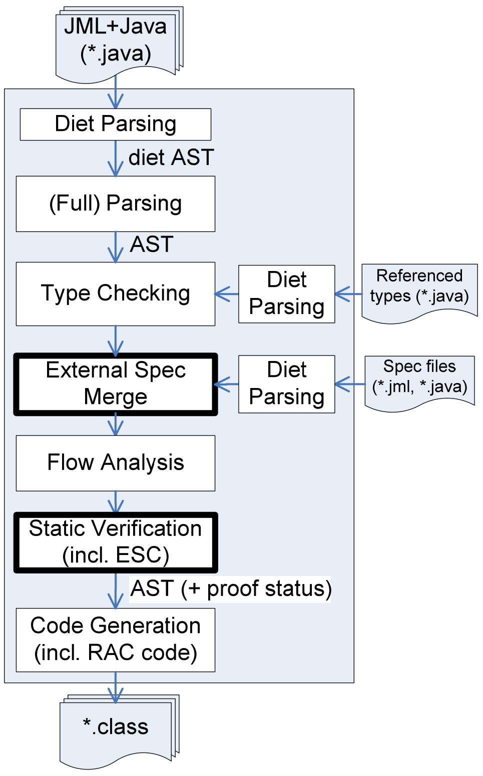


Figure 3. JDT/JML4 core phases (phases only present in JML4 are in bold)

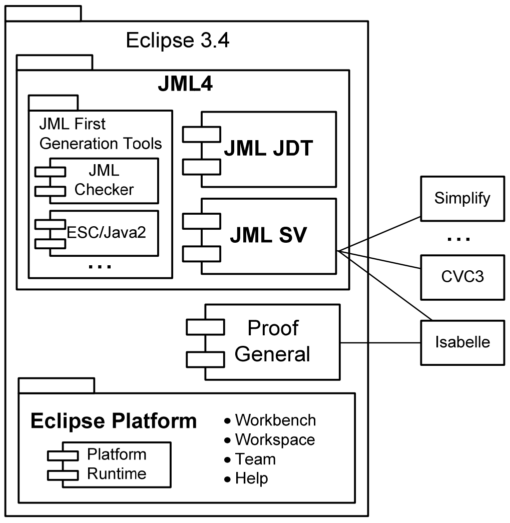


Figure 4. Eclipse and JML4 component diagram

4.2 JML Static Verifier (SV)

As was explained in the previous section, the JML SV supports two main kinds of verification: extended static checking—both the normal kind and Offline User Assisted (OUA) ESC—and full static program verification. These are realized by the subcomponents named ESC4 and the FSPV Theory Generator (TG), respectively. A diagram illustrating dataflow for the JML SV is given in Figure 5. The input to the JML SV is a fully resolved AST for the compilation unit actively being processed.

Initiated on user request, separate from the normal compilation process, the FSPV TG generates Isabelle/HOL theory files for the given compilation unit (CU). One theory file is generated per CU. The theory files can then be manipulated by the user using Proof General.

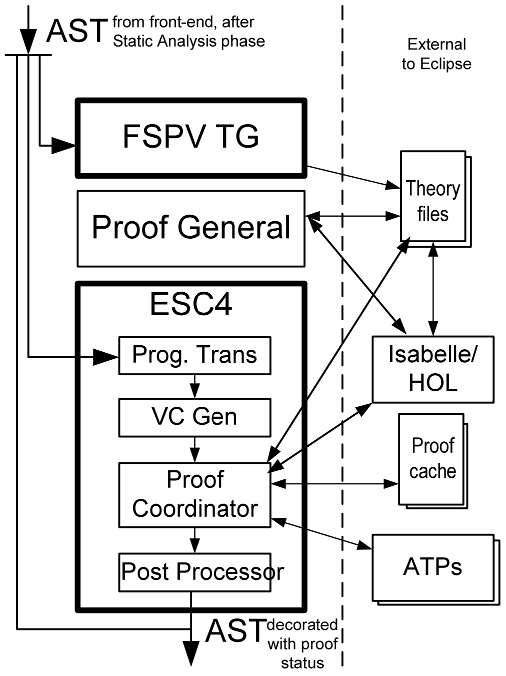


Figure 5. JML SV dataflow

When activated (via compiler preferences), ESC4 functionality kicks in during the normal compilation process following standard static analysis. The ESC phases are the standard ones [18], though the *approach* used by ESC4 is new in the context of JML tooling: it is a realization of the Barnett and Leino approach [4] used in Spec# in which the input AST is translated into VCs by using a novel form of guarded-command program as an intermediate representation. The Proof Coordinator decides on the strategy to use: e.g. single prover, cascaded VC proofs or OUA ESC. In the latter case, Isabelle theory files are consulted when sub-VCs are unprovable and a user-supplied proof exists. Unfortunately, the detailed design and full details of the behavior of the JML SV are beyond the scope of this paper.

