The Architecture of JML4, a Proposed Integrated Verification Environment for JML

Patrice Chalin, Perry R. James, George Karabotsos
www.dsrg.org

Technical Report 2007-006, revision 1
May 2007

Dependable Software Research Group
Department of Computer Science and Software Engineering
Faculty of Engineering and Computer Science
Concordia University

Keywords: integrated verification environment, Java Modeling Language, JML4
2001 CR Categories: D.2.1 [Software Engineering] Requirements/ Specifications — languages, tools, JML; D.2.2 [Software Engineering] Design Tools and Techniques; D.2.4 [Software Engineering] Software/Program Verification; D.3.2 [Programming Languages] Language Classifications — Object-oriented languages; F.3.1 [Logics and Meanings of Programs] Specifying and Verifying and Reasoning about Programs.

Revision History

- 2007, May – first public release.

Table of Contents

The Architecture of JML4, a Proposed Integrated Verification Environment 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. Java Modeling Language tools cover the full range of verification from runtime assertion checking (RAC) to full static program verification, with extended static checking (ESC) in between. Experience demonstrates that verification of sizeable programs is best achieved when these technologies are used together. Unfortunately, developers trying to do this must use separate applications and deal with problems like the tools accepting slightly different and incompatible variants of JML. Tool consolidation has become vital. Thankfully, "next generation" tools are taking shape. This paper presents the architecture and design rationale behind JML4, our proposal for an Eclipse-based Integrated Verification Environment; as a proof-of-concept, JML4 currently enhances Eclipse with JML's non-null type system and the ability to read JML API library specifications. In addition to the basic features expected from an IDE, JML4 provides RAC and hooks for an ESC subsystem. A discussion of other consolidation efforts and emerging tools is also given.

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 [16]. 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 [2]. 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.

In earlier work [5] 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 with respect to its tool base is necessary. In response to this need, three prototypical "next generation" tools have taken shape: JML3, JML4, and JML5. This paper is an introduction to the architecture and design rationale behind JML4. After discussing the goals to be achieved by any next generation JML tool base (Section 2), we move directly to the treatment of JML4 (Section 3). Section 4 briefly presents early results in the use of JML4, while we leave until Section 5 a discussion and comparison of JML4 with its siblings JML3 and JML5 as well as other tools like the ESC/Java2 plug-in and the Java Applet Correctness Kit (JACK). Conclusions and future work are presented in Section 6.

2 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.

2.1 First Generation Tools

The first generation JML tools essentially consist of:

- Common JML tool suite[1], also known to developers as JML2, which includes the JML RAC compiler and JmlUnit [2],
- ESC/Java2, an extended static checker [10], and
- LOOP a full static program verifier [17].

Of these, JML2 is the original JML tool set. ESC/Java2 and LOOP initially used their own annotation language, though 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 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[2]). This became evident as JML evolved, but the main hurdle which has yet to be fully addressed is the advent of Java 1.5 (especially generics).

2.1.1 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

[1] Formerly known as the Iowa State University (ISU) JML tool suite.
[2] (Physical) Source Lines of Code obtained by counting end-of-lines for non-comment code.

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 notes 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).

2.1.2 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[3] "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 wide spread 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.

2.2 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 capability 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 [12], Houdini [14] and JmlSpec [2].

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

[3] Some might qualify Emacs as Neolithic even though, once properly configured, Emacs has many of the features of modern IDEs. But, for some reason it remains less popular.

from compiler back-end support as well. One of the important challenges faced by the JML community is keeping up with the accelerated 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.

3 JML4

JML4 currently enhances 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 functionality added by the chosen subset of features is useful in its own right, somewhat independent of other JML features; i.e. the capabilities form a natural extension to the existing embryonic Eclipse support for nullity analysis.

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.

3.1 Architectural Overview

3.1.1 Eclipse

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.

