A JML Compiler on Eclipse Platform

Amritam Sarcar
Dept. Computer Science, University of Texas, El Paso
500 West University Avenue
El Paso, TX - 79902
1-325-513-5214
amritamsarcar@yahoo.co.in

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. 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. This paper presents the architecture and design rationale behind a JML Compiler on Eclipse Platform, an extension of [1], with a more detailed and focused study on runtime assertion checking

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*programming by contract, assertion checkers, class invariants*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, invariants, assertions*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object oriented programming*; D.3.2 [**Programming Languages**]: Language Classifications—*JML.*

General Terms

Languages

Keyword

JML, run-time checking, design by contract.

1. INTRODUCTION

The Java Modeling Language (JML) is the most popular Behavioral Interface Specification Language (BISL) for Java. It is a formal specification language that can document detailed designs of Java classes and interfaces. 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.

JML does not support the many new features of Java version 5, most notably generics. This inability to support current Java programs is limiting JML's user base, decreasing user feedback, and lessening the impact of JML-based research. The Verified Software grand challenge identified the lack of extensible tools for formal methods research as a major impediment to experiments.

Unfortunately, the current version, accepts different and incompatible variants of JML—sadly this is the case for practically all of the current JML tools. The reasons behind it 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 a possible extension of JML4.

Paper Structure. The structure of this paper is as follows. In section 2, we first give the background details of this project. Next in section 3 we sketch the limitations of existing JML tools. We then in section 4 elicit the objectives of the JML group in general and our goals for this project. Next in section 5 we discuss about several approaches to translating assertions into runtime checking code. In the next section, we give an overview of the existing Eclipse architecture and that of the JML4. In section 7 we present our proposed architecture, which actually contains several approaches. Section 8 and 9 talks about the implementation and bytecode architecture. In section 10 we enlist the evaluation criteria/methods that we would employ.

2. BACKGROUND

In this section we give a brief introduction to runtime assertion checking, extensible compiler construction, abstract syntax tree and Eclipse IDE.

2.1 Runtime Assertion Checking

Assertions are formal facts about the state of a program; they are statements that are true at certain points in program code [4]. They are very useful for both debugging and proving correctness of programs [5]. There may be several ways to support assertions in programming languages, but one of the most popular approaches is to use macro statements that are expanded into appropriate program statements by preprocessors. The main examples are the assertion facilities of C [6] and C++ [7] [8] (e.g., the assert macro).Meyer promoted simple assertions into what is referred to as the *design by contract* (*DBC*) [9][10].

This work was done as a class project for CS5381 under Dr. Yoonsik Cheon for Spring, 2008 at University of Texas at El Paso. This report also documents partially the work that the author did as a research assistant under Dr. Cheon. This report may be used in future as a memo for JML Group to address the issue of extensibility and support runtime assertion checking in Eclipse

2.2 Extensible Compiler Construction

While processing of programs is an old research topic, there are still plenty of opportunities for improvement when it comes to extensibility. This is particularly true for tools (like JML) that need to support full languages and process large bodies of code. Early stages in compilers such as scanning and parsing are well understood and there exists techniques for extensive specifications [2]. However in later stages, viz. name binding, type checking, and code optimization, they rely on context-sensitive and are still often hand coded in an ad-hoc fashion with little support for extensibility (like Eclipse).

2.2.1 *Modularity*

Modularity is often used to describe the possibility to decompose a system into modules [14]. This allows each sub problem, or concern to be studied in isolation, a property which is known as separation of concerns [15]. There may be several decomposition criteria, all of which are desirable depending on the situation. Some of them are –

• *Separate Computations* The goal is to be able to reuse and combine the different computations to obtain different tools, and to express and understand each computation in isolation.

• *Language Extension* The goal is to be able to reuse the base language implementation for many different extensions, to combine extensions, and to be able to express and understand each extension in isolation.

• *Language Specification* The goal is to provide traceability between the language specification and the implemented compiler.

