

# An Integrated Verification Environment for JML: Architecture and Early Results

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., almost three years following its release, Java 5 has yet to be fully supported. This paper presents the architecture of JML4, an Integrated Verification Environment (IVE) for JML that builds upon Eclipse’s support for Java, enhancing it with Extended Static Checking (ESC), an early form of Runtime Assertion Checking (RAC) and JML’s non-null type system. Early results indicate that the synergy of complementary verification techniques (being made available within a single tool) can help developers be more effective; we demonstrate new bugs uncovered in JML annotated Java source—like ESC/Java2—which is routinely verified using first generation JML tools.

## Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—programming by contract; D.3.3 [Programming Languages]; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs.

## General Terms

Design, Languages, Theory, Verification.

## Keywords

Integrated Verification Environment, Java Modeling Language, Eclipse, JML4.

## 1. 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 [18]. 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 [3]. Of these, RAC and ESC are the technologies which are most likely to be adopted by mainstream developers because of their ease of use and low learning curve.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, September 3-4, 2007, Cavtat near Dubrovnik, Croatia.

Copyright 2007 ACM ISBN 978-1-59593-721-6/07/0009 ... \$5.00

In earlier work [6] we confirmed (among other things) how RAC and ESC are most effective when used *together*, particularly when it comes to the verification of sizeable systems. Unfortunately, this is more challenging than it should be; one of the key reasons being that the tools accept slightly different and incompatible variants of JML—sadly this is the case for practically all of the current JML tools. The top factors contributing to the current state of affairs are

- partly historical—the tools were developed independently, each having their own parsers, type checkers, etc. and
- partly due to the rapid pace of evolution of both JML and Java.

Not only does this last point make it difficult for individual research teams to keep apace, it also results in significant and unnecessary duplication of effort.

For some time now the JML community has recognized that a consolidation effort is necessary with respect to its tool base. In response to this need, three prototypical “next generation” tools have taken shape: JML3, JML4, and JML5 [18]. This paper presents the architecture and design rationale behind JML4: we explain why we believe JML4 will not suffer from the maintenance overhead of other JML tools even in the face of the rapid pace of evolution of Java.

The remainder of the paper is organized as follows. In the next section, we present early results demonstrating that the synergy of complementary verification techniques (being made available within JML4) can help developers be more effective; we illustrate new bugs uncovered in ESC/Java2 source—despite the fact that the code is routinely verified using itself and other JML tools. The remaining sections focus on JML tool support, offering

- a discussion of the goals to be achieved by any next generation JML tool base (Section 3) and
- a presentation (Section 4) of the architectural and (some aspects of) the detailed design of JML4; our objective is to provide sufficient detail to allow JML4’s design to be assessed relative to the stated goals.

Section 5 provides initial arguments supporting our belief that JML4’s design will be less costly to maintain in the long run than current JML tools. Section 6 offers a brief discussion and comparison of JML4 with its predecessor JML2 and siblings JML3 and JML5 as well as other tools like the Java Applet Correctness Kit (JACK). Conclusions and future work are presented in Section 7.

## 2. EARLY RESULTS: BENEFITS OF SYNERGY

One of JML4’s first and most fully developed features is JML’s non-null type system [7]. This, coupled with the tool’s ability to read the extensive JML API library specifications, renders it quite effective at statically detecting potential null pointer exceptions (NPEs). Recently, JML4 was enhanced to

```

package escjava;
...
public class Main extends javafe.SrcTool {
...
    public static Options options() {
        return (Options)options;
    }
...
    public String processRoutineDecl(...) {
        ...
        VcGenerator vcg = null; ...
        try {
            ... // possible assignment to vcg
        } // multiple catch blocks
        catch (Exception e) {
            ...
        }
        ...
        fw.write(vcg.old2Dot()); // <<< possible NPE
        ...
    }
}

```

**Figure 1. Code excerpt from the escjava.Main class**

support Extended Static Checking (ESC) through the integration of ESC/Java2 [11]. While each verification technique has strengths and weaknesses, integration of complementary techniques into a single verification environment brings about a level of synergy that would not be achievable otherwise.

As a concrete example of the kind of verification technique synergy which JML4 achieves, consider the code fragment given in Figure 1, an excerpt from ESC/Java2’s `escjava.Main` class. JML4 correctly reports that a dereference of `vcg` inside of `processRoutineDecl()` could result in an NPE (Figure 2).