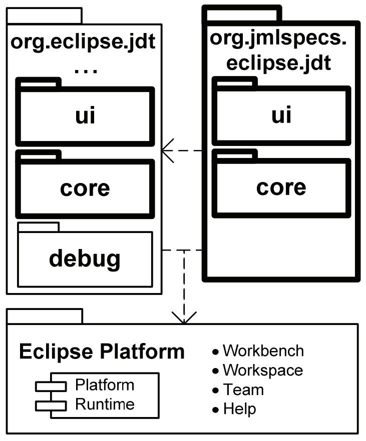


Figure 1. High-level package view

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`.

3.1.2 JML4

At the top-most level, JML4 consists of customized versions of the `org.eclipse.jdt.ui` and `org.eclipse.jdt.core` packages (details will be given below) that are used as drop-in replacements for the official Eclipse JDT `core` and `ui`. These packages are shown in bold in Figure 1.

The complete list of packages that have been customized is given in Figure 2. From this list we can note, among other things, that we have also customized the batch compiler so that JML4 can be used from the command line as well as within the Eclipse GUI. This will allow tools based on JML4

```
org/eclipse/jdt
   /core
     /compiler
   /internal/compiler
     /ast
     /batch
     /codegen
     /flow
     /lookup
     /parser
   /internal/core
     /builder
     /search/indexing
     /util
```

Figure 2. Packages customized to support JML4

to be used both interactively and in batch processing scripts. Most of the JML-specific modules in the replacement for the `org.jmlspecs.eclipse.jdt` plug-in are subclasses of the Abstract Syntax Tree (AST) node hierarchy, which we will examine in greater detail in Section 3.3.3 and a few utilities that help with external specifications.

3.2 Compilation Phases Overview

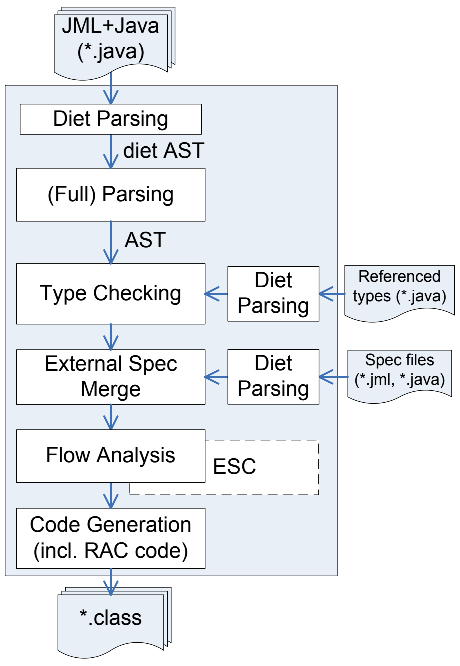


Figure 3. JDT/JML4 compilation phases

The main steps of the compilation process performed by JML4 are illustrated in Figure 3. 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 to fill in its methods' bodies. 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 binding in the case of a binary file). Finally, flow analysis and code generation are performed. We anticipate that extended static checking will be treated as a distinct phase immediately following flow analysis.

In the remaining subsections we cover some aspects of JML4 compilation in more detail.

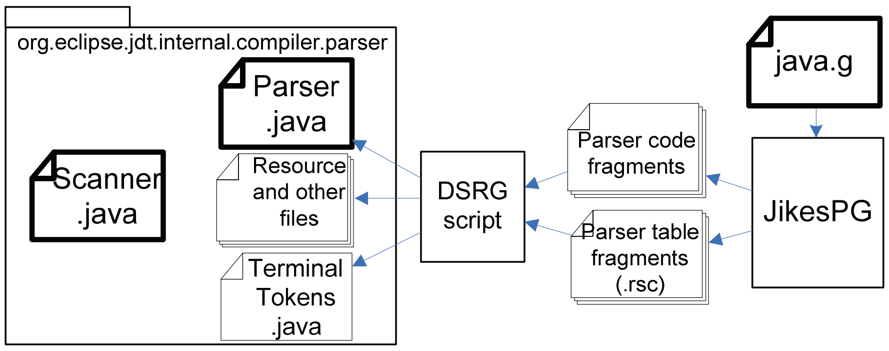


Figure 4. Customizing the JDT lexer and parser

3.3 Lexical Scanning, Parsing and the AST

In this section we describe some of the particularities of the scanner and parser as well as the approach we have taken to adapting them to support JML. Figure 4 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.