2.2.2 *Scalability*

It is important that a compiler technology is scalable both to full languages and large sized applications to be of practical use. Mainstream programming languages are often complex and contain many corner cases. Even small applications tend to use most language features in Java and fail to compile if only a subset is supported. It is also worth noticing that for a tool to be useful in an industrial setting it needs to be able to analyze code bases in the range of a hundred thousand lines of code or more.

2.3 Abstract Syntax Tree

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`. This comprises of the Abstract Syntax Tree (AST) for the Java language, as well as objects that perform operations on the AST. The knowledge of AST is required for us to better understand – where and how to merge (if necessary) our RAC generated AST with the existing AST (developed by the native compiler).

2.3.1 *AST class:*

The AST class is the owner of the AST. Any new AST nodes created by using an object of this class will be owned by that object. The class has two static fields that are used to specify which version of the Java Language Specification (JLS) should be used when parsing source code. `AST.JLS2` refers to the second version of the JLS, while `AST.JLS3` refers to the third version. The AST class also acts as a factory for producing ASTNode classes. A node of any type can be created using a method of the form newXXX where XXX is the name of the syntax element to be created. Each node that is created in this way does not have any type name or value specified. The node also has no parent. Finally, AST provides a utility method `resolveWellKnownType`(). This method takes in a String which names a well known type. It returns an ITypeBinding, which is an interface that represents a well known type.

2.3.2 *ASTNode class*

The ASTNode class is the superclass for the many AST node types. An ASTNode represents a syntactic element in the Java language. Each node has links to each of its children, as well as to its parent node. Therefore, the AST can be traversed either from the top down, or from the bottom up. In addition, each ASTNode object contains the range in the source file where the syntactic element can be found. The `getStartPosition()` method returns an index into the source file where the element starts, and the `length()` method returns the number of characters that comprise the element.

2.3.3 *ASTParser class*

The ASTParser class is responsible for converting source code into an AST. There are no constructors to use for this class. Instead a static factory method called `newParser()` is used to create a new ASTParser. The argument to this method specifies the level of the Java Language Specification to use. The `setSource()` method specifies the source to compile. The `createAST()` method will create the AST from the source that was given to the object. It returns an object of type ASTNode, which will represent the root of the produced AST. A useful method that this class provides is `setProject().` This method takes in an IJavaProject object that is used to specify a Java project on the workbench.This Java project will be used to resolve types in the source string that otherwise could not be resolved by the compiler.

2.3.4 *ASTVisitor class*

To perform operations on an AST, we use the ASTVisitor class. ASTVisitor is an abstract class. It provides two operations to be performed on every node of an AST. The `visit()` method returns true if the node has children that will be visited after the current node is visited. The `endVisit()` method is similar to `visit()` except that the children of the node will be visited before the node itself is visited. In addition to all the type-specific visit operations, there are two operations that perform work on an ASTNode in general, and not on specific types within the AST hierarchy. The `preVisit()` method is used to visit an ASTNode before the type-specific visit operation is called on that node. The `postVisit()` method visits the ASTNode after the type-specific visit operation on that node.

2.4 Eclipse IDE

Eclipse is a plug-in based application platform. Well known bundles 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