5 Assessment

Goals. In Section 2, we listed three goals to be satisfied by any next generation tooling infrastructure for JML. In summary, (1) the infrastructure should be built as an extension to a compiler + IDE whose maintenance is guaranteed by others, (2) minimize integration efforts as the IDE code base evolves, and (3) demonstrate the feasibility of supporting the full range of current verification technologies.

The first goal has been achieved simply by our choice of the Eclipse JDT as a tool base—though doing so introduced some risks which we discuss in the next section. Since the second goal (ease of maintenance) is related to the risk items, we defer assessment of this goal to the next section as well. By the implementation of the most recent JML4 feature set we have achieved, for a non-trivial subset of JML, support for RAC, ESC and FSPV, thus demonstrating the third goal.

As evidence that JML4 is actually usable in practice, we point out that we have successfully applied it to a case study (totaling over 470K SLOC) in which we made use of the enhanced non-null-type static analysis and RAC capabilities of JML4 [9]. Furthermore, we are also currently capable of compiling first generation tools (e.g. ESC/Java2) using the first generation tools themselves within JML4.

Risk items. The Eclipse JDT is a very dynamic code base, with code changes introduced every few weeks. This was perceived as an important risk item for JML4. As we explain next, we believe that we have found a suitable means of allowing JML4 to extend the JDT so as to keep JML4 code rework to a minimum when JDT changes are released.

JML4, like JML2, is built as a closely integrated and yet (relatively) loosely coupled extension to an existing compiler. An additional benefit for JML4 is that the timely compiler-base maintenance is assured by the Eclipse Foundation developers. Hence, as compared to JML2, we have traded in committer rights for free maintenance; a choice which we believe will be more advantageous in the long run—in particular due to the rapid pace of the evolution of Java. Unfortunately, loosing committer rights means that we must maintain our own mirror of the JDT code.

While we originally had the goal of creating JML4 as a proper Eclipse plug-in, only making use of public JDT APIs (rather than as a replacement plug-in for the JDT), it rapidly became clear that this would result in far too much copy-and-change code; so much so that the advantage of coupling to an existing compiler was lost (e.g. due to the need to maintain our own full parser and AST). Nonetheless we were also originally reluctant to build atop internal APIs, which contrary to public APIs, are subject to change—with weekly releases of the JDT code, it seemed like we would be building on quicksand. Anticipating this, we established several conventions that make merging in the frequent JDT changes both easier and less error prone. These include

- avoiding introducing JML features by the copy-and-change of JDT code, instead we make use of subclassing and method extension points;
- bracketing any changes in our copy of the JDT code with special comment markers; and
- segregating all JML-specific fields and methods at the end of the file.

While following these conventions, incorporating each of the regular JDT updates since 2006 has taken less than 15 minutes, on average. While we believe that further decoupling from the JDT is possible, we feel it is too early to undertake this since we have yet to experience the integration of the full JML feature set.

One of our objectives is also to make it easy for all members of the JML research community to extend JML4, e.g., to integrate their own tools. JML developers who have been initiated to JML4 development have expressed concern in this respect due to the complexities of the JDT which developers are currently exposed to. With more experience in the development of JML4, we are hopeful that it will be possible to hide some of the unnecessary JDT complexities.

Tool Scope and moving forward. Though the JML4 infrastructure and the JML SV are at a preliminary stage of development they support, we believe, a rudimentary yet useful subset of JML. E.g., verification is implemented for the normal behavior (no exceptional behavior) of simple method contracts, without behavioral subtyping. Though we cannot use JML SV on the full ESC/Java2 source, ESC/Java2 can be compiled with JML4's static and runtime processing of nullity constraints. This in itself has proven useful in indentifying bugs with ESC/Java2 code and annotations [11].