3.3.1 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. Figure 5 illustrates part of the code added for the recognition of `non_null` and `nullable` tokens.

```
…
switch (data[index]) {
  case 'n' : //non_null nullable …
    switch(length) {
      case 8:
        if (data[++index] == 'o')
           if ((data[++index] == 'n')
            && (data[++index] == '_')
            && (data[++index] == 'n')
            && (data[++index] == 'u')
            && (data[++index] == 'l')
            && (data[++index] == 'l')) {
                return TokenNamenon_null;
           } else
                return TokenNameIdentifier;
        else if ((data[index] == 'u')
             && (data[++index] == 'l')
            && (data[++index] == 'l')
            && (data[++index] == 'a')
            && (data[++index] == 'b')
            && (data[++index] == 'l')
            && (data[++index] == 'e')) {
                return TokenNamenullable;
        } …
```

Figure 5. Lexer code for nullity keywords

3.3.2 Parsing

The JDT's parser is auto-generated from a grammar file (`java.g`) using the Jikes Parser Generator (JikesPG)—see Figure 4. Instead of producing a complete parser as a single unit, the JikesPG creates resource files and code fragments that must be incorporated into various other files. We have automated the parser-generation process with a script; the grammar and script reside in a top-level `grammar`

```
Nullity -> 'nullable'
Nullity -> 'non_null'
…
ReferenceType ::= Nullity ClassOrInterfaceType
/.$putCase consumeReferenceType();  $break ./
```

Figure 6. JikesPG grammar productions

directory within the JDT core project, while the `Scanner` and `Parser` are in the compiler's internal `parser` directory. As mentioned earlier, there are two forms of parsing: diet parsing, which only gathers signature information, and full parsing. Both are driven by the same grammar file and use the same `Parser` class.

Grammar. On a positive note, the grammar file, `java.g`, closely follows the Java Language Specification [15]. Somewhat more of a challenge is dealing with the lack of JikesPG documentation. Even though the JikesPG is hosted on SourceForge, the site contains no documentation. To our knowledge the only up-to-date documentation is part of a (master's) thesis written in German [18]. (LPG, apparently a successor to the JikesPG project, is also hosted on SourceForge and has documentation, but it is unclear how different these two applications are. LPG appears to be part of a larger IBM initiative named SAFARI which is described to be "A Meta-Tooling Platform for Creating Language-Specific IDEs". This sounds promising, but SAFARI has yet to be released.)

The `java.g` grammar file has sections defining Terminals, Aliases, and Rules. Aliases are used to give readable names to terminals formed from punctuation. An example of production rules for adding support for nullity modifiers to reference types is given in Figure 6. Three rules are shown, and both "`->`" and "`::=`" can be used to separate a non-terminal from its definition. Semantic actions come after the production rule between '`/.`' and '`./`' markers. `$putCase` and `$break` are macros that output case and break statements for the production. Most—if not all—of the semantic actions in the JDT grammar are single calls to methods that are defined in the `Parser` class. This JDT convention helps to both reduce the size of the grammar file and increase its readability.

Parser Token Stacks. Unlike other auto-generated parsers, those generated by JikesPG do not provide a token stack, and this must be handled in the hand-written code. The `consumeToken()` method is called as each token is processed, and it is used to store information about the current token onto various stacks. Witte provides a summary of the eight stacks maintained by the JDT's parser [18, §3.4.3.3].

Other than modifying the methods corresponding to grammar-rule reductions, the most prominent change to the parser is the replacement of calls to constructors JDT AST nodes by those of JML-specific AST subclasses.

3.3.3 Abstract Syntax Tree Hierarchy

The Abstract Syntax Tree (AST) hierarchy for JML4 is obtained by subclassing specific JDT AST nodes as needed. An illustration of how this is done is given in Figure 7. For example, JML type references are like Java type references but have additional information such as nullity. (As can be seen, since Java does not allow multiple inheritance, adaptations of the AST can be a bit more involved than merely subclassing.)