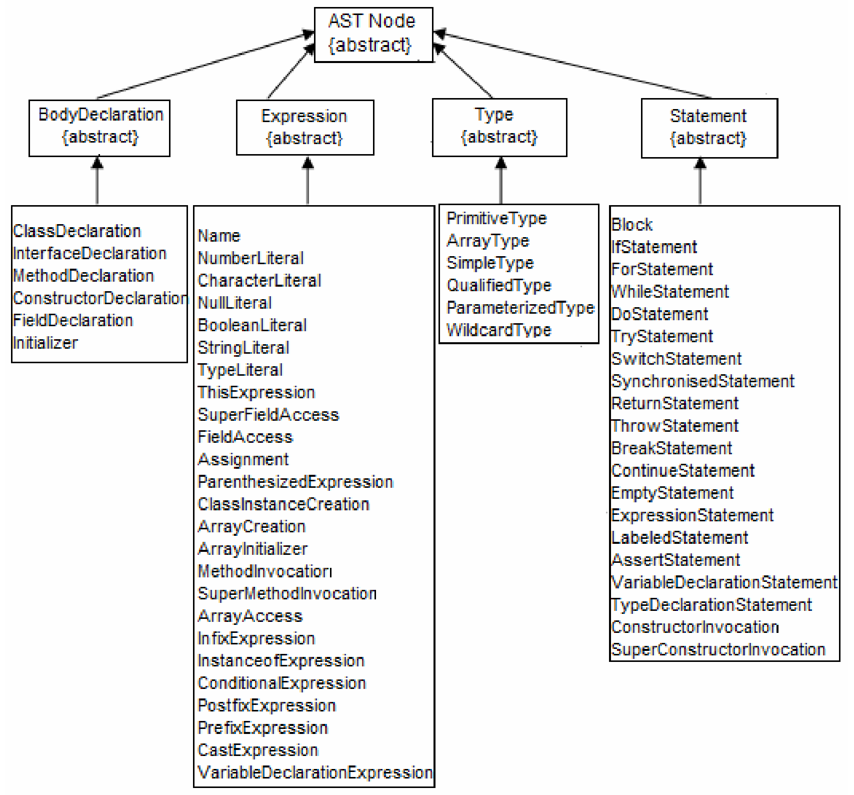


Fig1. The AST hierarchy

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 JDT Core plug-in provides a JCK-compliant Java compiler with incremental recompilation, which allows high performance and scales up to large projects.

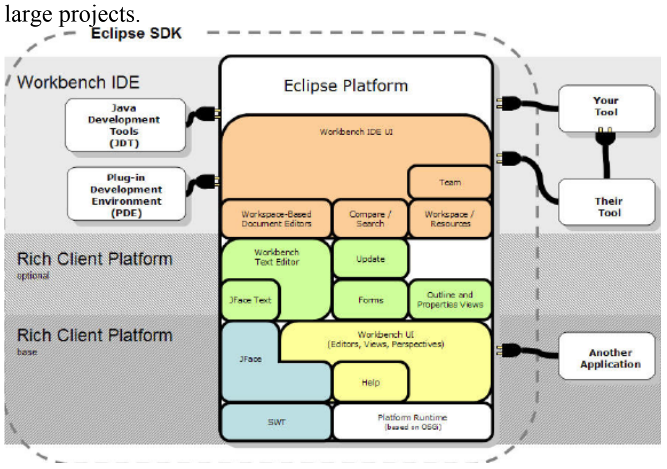


Figure 2. The Eclipse Architecture showing the Eclipse SDK and RCP

3. LIMITATIONS OF JML2

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 [16], and
- LOOP a full static program verifier [17].

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.

3.1 Problem in Existing JML Infrastructure

Despite JML's demonstrated potential, its inadequate infrastructure threatens its continued use in this role. The initial version of JML does not support sufficiently robust tools.

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

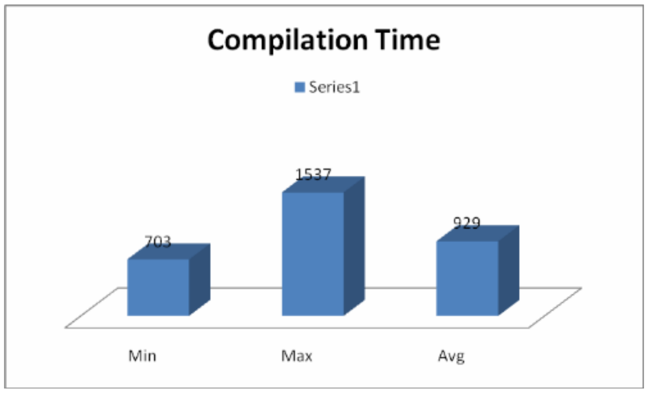
• many JML features are naturally implemented by subclassing MJ features and overriding methods—e.g. Abstract syntax tree nodes with their associated type checking methods;

• 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.

The multijava compiler was not designed for extensibility; after much extension and refactoring to support JML it has become unwieldy and difficult to maintain. Extensibility is needed not just to track evolution in the programming language, but also to allow experimentation in specification language design.
Version 5 of the Java language substantially extends Java with any additional features, most notably generics. No JML tool properly supports generics. The common JML Tools do not support other features of Java 5. Researchers want to evaluate their tools on current (Java 5) benchmarks.
This puts great pressure on fundamental tool support: parsing and basic compiler infrastructure have to be frequently updated to track the language's evolution. The first generation JML tools 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). 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.
Another issue is practical adoption of building integrated development environments (Ides).