Since ESC/Java2 is routinely run on itself, why was this error not detected before? Because analyzing `processRoutineDecl()`, which consists of 386 lines of code, is beyond the capabilities of ESC/Java2 (it gives up on attempting to verify the method because the verification condition is too big). Several errors that arise under such circumstances were identified in

ESC/Java2 source by JML4.

As another example, consider the static `options()` method of `escjava.Main` (Figure 1) which returns a reference to ESC/Java2’s command line options. This method is used throughout the code (272 occurrences) and its return value is directly dereferenced even though the method can return null.

While JML4 reports the 250+ NPEs related to the use of this method, ESC fails to do so because another ESC error prevents it from determining that the method can return null: namely, a possible type cast violation. The effect of having one error mask others is particularly acute for ESC/Java2 (even more so than in ordinary compilers) thus making effective the more resilient, though less powerful, complementary verification capabilities of other techniques such as those implemented in JML4 (and recently added to ESC/Java2 [17]). Our preliminary use of JML4 has demonstrated that, e.g., nullity type errors once fixed allow ESC to push further its analysis, helping expose yet more bugs in code and specifications, which leads to uncovering further nullity type errors, etc.

### 3. JML TOOLS: BACKGROUND AND GOALS

In this section we discuss the main goals to be satisfied by any next generation tool base for JML. Before doing so we give a brief summary of the JML’s first generation of tools.

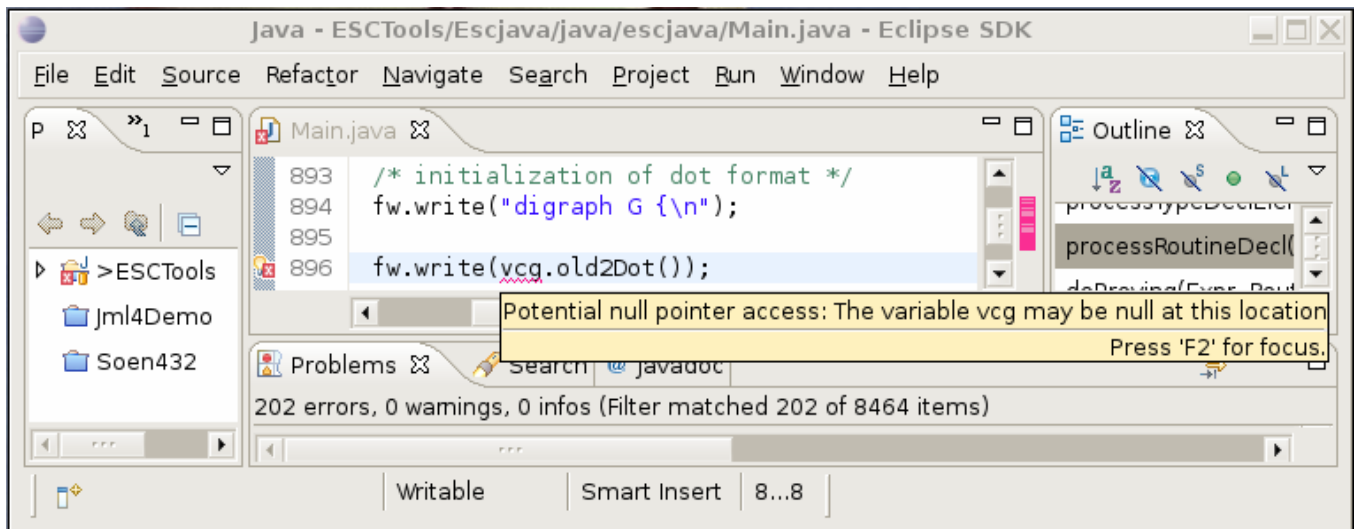
#### 3.1 FIRST GENERATION TOOLS

The first generation JML tools essentially consist of:

- Common JML tool suite—formerly the Iowa State University (ISU) JML tool suite—also known to developers as JML2, which includes the JML RAC compiler and `JmlUnit` [3],
- ESC/Java2, an extended static checker [11], and
- LOOP a full static program verifier [20].

Of these, JML2 is the original JML tool set. Although ESC/Java2 and LOOP initially used an annotation language other than JML, they quickly switched to use JML.