Reengineering the first generation tool base while creating new tools and elaborating the tooling infrastructure is going to take time. Thankfully, with the increased contributions by other JML research teams, we are hopeful that feature increments will be added on a regular basis so that by year's end, JML4 will have achieved a level of support for JML that exceeds the capabilities of the first generation tools.

6 Related Work

6.1 Verification tool support for Java and/or JML

In this section we briefly compare JML4 to its sibling next generation projects JML3, JML5, JaJML as well as to the Java Applet Correctness Kit (JACK). Further details, examples and tools are covered in [10].

JML3. The first next-generation Eclipse-based initiative was JML3, created by David Cok. The main objective of the project was to create a proper Eclipse plug-in, independent of the internals of the JDT [14]. Considerable work has been done to develop the necessary infrastructure, but there are growing concerns about the long term costs of this approach.

Because the JDT's parser is not extensible from public JDT extensions points, a separate parser for the entire Java language and an AST had to be created for JML3; in addition, Cok notes that "JML3 [will need] to have its own name / type / resolver / checker for both JML constructs [and] all of Java" [14]. Since one of the main goals of the next generation tools is to escape from providing support for the Java language, this is a key disadvantage.

JACK. The Java Applet Correctness Kit (JACK) is a tool for JML annotated Java Card programs initially developed at Gemplus (2002) and then taken over by INRIA (2003) [5]. It uses a weakest precondition calculus to generate proof obligations that are discharged automatically or interactively using various theorem provers [8]. While JACK is emerging as a candidate next generation tool (offering features unique to JML tools such as

Table 1. A Comparison of Possible Next Generation JML Tools

		JML2	JML3	JML4	JML5	JaJML	EJ2	JACK
Base Compiler / IDE	**Name**	MJ	JDT	JDT	any Java 7+	JastAdd Java	EJ2	JDT
	Maintained (supports Java ≥ 5)	×	✓	✓	✓	✓	×[1]	✓
Reuse/extension of base (e.g. parser, AST) vs. copy-and-change		✓	×	✓	×	✓	×	×
Tool Support	**RAC**	✓	✓	✓	(✓)	✓	N/A	N/A
	ESC	N/A	(✓)	✓	N/A	×	✓	✓[2]
	FSPV	N/A	×	✓	N/A	×	N/A	✓

MJ = MultiJava, JDT = Eclipse Java Develoment Toolkit EJ2 = ESC/Java2
N/A = not possible, practical or not a goal, (✓) = planned

[1] ESC/Java2 is being maintained, but its compiler front end has yet to reach Java 5.

[2] Strictly speaking, JACK supports an automated form of FSPV, not ESC.

verification of annotated byte code [7]), being a proper Eclipse plug-in, it suffers from the same drawbacks as JML3 with respect to the need to maintain a separate compiler front-end. Additionally JACK does not provide support for RAC which we believe is an essential component of a mainstream IVE. An advantage that JACK has over JML4's current capabilities is that it can present VCs to the user in a Java/JML-like notation.

JML5. The JML5 project has taken a different approach [24]. Its goal is to embed JML specifications in Java 5 annotations rather than Java comments. Such a change will allow JML's tools to use any Java 5 compliant compiler. Unfortunately, the use of annotations has important drawbacks as well. In addition to requiring a separate parser to process the JML specific annotation contents (e.g. assertion expressions), Java's current annotation facility does not allow for annotations to be placed at all locations in the code at which JML can be placed. JSR-308 is addressing this problem as a consequence of its mandate, but any changes proposed would only be present in Java 7 and would not allow support for earlier versions of Java [17].

JaJML. JaJML is an early research prototype built atop JastAdd Java compiler [19]. JastAdd is a compiler framework that uses attribute grammars and supports Aspect Oriented Programming . JaJML's main advantage over JML4 is the ease with which it can be extended. Indeed, Haddad and Leavens note that adding RAC support to JaJML for while loop variants and invariants was done in a fraction of the number of lines of code that were needed to add them to JML4. The main disadvantages of JaJML include its lack of integration with an IDE and no guaranteed third-party maintenance for the underlying JastAdd Java compiler. While use of JastAdd offers increased modularity, there is likely to be a performance impact. The ability of JaJML to provide support for static verification has yet to be derisked.