3.2 Performance & Size Issue

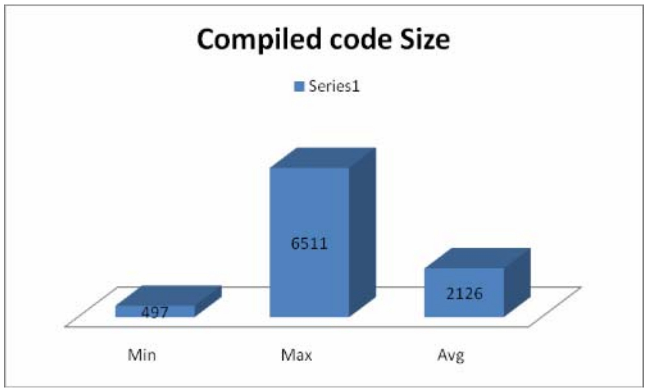
The existing JML2 tools, more importantly 'jmlc' the runtime assertion checker from the performance point of view is really very slow. The compilation time taken is huge compared to the compilation speed of 'javac'. However, it is evident that since jmlc does more work than javac, it would take more time. The following graph depicts that the minimum time difference between the two compilation tools is ~700%, whereas the average time difference is ~930%. The JML group at large had not forseen such an enormous difference.



On compilation, another important criteria is the bytecode size. This becomes of utmost importance when space is of prime importance. In this graph too, we see a huge difference in terms of %difference between the size of the compiled code. The maximum difference is almost 7000 percent. That is, about 70 times the size of a .class file compiled using javac.

The results however need to be further reviewed to come to a proper conclusion.

This paper focuses to come up with such an architecture which would help us to decrease time and size of the compiled code.



4. OBJECTIVES AND GOALS

In this section we elicit the objectives behind JML4 and a possible extension to it. We also present the goals of this project.

4.1 Objectives

Resource constraints and architecture of the current tools of JML prevent their thorough integration into a development environment such as Eclipse. The tools are written in a tightly integrated way that is not flexible and extensible. Some of the objectives of JML group are –

• The infrastructure must be extensible. They must smoothly accommodate future changes to the Java programming language.

• The proposed software infrastructure must keep with the evolution of Java. It must support the features of Java 5 and Java 6, and be extensible to future versions of Java.

• Allowing the language itself to be extensible implies that the tool architecture should also be modular and designed for extension. The modularity and extensibility of the architecture is very crucial.

• Apart from Eclipse, the software infrastructure should also be integrated to other Ides.

4.2 Runtime Assertion Checking Features

The JML group in their initial draft had come up with a comprehensive study on how to divide JML statements into different levels. This partitioning of features was mainly due to the complexity of understanding and usage from the user point of view. The table shown in Appendix A-3 provides us with a feature table that would enable us to figure whether our prototype or architecture conforms to the basic and/or advanced needs to support primarily runtime assertion checking code. This code helps us to validate our proposed architecture against these features.

4.3 Design Goals

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. 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. However, this is not always the best approach, since existing compilers may not have been designed with extensibility as one of the main goals. Furthermore, they may be constrained to work with infrastructures which themselves are not easily extensible.

Our approach was to design and implement *A JML Compiler on Eclipse Platform*, with extensibility as its primary design goal. 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.

To support our objectives, we distilled the following requirements from the above discussion involved.

• *Extensibility:* The architecture so designed should support extensibility. The JML features should be incorporated into the Eclipse framework in such a manner so that minimum extension points are used so that the extensibility of Eclipse is not hampered.

• *Simplicity:* It must be relatively simple to develop new extensions. Users of the framework should not need to understand complicated new concepts or a complex software design in order to implement their extensions.