Being independent development efforts, each of the tools mentioned above has its own Java/JML front end including scanner, parser, abstract syntax tree (AST) hierarchy and static



**Figure 2. JML4 reporting non-null type system errors in a method too big for ESC to verify**

analysis code—though not all developed to the same level of completeness or reliability. This is a considerable amount of duplicate effort and code (of the order of 50-100K SLOC<sup>1</sup>). This became evident as JML evolved, but the main hurdle which has yet to be fully addressed is the advent of Java 5 (especially generics).

### 3.1.1 LESSONS LEARNED FROM JML2

Which lessons can be learned from the development of the first generation of tools, especially JML2 which, from the start, 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—e.g. abstract syntax tree nodes with their associated type checking 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 since 2002 by an increasing developer pool (there are currently 49 registered developers). In that case what, if anything, 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. Hence our approach in JML4 has been to repeat the successful approach adopted by JML2 but to ensure that we choose to extend a Java compiler that we are confident will be maintained (outside of the JML community).

### 3.1.2 EVOLUTION OF IDEs

Another important point to be made about the first generation of JML tools is that they are mainly command line tools, though some developers were able to make comfortable use of them inside Emacs, which in a sense, can be considered an early integrated development environment (IDE).

With a phenomenal increase in the popularity of modern IDEs like Eclipse, it seems clear that to increase the likelihood of getting widespread adoption of JML, it will be necessary to have its tools operate well within one or more popular IDEs. In recognition of this, early efforts have successfully provided basic JML tool support via Eclipse plug-ins, which mainly offer access to the command line capabilities of the JML RAC or ESC/Java2.

Other efforts (generation 1.5), resulted in tools that were built from the outset within an IDE but have *not* been designed to support RAC and ESC. These include the

- Java Applet Correctness Kit (JACK), built directly as an Eclipse plug-in, supports interactive static verification [2].
- KeY tool, which was recently adapted to support JML as a constraint language for expressing specifications in design models. The KeY tool is built on top of Borland’s Together IDE [1, 12].

## 3.2 GOALS FOR NEXT GENERATION TOOL BASES

We are targeting mainstream industrial software developers as our key end users. From an end user point of view, we strive to offer a single Integrated (Development and) Verification Environment (IVE) within which they can use any desired combination of RAC, ESC, and FSPV technology. No single tool currently offers this feature set for JML. In addition, user assistance by means of the auto-generation of specifications (or specification fragments) should be possible—e.g. based on approaches currently offered by tools like Daikon [14], Houdini [15] and JmlSpec [3].

Since JML is essentially a superset of Java, most JML tools will require, at a minimum, the capabilities of a Java compiler front end. Some tools (e.g., the RAC) would benefit from compiler back-end support as well. One of the important challenges faced by the JML community is keeping up with the rapid pace of the evolution of Java. As researchers in the field of applied formal methods, we get little or no reward for developing and/or maintaining basic support for Java. While such support is essential, it is also very labor intensive. Hence, an ideal solution would be to extend a Java compiler, already integrated within a modern IDE, whose maintenance is assured by a developer base outside of the JML research community. If the extension points can be judiciously chosen and kept to a minimum then the extra effort caused by developing on top of a rapidly moving base can be minimized.

In summary, our general goals are to provide

- a base framework for the integrated capabilities of RAC, ESC, and FSPV
- in the context of a modern Java IDE whose maintenance is outside the JML community
- by implementing support for JML as extensions to the base support for Java so as to minimize the integration effort required when new versions of the IDE are released.

A few recent projects have attempted to satisfy these goals. In the next section, we describe how we have attempted to satisfy them in our design of JML4; the other projects are discussed in the section on related work.

## 4. JML4

In our first feature set, JML4 enhanced Eclipse 3.3 with: scanning and parsing of nullity modifiers, enforcement of JML’s non-null type system (both statically and at runtime) and the ability to read and make use of the extensive JML API library specifications. This 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
- providing for runtime assertion checking (RAC).

Also, the chosen subset of features is useful in its own right, somewhat independent of other JML features [7]; i.e. the capabilities form a natural extension to the existing embryonic Eclipse support for nullity analysis.

We have since been pursuing our enrichment of the JML4 feature set so that to date, we have completed a full integration of

---

<sup>1</sup> (Physical) Source Lines of Code obtained by counting end-of-lines for non-comment code.