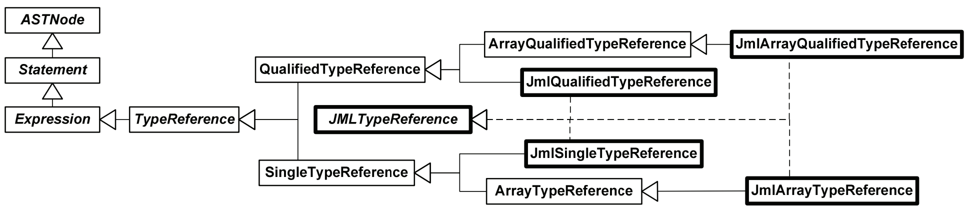


Figure 7. Part of the AST hierarchy (`org.eclipse.jdt.internal.compiler.ast`)

3.4 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 `analyseCode()` method. Addition of JML functionality is achieved by inserting "hooks" into the previously mentioned methods—i.e. 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.

3.4.1 Merging of External Specifications

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. Also, since binary types do not have declarations, but only bindings, information cannot be easily attached to them. For now, we store the specification information about fields and methods of binary types in a cache that is managed by the `JmlBinaryLookup` class.

3.4.2 Non-Null Type System

JML4's non-null type system is similar to that presented in [13]. That is, each Java reference type *T* is replaced with a pair of types, `nullable` *T* and `non_null` *T*, with `non_null` *T* a subtype of `nullable` *T*. We have initially left out support for generic types and concern for the initialization soundness issues addressed with raw types. Work is underway to address these, as well as limitations on the handling of arrays (e.g., array elements are currently restricted to having the same nullity as the array reference, and assignment of arrays is invariant with respect to nullity).

It was much easier to add a non-null type system to the JDT than it was to the JML2 Checker. This is partly due to the intra-procedural nullity analysis already implemented in the JDT, but even more so because of the ease of adding nullity information to the AST. The JDT's nullity analysis was only concerned with local variables and method parameters, as the latter are implemented in the AST as a subclass of local variables. In the JML2 Checker, it was necessary to modify every AST node that contained a reference to a type to also store its nullity. In the JDT, each mention of a type in the source code is reflected in the AST with a node that is a subclass of `TypeReference`. These were replaced with instances of subtypes of `JmlTypeReference`, which simply add a `Nullity` field. The nullity of a method's return type is stored in a `MethodDeclaration`, and that of all other typed items is stored in subclasses of `AbstractVariableDeclaration`. Since much of the type-checking and flow-analysis code makes use of *bindings* and not *declarations*, modifications were made so that each `FieldBinding` and `MethodBinding` created for a source file stores a reference to its declaration.

Once nullity information was easily accessible, the intra-procedural nullity checking only needed to be modified slightly to add the checking of fields and return types. All dereferences were already guarded during flow analysis by a call to `Expression`'s `checkNPE` method. This method originally ignored cases in which the reference being checked was not to a local variable (or method parameter), so it was modified to also complain when the reference was neither to a local (or parameter) nor declared to be non-null. This required the addition of an `isDeclaredNonNull` method to `Expression` that was overridden in only a small number of its subclasses. We also had to ensure that nullity declarations and information were in fact used. For example, since originally their nullity was taken to be unknown, the declared nullity of a method's parameters must be explicitly initialized before doing flow analysis on it. Also, local variables were originally set to definitely null, non-null, or unknown, and this was changed so they are either definitely non-null or potentially null.

Assignments were the only case for which there was no built-in support. There are two forms of assignment: explicit assignment to a variable and parameter binding during a method call. Assignment of a reference is only allowed if the `lhs` is declared to be nullable or the `rhs` is known to be non-null. Checking method parameters only required a straightforward change to the behavior of the `messageSend` class: before doing the original flow analysis, a check is made to determine if the `Expression`s representing the actual parameters can be assigned to the formal `Argument`s. When an assignment would not be allowed, an error is reported.