• *Modularity:* We require two kinds of modularity. First, the workbench itself should be very modular, so that the different facets of each extension can be easily identified with the correct module of the workbench. Second, the extension should be modular (separate from the workbench code). Users of the workbench should not need to touch existing code; rather, they should be able to describe the extensions as specifications or code that is separate from the main code base.

• *Proportionality:* Small extensions should require a small amount of work and code. There should not be a large overhead required to specify an extension.

• *Analysis capability:* The workbench infrastructure should provide both an intermediate representation and a program Analysis framework. This is necessary because some extensions may lead to a lot of runtime overhead unless compiler optimization techniques are used to minimize that overhead.

Some of the other measurable goals that can be incorporated are Time and Space. We should be able to reduce time substantially than the former JML tools. For achieving so, we require to judiciously choose between our several architectural approaches. Another important criteria is the size of the compiled code must not be enormously high as in previous tools.

4.4 Contributions

The contributions of this paper are the following –

• We have identified the requirements for a workbench for extending JML tools by analysing previous research in this area.

• We present *A JML Compiler on Eclipse Platform*, an instance of such a workbench with a clean, extensible architecture.

• We have enlisted some key points, features of runtime assertion checker which would help us to validate our architecture against these requirements.

• The extensibility of Eclipse can be seen as a form of substantial exercise in extensible software development, with the primarily goal of disentangling features from the existing base compiler.

5. JML RAC

An essential requirement for runtime assertion checking is transparency; unless an assertion is violated and except for performance measures (time and space), the behavior of the original program should be unchanged. Another requirement is that runtime assertion checking should reflect the semantics of JML. It should be sound in that it does not produce any false positives [18]. The runtime assertion checker should also strive to detect as many potential errors – inconsistencies between specifications and code – as possible; ideally, it should be complete in that it detects all such errors.

5.1 Assertion Blocks, Methods, and Classes

An important design dimension is to define the structure of runtime assertion checking code and the way it interacts with other assertion checking code and the code being checked. There are several possibilities in organizing assertion checking code:

• *Blocks of Java statements:* A specification may be translated into a sequence of Java statements, called an *assertion checking block* or *assertion block* for short. The assertion block may be injected into the appropriate position in the method body.

• *Separate methods:* An assertion checking block may become a separate method of the class being checked, called an *assertion checking method* or an *assertion method* for short. To check an assertion, the assertion method is called in place of the assertion block.

• *Separate classes:* The assertion methods, instead of being members of the class being checked, may form a separate class, called an *assertion checking class* or an *assertion class* for short. To check an assertion, an assertion object is created and an appropriate assertion method is called on the assertion object.

5.2 Strategies for Assertion Support

Assertions may be specified at the class level (invariants) or on the method level (preconditions and postconditions). A number of systems exist for the Java programming language that support assertion techniques in different ways. The goal of this section is to describe different possible approaches for supporting contracts for the Java programming language. Generally three different approaches are possible:

• *Built in/Compilation-based approach:* This means that support for contracts is directly included in the programming language. The programming language contains language constructs to formulate assertions in one way or another. The syntactical correctness of assertions is checked directly by the compiler. In addition a runtime environment must be available to perform the runtime assertion checks. Ideally the runtime environment is flexible enough to allow a very fine-grained control of the assertion checking mechanism, i.e., it should be possible to selectively enable and disable assertion checking. The main advantage of this approach is the homogeneous integration of assertions into the programming language, i.e., compiler error messages are consistent, debugging tools can properly consider assertions (e.g., correct line numbers and stack traces).

• *Preprocessing:* This is the most popular kind of support for assertions in a programming language. The general idea is to formulate assertions separate from the program or to include the assertions as comments. A preprocessor is used to weave the assertions into the program or to transform the comments containing assertion formulas into programming language code. The main advantage of this approach is the separation of programming logic and contracts. This is important in cases, where the programming language itself does not support assertions and the programming language must not be altered for various reasons (e.g., conformance to standards, not enough knowledge available for changing the compiler). The main disadvantage of this approach is that the original program code is changed by the preprocessor, i.e., line numbers of compiler errors do not actually fit the line numbers of the program. The same problem arises with debugging or runtime exceptions.