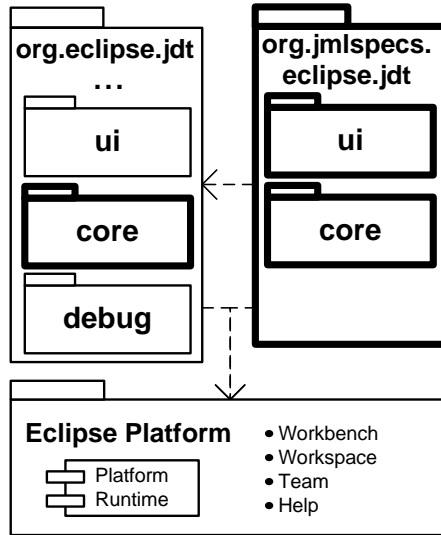


Figure 3. High-level package view

ESC/Java2 and begun work towards the support of runtime assertion checking of JML Level 0 [19, Section 2.9].

In the remainder of this section, we present our proposed means of extending Eclipse to support JML, appealing at times to the specific way in which the JML4 features described above have been realized.

#### 4.1 ARCHITECTURAL OVERVIEW

Eclipse is a plug-in based application platform. 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. Well known bundlings of Eclipse plug-ins include the Eclipse Software Development Kit (SDK) and the Eclipse Rich Client Platform (RCP). 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—called the Eclipse Java Development Tooling (JDT)—offering, among other things, a standard Java compiler and debugger.

The main packages of interest in the JDT are the `ui`, `core`, and `debug`. As can be gathered from the names, the `core` (non-UI) compiler functionality is defined in the `core` package; UI elements and debugger infrastructure are provided by the components in the `ui` and `debug` packages, respectively.

One of the rules of Eclipse development is that public APIs must be maintained *forever*. This API stability helps avoid breaking client code. The following convention was established by Eclipse developers: only classes or interfaces that are *not* in a package named `internal` can be considered part of the *public API*. Hence, for example, the classes for the JDT’s internal AST are found in the `org.eclipse.jdt.internal.compiler.ast` package, where as the public version of the AST is (partly) reproduced under `org.eclipse.jdt.core.dom`. For JML4 we have generally made changes to internal components (to insert hooks) and then moved most of the JML specific code to `org.jmlspecs.eclipse.jdt`.

At the top-most level, JML4 consists of:

- a customized version of the `org.eclipse.jdt.core` package (details will be given below) that is used as a drop-in replacement for the official Eclipse JDT `core`.

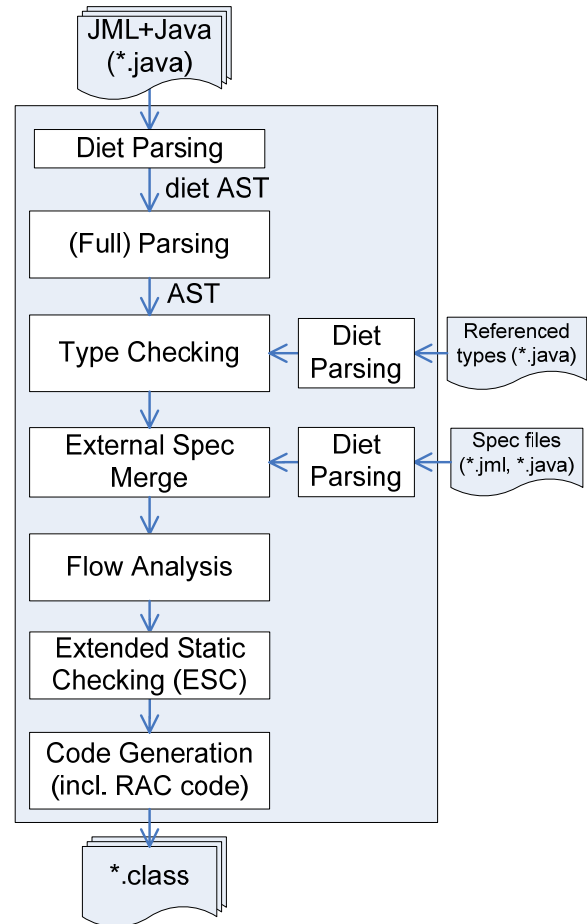


Figure 4. JDT/JML4 compilation phases

- JML specific classes contained in `org.jmlspecs.eclipse.jdt` including core classes (most of which are subclasses of the JDT Abstract Syntax Tree (AST) node hierarchy) and `ui` classes (e.g. for JML related preferences).