3.4.3 Problem reporting

Eclipse has a sophisticated error reporting subsystem that supports flexible filtering controlled by compiler preferences that are based on individual compiler options and option groups. Nonetheless, adding support for a new problem (i.e., something that can translate into an error or warning to the user) requires only small additions to the compiler's problem reporter and the addition of some minor glue in the form of `int` and `String` constants.

```
public static void generateNullityTest(CodeStream codeStream,
                                        String exceptionType, String msg) {
  BranchLabel nonnullLabel = new BranchLabel(codeStream);
  codeStream.dup();
  codeStream.ifnonnull(nonnullLabel);
  codeStream.newClassFromName(exceptionType.toCharArray(), msg);
  codeStream.athrow();
  nonnullLabel.place();
}
```

Figure 8. Code generation example for runtime checking of a cast (to non-null)

Any violations of the non-null type system are reported using the JDT's problem-reporting mechanisms. This requires adding a new method to the `ProblemReporter` class for each kind of problem. These pass the necessary information to an inherited `handle` method, which presents the problems in the GUI by, among other things, underlining the offending code in yellow or red and adding an entry in the `Problem` view. When using the JDT's batch compiler, handling a problem causes information to be sent to the standard output.

3.5 Runtime 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.

As an example, consider Figure 8, which shows the code from `JmlCastExpression` for checking a cast to non-null. This method tests the top element on the stack against null and throws an exception if it is. First, a label that will be used as a jump target is created. Then, the value on top of the runtime stack is duplicated so that a copy will remain after the execution of the next instruction, which removes the top element and jumps to the given target if the value popped is not null. A new exception of the given type is constructed, with the given message as its parameter, and placed on the runtime stack. The next statement `throws` the item on the top of the stack. Finally, the label for the conditional jump's target is placed after the exception-throwing code.

Only very early analysis has been conducted with respect to integration of ESC. We anticipate that hooks would be provided to invoke ESC between the compiler's calls to `analyseCode()` and `generateCode()`. In addition to providing warnings to the user, this would allow the ESC subsystem to manipulate the AST before bytecode is generated, possibly removing checks for properties that the ESC has proven. Since ESC is neither sound nor complete, removing checks should always be an option and not forced.