Table 1 presents a summary of the comparison of the tools supporting JML. As compared to the approach taken in JML4, the main drawback of the other tools is that they are likely to require more effort to maintain over the long haul as Java continues to evolve and due to the looser coupling with their base.

6.2 Verification tool support for other languages

KeY. While the KeY tool was recently adapted to accept JML, it also supports other languages [1]. KeY is an integrated development environment which targets verification at a slightly higher level than most other tools. This is because KeY supports the annotation of design artifacts such as class diagrams. KeY does not support RAC or ESC though both automated and interactive FSPV are supported. Like JACK, KeY presents VCs in a JML-like notation.

Omnibus. Omnibus is a functional language with syntax similar to Java's that compiles to JVM bytecode [26]. The language was designed with reduced capabilities as compared to Java (e.g. it lacks support for exceptions, interface inheritance, and concurrency) so as to ease verification. Each of the source files in an Omnibus project has an associated verification policy that gives the level of verification required for it, which can be RAC, ESC, or FSPV. A custom IDE was developed that make these and other development activities easier, including tracking the verification status of (and method used for) each file. Simplify and PVS are the theorem provers used for ESC and FPV, respectively. Hence, Omnibus offers verification support comparable to that of JML4, though not for a mainstream language.

SPARK. SPARK [3] was developed for the implementation of safety-critical control systems. It is a subset of Ada extended with annotations to provide support for (an enriched form of) DBC. The subset was chosen to be amenable to ESC and FSPV and yet be useful for writing industrial applications. Unlike the other languages discussed in this section, there is no support for RAC, since static verification is used to show that errors—including contract violations—cannot happen at runtime. Static analysis is performed in three stages. The first stage is provided by the Examiner and is similar to the JDT's flow analysis. In the second stage, the Simplifier automatically discharges those VCs that it can and leaves the rest for the Proof Checker, an interactive theorem prover. The Proof Obligation Summary (POGS) tool is used to reduce the various outputs of the static analysis and proof tools to a single report that gives the status of the verification process overall including, in particular, a list of any VCs that remain unproved. In a sense, when using Offline User Assisted ESC, JML4 can be seen to behave like the POGS. All of these tools are stand-alone command-line tools.

7 Conclusion and Future Work

The idea of providing JML tool support by means of a closely integrated and loosely coupled extension to an existing compiler was successfully realized in JML2. Although this worked well, unfortunately the chosen base Java compiler is no longer officially maintained. Applying the same approach we have extended the Eclipse JDT to create the base infrastructure of JML4.

The first JML4 prototype served as a basis for discussion by some members of the JML consortium, and eventually it came to be adopted as the main avenue to pursue in the JML Reloaded effort [22]. A JML Winter School followed in February of this year, during which members of the community were given JML4 developer training [20, Wiki].

Since then, we have enhanced JML4's feature set, in particular, with support for next generation ESC and FSPV components.

Even though JML4's approach is currently more invasive than a proper plug-in design, using this approach we have since 2006, been able to (i) maintain JML4 despite the continuous development increments of the Eclipse JDT, and (ii) demonstrated, through the recent addition of the JML SV, that JML4's infrastructure is capable of supporting the full range of verification approaches from RAC to FSPV. Hence, we are hopeful, that JML4 will be a strong candidate to act as a next generation research platform and industrial grade verification environment for Java and JML.

Of course, there is much work yet to be done. The most pressing is completing the compiler front-end to support all or most of JML. Once this is done, it will be necessary to propagate support for JML constructs into the RAC, ESC and FSPV components. Completion of the front-end will also mark a milestone after which developers of other JML tools will be able to explore the possibility of integrating their tools within the JML4 framework.

Acknowledgments

We thank the anonymous referees for their comments and insightful questions. We also thank Gary Leavens for hosting the first JML Winter School which acted as the kick-off developer meeting for JML4, as well as Jooyong Lee and Daniel Zimmerman for their continued contributions to JML4. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Québec Fonds de Recherche sur la Nature et les Technologies.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt, "The KeY Tool", *SoSyM*, 4:32-54, 2005.

[2] D. Aspinall, "Proof General": http://proofgeneral.inf.ed.ac.uk, 2008.