These packages are shown in bold in Figure 3.

#### 4.2 COMPILATION PHASES OVERVIEW

The main steps of the compilation process performed by JML4 are illustrated in Figure 4. In the Eclipse JDT (and JML4), there are two types of parsing: in addition to a standard full parse, there is also a diet parse, which only gathers signature information and ignores method bodies. When a set of JML annotated Java files is to be compiled, all are diet parsed to create (diet) ASTs containing initial type information, and the resulting type bindings are stored in the lookup environment (not shown). Then each compilation unit (CU) is fully parsed. During the processing of each CU, types that are referenced but not yet in the lookup environment must have type bindings created for them. This is done by first searching for a binary (`*.class`) file or, if not found, a source (`*.java`) file. Bindings are created directly from a binary file, but a source file must be diet parsed and added to the list to be processed. In both cases the bindings are added to the lookup environment. If JML specifications for any CU or referenced type are contained in a separate external file (e.g. a `*.jml` file), then these specification files are diet parsed and the resulting information merged with the CU AST (or associated with the

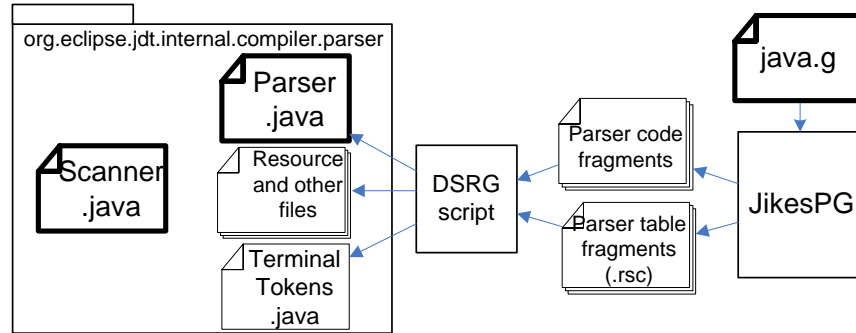


Figure 5. Customizing the JDT lexer and parser

binding in the case of a binary file). Finally, flow analysis and code generation are performed. Extended static checking is treated as a distinct phase between flow analysis and code generation. In the remaining subsections we briefly cover some aspects of JML4 compilation—details can be found in [8].

### 4.3 LEXICAL SCANNING, PARSING AND THE AST

Figure 5 provides an overview of the main parser components as well as the means by which they are generated; components in bold are those that have been customized under JML4.

**Scanning.** Since all of JML is contained within specially marked comments, the main change to the lexical scanner was to enhance it to recognize JML annotations. This is currently handled using a Boolean field that indicates if the scanner is in a JML annotation or not. Adding support for new keywords requires a little more work than usual since the JDT’s scanner is highly optimized and hand crafted. Keywords, for example, are identified by a set of nested case statements based on the first character of a lexeme and its length.

**Parsing.** The JDT’s parser is auto-generated from a grammar file (`java.g`) using the Jikes Parser Generator (JikesPG) and a custom script we have written. On a positive note, the grammar file, `java.g`, closely follows the Java Language Specification [16] and hence has been relatively easy to extend. Possibly the main source of difficulty in the parser is the lack of automatic support for token stacks.

Other than adding methods corresponding to new grammar-rule reductions, the most prominent change to the parser is the replacement of calls to constructors of JDT AST nodes with those of JML-specific AST subclasses. The abstract syntax tree hierarchy for JML4 is obtained by subclassing specific JDT AST nodes as needed. An illustration of how this is done is given in Figure 6. For example, JML type references are like Java type

references but have additional information such as nullity. Currently we subclass 20% of the AST node types; the JML specific subclasses generally contain very little code (and in particular, no code is copied from superclasses).

### 4.4 TYPE CHECKING AND FLOW ANALYSIS

Type checking is performed by invoking the `resolve()` method on a compilation unit. Similarly, flow analysis is performed by the `analyzeCode()` method. Addition of JML functionality is achieved by inserting “hooks” into the previously mentioned methods—i.e. calls to methods with empty bodies in the parent class that are then overridden in JML-specific AST nodes. Our hope is that such hooks will be ported back into the Eclipse JDT, something the JDT developers have confirmed is feasible provided we can demonstrate that no public APIs are changed and that there is little or no impact on runtime performance.