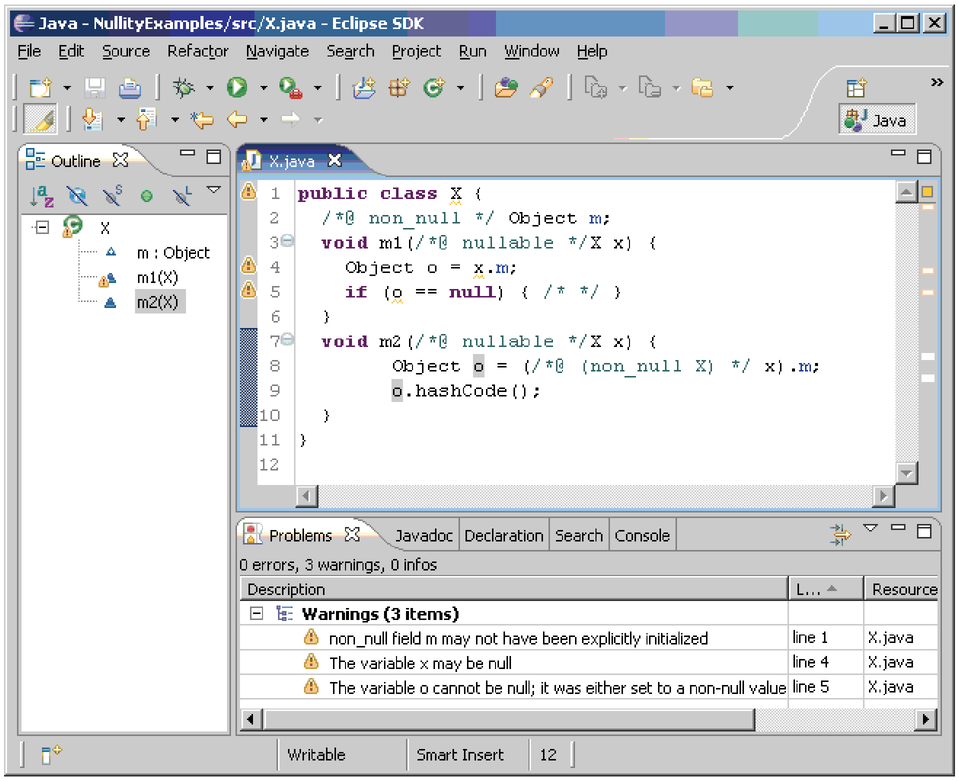
3.6 Testing Framework

```
public void test_0004_AssignmentExpression() {
  this.runNegativeTest(
    new String[] {
      "AssignmentExpression.java",
      "/*@ nullable_by_default */\n" +
      "\n" +
      "\n" +
      "\n" +
      "class AssignmentExpression {\n" +
      " /*@ non_null */ String non  = \"hello\"; //$NON-NLS-1$\n" +
      " /*@ nullable */ String able = null;\n" +
      "\n" +
      " void m1(/*@non_null*/ String s) { this.non = s; }\n" +
      " void m2(/*@nullable*/ String s) { this.non = s; } //error\n" +
      " void m3(/*@non_null*/ String s) { this.able = s; }\n" +
      " void m4(/*@nullable*/ String s) { this.able = s; }\n" +
      " void m7(/*@non_null*/ String s) { if (s!=null) this.non = s; }\n" +
      " void m8(/*@nullable*/ String s) { if (s!=null) this.non = s; }\n" +
      " void m9(/*@non_null*/ String s) { if (s!=null) this.able = s; }\n" +
      " void m10(/*@nullable*/ String s) { if (s!=null) this.able = s; }\n" +
      "}\n"
    },
    "----------\n" +
    "1. ERROR in AssignmentExpression.java (at line 10)\n" +
    " void m2(/*@nullable*/ String s) { this.non = s; } //error\n" +
    "                                   ^^^^^^^^^^^^\n" +
    "Possible assignment of null to an L-value declared non_null\n" +
    "----------\n" +
    "2. ERROR in AssignmentExpression.java (at line 13)\n" +
    " void m7(/*@non_null*/ String s) { if (s!=null) this.non = s; }\n" +
    "                                       ^\n" +
    "The variable s cannot be null; it was either set to a non-null value or
assumed to be non-null when last used\n" +
    "----------\n" +
    "3. ERROR in AssignmentExpression.java (at line 15)\n" +
    " void m9(/*@non_null*/ String s) { if (s!=null) this.able = s; }\n" +
    "                                       ^\n" +
    "The variable s cannot be null; it was either set to a non-null value or
assumed to be non-null when last used\n" +
    "----------\n"
  );
}
```

Figure 9. JML-JDT unit test

Unit tests of both the compile time and runtime checking have been developed. The compile-time tests have been integrated into the JDT's unit-testing framework, and an example can be seen in Figure 9. To create a new set of tests, a subclass of `AbstractRegressionTest` is created. Compiler options can be set by overriding the `getCompilerOptions()` method. The names of the individual test methods begin with the word `test`, which is usually followed by an (ordinal) test number and a descriptive name. The body of the test is often a single method call, to either `runPositiveTest` or `runNegativeTest`, depending on whether output is expected. The source of the code to be compiled is inlined as a `String`, as is any expected output.

The runtime tests remain for now independent JUnit tests, but they are to be integrated into the JDT's framework soon.