[3] J. Barnes, *High Integrity Software: The Spark Approach to Safety and Security*. AW, 2003.

[4] M. Barnett and K. R. M. Leino, "Weakest-Precondition of Unstructured Programs". *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Lisbon, Portugal. ACM Press, 2005.

[5] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet, "JACK: a tool for validation of security and behaviour of Java applications". *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO)*, 2007.

[6] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications", *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212-232, 2005.

[7] L. Burdy, M. Huisman, and M. Pavlova, "Preliminary Design of BML: A Behavioral Interface Specification Language For Java Bytecode". *Proceedings of the Fundamental Approaches to Software Engineering (FASE)*, vol. 4422 of *LNCS*, pp. 215-229, 2007.

[8] L. Burdy, A. Requet, and J.-L. Lanet, "Java Applet Correctness: A Developer-Oriented Approach". *Proceedings of the International Symposium of Formal Methods Europe*, vol. 2805 of *LNCS*, pp. 422-439. Springer, 2003.

[9] P. Chalin and P. R. James, "Non-null References by Default in Java: Alleviating the Nullity Annotation Burden". *Proc. of the 21st European Conference on Object-Oriented Programming (ECOOP)*, Germany, vol. 4609 of *LNCS*, pp. 227-247. Springer, 2007.

[10] P. Chalin, P. R. James, and G. Karabotsos, "An Integrated Verification Environment for JML: Architecture and Early Results". *Proceedings of the Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, Cavtat, Croatia, Sept. 3-4, pp. 47-53. ACM, 2007.

[11] P. Chalin, P. R. James, F. Rioux, and G. Karabotsos, "Towards a Verified Software Repository Candidate: Cross-Verifying a Verifier", Concordia University, Dependable Software Research Group Technical Report, 2008.

[12] P. Chalin, J. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2", in *Fourth International Symposium on Formal Methods for Components and Objects (FMCO'05)*, vol. 4111, *LNCS*, pp. 342-363, 2006.

[13] Y. Cheon, "A Runtime Assertion Checker for the Java Modeling Language", Iowa State University, Ph.D. Thesis, also TR #03-09. April, 2003.

[14] D. R. Cok, "Design Notes (Eclipse.txt)", http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/trunk/docs/eclipse.txt, 2007.

[15] D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML". In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean editors, *Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, France, March 10-14, vol. 3362 of *LNCS*, pp. 108-128. Springer, 2004.

[16] X. Deng, Robby, and J. Hatcliff, "Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-oriented Systems", Kansas State University, 2007.

[17] M. Ernst and D. Coward, "Annotations on Java Types", JCP.org, JSR 308. October 17, 2006.

[18] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java". *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June, vol. 37(5), pp. 234-245. ACM Press, 2002.

[19] G. Haddad and G. T. Leavens, "Extensible Dynamic Analysis for JML: A Case Study with Loop Annotations", University of Central Florida CS-TR-08-05. April, 2008.

[20] G. T. Leavens, "The Java Modeling Language (JML)": http://www.jmlspecs.org, 2007.

[21] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin, "JML Reference Manual", http://www.jmlspecs.org, 2007.

[22] Robby, P. Chalin, D. R. Cok, and G. T. Leavens, "An Evaluation of The Eclipse Java Development Tools (JDT) as a Foundational Basis for JML Reloaded": jmlspecs.svn:/reloaded/planning, 2008.

[23] N. Schirmer, “A Sequential Imperative Programming Language Syntax, Semantics, Hoare Logics and Verification Environment”, in *Isabelle Archive of Formal Proofs*, 2008.

[24] K. B. Taylor, “A specification language design for the Java Modeling Language (JML) using Java 5 annotations”. Masters thesis, Iowa State University, 2008.

[25] J. van den Berg and B. Jacobs, “The LOOP compiler for Java and JML”. In T. Margaria and W. Yi editors, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, vol. 2031 of *LNCS*, pp. 299-312. Springer, 2001.

[26] T. Wilson, S. Maharaj, and R. G. Clark, “Omnibus: A Clean Language and Supporting Tool for Integrating Different Assertion-Based Verification Techniques”. *Proceedings of the Proceedings of REFT 2005*, Newcastle, UK, July, 2005.