Between type checking and flow analysis, the compiler checks for external specification files (e.g., `*.jml` files) corresponding to the file being compiled. If one is found, it is parsed and any annotations are added to the corresponding declarations. Binary types (i.e., those found in `*.class` files) whose specifications are needed are handled differently. For these, the system searches for both a source and external specification file.

### 4.5 RUNTIME CHECKING AND EXTENDED STATIC CHECKING

Code generation is performed by each `ASTNode`’s `generateCode()` method. Its `CodeStream` parameter provides methods for emitting JVM bytecode and hides some of the bookkeeping details, such as determining the generated code’s runtime stack usage. Hence, supporting runtime checking is relatively straightforward.

Extended Static Checking in JML4 is currently achieved by a

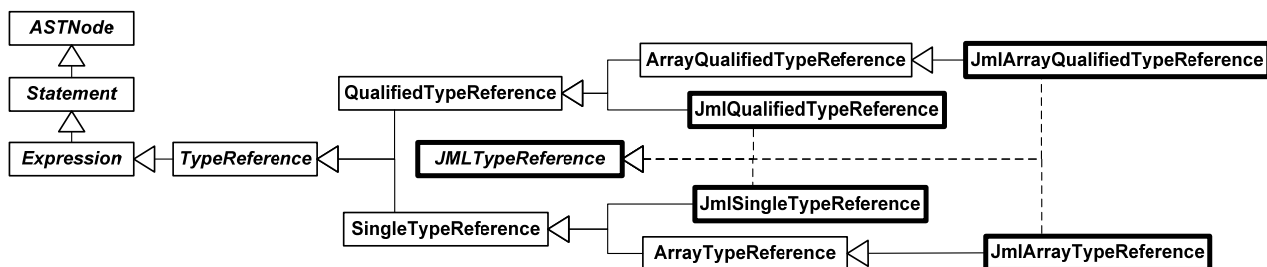


Figure 6. Part of the AST hierarchy (`org.eclipse.jdt.internal.compiler.ast`)

preliminary integration of ESC/Java2. That is, during compilation in a step just following flow analysis, we invoke `escjava`'s main processing method—which effectively reparses the file inside of ESC/Java2. While such an approach is inefficient, it has allowed us to focus on the integration of the problem reporting. As a next step, we will create a visitor which will map Eclipse JDT AST's into ESC/Java2's AST, thus avoiding the reparsing. Finally, we plan on building a custom transformation from the JDT's AST into ESC/Java2's guarded command language, hopefully allowing us to reuse the rest of ESC/Java2's verification condition generation back-end.

## 5. VALIDATION OF ARCHITECTURAL APPROACH

JML4 was recently used to help validate our proposal that JML's non-null type system should be non-null by default [7]. It was used to produce RAC-enabled versions of five case studies (totaling over 470K SLOC), which were then used to execute those systems' extensive test suites. This exercise gave us confidence in JML4's runtime checking capabilities and its ability to process JML API specifications.

JML4, like JML2, is built as a closely integrated and yet 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, losing committer rights means that we must maintain our own version of the JDT code. Use of the CVS vendor branch feature has made this manageable.

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 to our copy of the JDT code with special comment markers.

While following these conventions, incorporating each of the regular JDT updates since the fall of 2006 (to our surprise) has taken less than 10 minutes, on average.

## 6. RELATED WORK

In this section we briefly compare JML4 to its sibling next generation projects JML3, JML5 as well as to the Java Applet Correctness Kit (JACK). Further details, examples and tools are covered in [8].

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 [9]. 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” [9]. 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.

The Java Applet Correctness Kit (JACK) is a proprietary tool for JML annotated Java Card programs initially developed at Gemplus (2002) and then taken over by INRIA (2003) [2]. It uses a weakest precondition calculus to generate proof obligations that are discharged automatically or interactively using various theorem provers [5]. While JACK is emerging as a candidate next generation tool (offering features unique to JML tools such as verification of annotated byte code [4] and a proof obligation viewer), being a proper Eclipse plug-in, it suffers from the same drawbacks as JML3. Additionally JACK does not provide support for RAC which we believe is an essential component of a mainstream IVE.