Java - NullityExamples/src/X.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Outline
X
m : Object
m1(X)
m2(X)
X.java
Problems
Javadoc
Declaration
Search
Console
Writable
Smart Insert
12

```
 1 public class X {
 2   /*@ non_null */ Object m;
 3   void m1(/*@ nullable */X x) {
 4     Object o = x.m;
 5     if (o == null) { /* */ }
 6   }
 7   void m2(/*@ nullable */X x) {
 8         Object o = (/*@ (non_null X) */ x).m;
 9         o.hashCode();
10   }
11 }
12
```

```
0 errors, 3 warnings, 0 infos
Description                                                        L...   Resource
Warnings (3 items)
  non_null field m may not have been explicitly initialized       line 1  X.java
  The variable x may be null                                      line 4  X.java
  The variable o cannot be null; it was either set to a non-null value line 5  X.java
```

Figure 10. Screenshot of JML4

4 Early Results and Validation of Architectural Approach

4.1 Use of JML4

JML4 was recently used to help validate our proposal that JML's non-null type system should be non-null by default [6]. 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 the runtime checking and the processing the JML API specifications. A screenshot of the edit-time and compile-time checking of nullity annotations is illustrated in Figure 10.

4.2 Validation of Architectural Approach

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 accelerated pace of evolution of Java. Unfortunately, loosing 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 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 the regular JDT updates since the fall of 2006 (to our surprise) has taken less than 10 minutes, on average.

5 Related Work

In this section we provide a brief discussion of the tools which have either positioned themselves as next generation JML tools or at least could be considered potential candidates.

5.1 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 [8]. In addition, JML3 goals included: providing functionality similar to that made available by command line tools (e.g. checker, RAC, ESC), "classic Eclipse UI enhancements" for JML (e.g. syntax highlighting), as well as support for generation of specifications. Considerable work has been done to develop the necessary infrastructure, but there are growing concerns about the long term costs of this approach.

Due to the closed non-extensible nature of the public JDT extensions points, Cok had to write a separate parser for the entire Java language and AST. As was mentioned earlier, the JDT creates two AST structures, one internal (using nodes from `org.eclipse.jdt.-internal.compiler.ast`) and the other part of the public API (`org.eclipse.jdt.core-.dom`). The public AST is generated from the internal version, but this conversion is one way. JML annotations are parsed with a custom parser. This JML parser is applied only to the comments found in the source code. The resulting JML AST nodes are used to

```
class Tester {
  private @spec_public int a;
  private @spec_public int b;
  @InvariantDefinitions({
    @Invariant( value = “a > 0”, msg = ”a is positive”),
    @invariant( value = “b > 0”, msg = “b is positive”)
  })

  @SpecCase(type = Type.normal_behavior,
            requires = "n.length == 2",
            ensures = "a == @old(a)+n[0] && b == @old(b)+n[1]")
  public @Pure bool m(int @NonNull [] n) {a+=n[0]; b+=n[1];}
}
```

Figure 11. JML5 Example Specification

decorate the original JDT DOM AST, and a second step is needed to match the JML AST nodes to the correct JDT AST nodes.

Cok notes that “JML3 [will need] to have its own name/type/resolver/checker for both JML constructs [and] all of Java...” in addition to the duplicated parser and AST [8]. Since one of the main reasons for integrating JML with Eclipse was to escape from providing support for the base Java language, this is a key disadvantage.

5.2 JML5

An annotation apparatus was introduced in Java 5 for decorating classes, fields, and methods with meta-data. JML5 is a project, recently initiated at Iowa State University, with the goal to replace JML specifications in Java comments with annotations. Such a change will allow JML’s tools to use any Java 5 compliant compiler.

An example of a JML5 specification is shown in Figure 11. It illustrates the use of a JML declaration modifier (`@spec_public`) on the two fields `a` and `b` to make them accessible to specifications in other classes. The two fields are constrained to being positive through the definition of two invariants. These are enclosed within an `@InvariantDefinitions` annotation because a declaration cannot currently have multiple annotations of the same type. Method specifications are enclosed within `@SpecCase(…)` annotations. Here, the method `m` has a normal-behavior heavyweight specification case, as denoted by the `Type.normal_behavior` attribute. `Type.exceptional_behavior` and `Type.behavior`, both with their usual meaning, are also defined in JML5. The absence of a `type` attribute indicates a lightweight specification. Multiple specification cases can be defined with the `@Also` annotation.