• *Meta-programming:* Programs that have the possibility to reason about themselves have so called reflective capabilities. The Java programming language, e.g., has reflective capabilities and may access information about elements of a Java program by means of a reflection API. The main advantage of meta-programming approaches is that no specialized preprocessor has to be used but the native compiler. Nevertheless a specialized runtime environment has to be used to enable assertion checking.

• *Byte Code Manipulation:* Another approach for languages based on virtual machines, such as Java, is to manipulate the virtual machine's bytecode to inject assertion checking code. The manipulation can be done either at compile time or loading time, e.g., using a customized class loader.

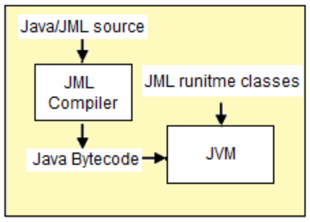


Figure 3. Built in approach to runtime assertion checking.

In JML, the compilation-based approach is adopted, as it is an intuitive and easy-to-use approach (see Figure 3). JML compiler is essentially a Java compiler with additional capability of translating JML specifications into automatic runtime checks.

6. ECLIPSE JDT ARCHITECTURE

Before we discuss about the Eclipse Architecture in detail, we present some of the issues that are inherent about Eclipse. Eclipse does not create applications with true Java functionality. Developers must port the SWT to all platforms on which Eclipse runs, which can be complex, time consuming, and expensive.

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*.

6.1 Compilation Phases Overview

The main steps of the compilation process performed by JDT are illustrated in Figure 4. In the Eclipse JDT (and also in 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. 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. Finally, flow analysis and code generation are performed.

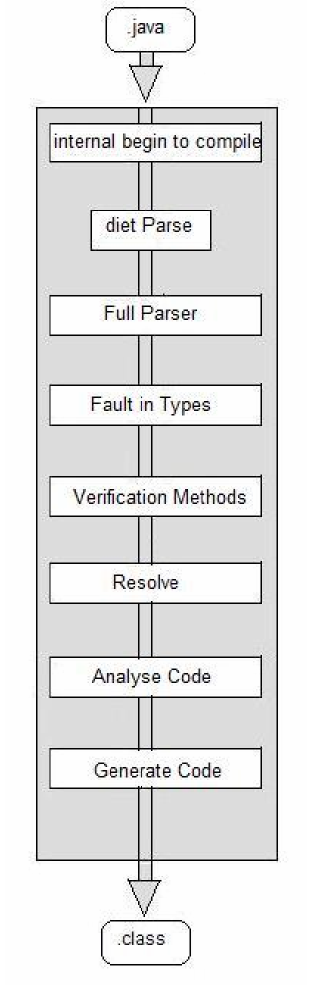


Figure 4. JDT compilation phases.

6.2 Components of JDT

This section describes very briefly the most important components of the JDT compilation phases.

6.2.1 Scanning
Scanning of source code is done at a much later stage than anticipated. When each CU is diet parsed by invoking the `DietParse()` method, it in turn calls the `Scanner()` method which actually scans the source code that is linked/bounded to the CU.

6.2.2 Parsing
The JDT's parser is auto-generated from a grammar file (`java.g`) using the Jikes Parser Generator (JikesPG) and a custom script that resides at `org.eclipse.dt.core/scripts`. The grammar file, `java.g`, closely follows the Java Language Specification[19]