The JML5 project, recently initiated at Iowa State University, has taken a different approach. 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 [13].

Table 1 presents a summary of the comparison of the tools. 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.

## 7. CONCLUSION AND FUTURE WORK

The idea of providing JML tool support by means of a closely integrated and yet loosely coupled extension to an existing compiler was successfully realized in JML2. This has worked well since 2002, but unfortunately the chosen Java compiler is not being kept up to date with respect to Java in a timely manner. We propose applying the same approach by extending the Eclipse JDT (partly through internal packages). Even though it is more invasive than a proper plug-in solution, using this approach we have demonstrated that it was relatively easy to enhance the type system and provide RAC and ESC support.

**Table 1. A Comparison of Possible Next Generation JML Tools**

		JML2	JML3	JML4	JML5	ESC/Java2 Plug-in	JACK
<b>Base Compiler / IDE</b>	<b>Name</b>	MJ	JDT	JDT	any Java 7+	ESC/Java2 and JDT	JDT
	<b>Maintained</b> (supports Java $\geq 5$ )	×	✓	✓	✓	×	✓
<b>Reuse/extension of base</b> (e.g. parser, AST) vs. copy-and-change		✓	×	✓	×	×	×
<b>Tool Support</b>	<b>RAC</b>	✓	✓	✓	(✓)	N/A	N/A
	<b>ESC</b>	N/A	(✓)	✓	N/A	✓	✓
	<b>FSPV</b>	N/A	(✓)	(✓)	N/A	N/A	✓

MJ = MultiJava,

JDT = Eclipse Java Development Toolkit

N/A = not possible, practical or not a goal,

(✓) = planned

<sup>1</sup> ESC/Java2 is currently being maintained to support new verification functionality, but its compiler front end has yet to reach Java 5 [10].

Other possible next generation JML tools have been discussed, but all seem to share the common overhead of maintaining a full Java parser, AST, and type checker separate from the base tools they are built from. This seems like an overhead that will be too costly in the long run. We are certainly not claiming that JML4 is the only viable next generation candidate but are hopeful that this paper has demonstrated that it is a likely candidate.

## 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”, *Software and System Modeling*, 4:32-54, 2005.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] 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. Springer, 2003.
- [6] P. Chalin and P. James, “Cross-Verification of JML Tools: An ESC/Java2 Case Study”. *Proceedings of the Workshop on Verified Software: Theories, Tools, and Experiments (VSTTE)*, Seattle, Washington, August, 2006.
- [7] P. Chalin and P. James, “Non-null References by Default in Java: Alleviating the Nullity Annotation Burden”. *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, Berlin, Germany, July-August, 2007.
- [8] P. Chalin, P. R. James, and G. Karabotsos, “The Architecture of JML4, a Proposed Integrated Verification Environment for JML”, Dependable Software Research Group, Concordia University, ENCS-CSE-TR 2007-006. May, 2007.
- [9] D. R. Cok, “Design Notes (Eclipse.txt)”, <http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/trunk/docs/eclipse.txt>, 2007.
- [10] D. R. Cok, E. Hubbers, and E. Rodríguez, “Esc/Java2 Eclipse Plug-in”, <http://sort.ucd.ie/projects/escjava-eclipse/>, 2007.
- [11] 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)*, Marseille, France, March 10-14, vol. 3362 of LNCS, pp. 108-128. Springer, 2004.
- [12] C. Engel and A. Roth, “KeY Quicktour for JML”: [www.key-project.org](http://www.key-project.org), 2006.
- [13] M. Ernst and D. Coward, “Annotations on Java Types”, JCP.org, JSR 308. October 17, 2006.
- [14] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The Daikon system for dynamic detection of likely invariants”, *Science of Computer Programming*, 2007.
- [15] C. Flanagan and K. R. M. Leino, “Houdini, an Annotation Assistant for ESC/Java”. *Proceedings of the International Symposium of Formal Methods Europe*, Berlin, Germany, vol. 2021, pp. 500-517. Springer, 2001.
- [16] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison-Wesley Professional, 2005.
- [17] M. Janota, R. Grigore, and M. Moskal, “Reachability Analysis for Annotated Code”, UCD Dublin, submitted to SAVCBS, 2007.
- [18] G. T. Leavens, “The Java Modeling Language (JML)”: <http://www.jmlspecs.org>, 2007.
- [19] 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.
- [20] 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.