Unfortunately, the use of annotations has important drawbacks as well. 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 (Annotations on Java Types) is addressing this problem as a consequence of its mandate [11], but any changes proposed would only be present in Java 7 and would not allow support for earlier versions of Java [11]. Additionally, provisions would have to be made to allow for the conversion of the extensive JML libraries to be accessible to the new tools.

5.3 Java Applet Correctness Kit (JACK)

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) [1]. It uses a weakest precondition calculus to generate proof obligations that are discharged automatically or interactively using various theorem provers [4].

JACK's main goals are that (1) it should be supported in an environment familiar to developers and (2) it should be easy for Java developers to verify their own code [4]. The first goal is accomplished by providing JACK as an Eclipse plug-in and the second by providing developers with a proof obligation viewer. This viewer is used to communicate the proof obligations along with their associated JML and Java code to the user. To further facilitate ease of use, these proof obligations are displayed in the Java/JML Proof Obligation Language (JPOL). JPOL shares its syntax with Java and JML thus hides theorem prover specific syntax.

The proof obligations are discharged using one of the supported automated and interactive provers, currently the B prover, Coq, PVS, and Simplify. Through the Jack proof viewer a user can see the proof obligation in either JPOL or the prover's native representation. Through this viewer, user interaction is limited to identifying false hypotheses or showing invalid execution paths in the code. If the user is has the required expertise, then the proof obligations native to a specific theorem prover are displayed and the user can interactively attempt to discharge them.

While JACK is emerging as a candidate next generation tool (offering features unique to JML tools such as byte code verification [3]), being a proper Eclipse plug-in, it suffers from the same drawbacks as JML3.

Table 1. A Comparison of Possible Next Generation JML Tools

		JML2	JML3	JML4	JML5	ESC/Java2 Plug-in	JACK
Base Compiler / IDE	**Name**	MJ	JDT	JDT	any Java 7+	ESC/Java2 and JDT	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	✓	✓
	FSPV	N/A	(✓)	(✓)	N/A	N/A	✓

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

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

5.4 ESC/Java2 plug-in

An Eclipse plug-in was developed for ESC/Java2 with the latest release dating to February, 2005 [9]. It provides functionality similar to that of the command line tool. Additionally, code and specification elements responsible for verification violations are highlighted and associated with useful error messages in a fashion similar to other Java warnings.

To construct this plug-in, the code base of ESC/Java2 is packaged into a .jar file. Provided a Java file is being edited, options are available to the user to statically verify the code. Upon the user's invocation, the environment is prepared and a top-level method is called that causes the Java source file to be parsed and a verification condition (VC) to be generated and fed to the prover. Violations are then reported in Eclipse.

Simply put, this is a wrapper for the command line tool. Parsing is done using the ESC/Java2 parser. Nevertheless, this is an improvement to the command line tool simply because it integrates ESC/Java2 with Eclipse and makes it even easier to verify code.

5.5 Summary

Table 1 presents a summary of the comparison of the tools that have been suggested as possible foundations for the next generation of support for 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 due to the looser coupling with their base.

6 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 support. Including support for ESC is currently under consideration, as are other core JML features.

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.

While JML4 is currently only on our local servers, it will be published to the JmlSpecs [16] Source Forge CVS by the end of May 2007.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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*, Berlin, Germany, July-August, 2007.

[7] 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.

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

[9] D. R. Cok, E. Hubbers, and E. Rodríguez, “Esc/Java2 Eclipse Plug-in”, http://sort.ucd.ie/projects/escjava-eclipse/, 2007.

[10] 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.

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

[12] 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.

[13] M. Fähndrich and K. R. M. Leino, “Declaring and Checking Non-null Types in an Object-Oriented Language”, in *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA'03*: ACM Press, pp. 302-312, 2003.

[14] 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.

[15] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison-Wesley Professional, 2005.

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

[17] 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.

[18] M. Witte, “Portierung, Erweiterung und Integration des ObjectTeams/Java Compilers für die Entwicklungsumgebung Eclipse”, Technische Universität Berlin, 2003.