6.2.3 Type Checking
Type checking is performed by invoking the `resolve()` method on a compilation unit.

6.2.4 Flow Analysis
Flow analysis is performed by the `analyseCode()` method on a compilation unit.

6.3 Eclipse back-end Compiler
Contrary to popular belief, the Eclipse framework has just one back-end compiler support. This section discusses the overall interaction between the command-line tools, GUI and the compiler. Figure 5 illustrates the interactivity between the different APIs. The command line API (at an abstraction level) contains 3 methods basically. The `main()` method is the starting point of interaction with Eclipse via command prompt. It receives the arguments and in turn invokes the `compile()` method. This method decodes the command line arguments and call the `performCompilation()` method. This method initializes the Batch.Compiler to its default settings and passes the CompilationUnits denoted as CUs to `internalBeginToCompile()` method, which basically is the starting point of the compiler API.
In GUI case, on invoking the Eclipse framework, several threads concurrently starts, of which the `run` thread is invoked from `org.eclipse.core.internal.jobs.Worker.run()` method. This run thread in turn calls the `build()` method in the BuildManager class. This invokes the `basicbuild()` method. If the user now, writes some code and builds/saves it, automatically `builddeltas()` method is invoked. This method tells the framework only those resources that have changed since the last build need to be considered for compilation. The delta only tells you the file was changed. If any delta is found, they are send to `incrementalbuild()` method, which in turn invokes `incrementalcompiler()` method. This method identifies which CU is to be built. And then sends this to the `jdt.compiler` class.

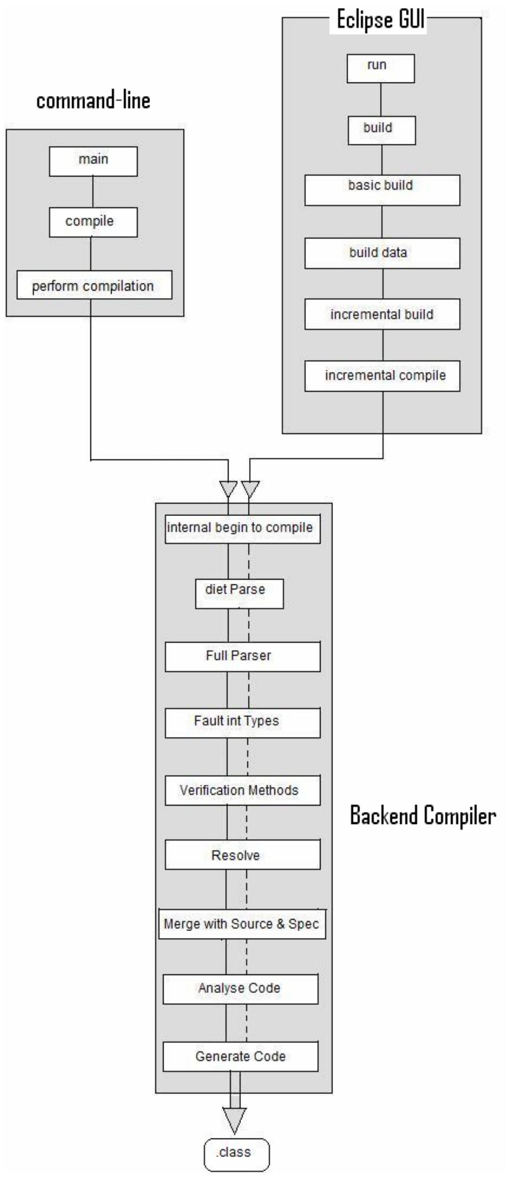


Figure 5. Interaction between command-line tools, Eclipse GUI and the backend compiler.

7. PROPOSED TOOL ARCHITECTURES
The new proposed architecture has several alternatives. Each of them is presented below.

7.1 Double Round Approach
The JML2 runtime assertion checking method[11] has been implemented in the context of Eclipse Architecture. The approach for implementing the JML RAC compiler is to reuse the existing source code of JML and Eclipse tools as much as possible, if necessary, by refactoring it. The Eclipse type checker and its underlying Java compiler provide a good code base for the JML RAC compiler. They consist of several compilation passes. Our idea is to introduce new compilation passes to generate assertion checking code, and to rewire the whole compilation passes to

generate bytecode for both the original and assertion checking code.

Ideally, we would like to have a minimal duplication of compilation passes. However, the complexity of assertion checking code and the infrastructure of existing tools make such an optimal solution difficult, and which lead us to a strategy, called *double-round compilation*[11]. Figure 5 shows the architecture of the Eclipse compiler. The original path of Java file not annotated with JML specifications would flow through Scanner, Parser, and then to Static analysis and code generation.

To implement the double-round compilation strategy, we added a new compilation pass "RAC code" after the typechecking and static analysis pass. This pass generates runtime assertion checking code from the typechecked abstract syntax tree. It generates actual source code and then combines with the existing source code. This newly generated code is again sent through the compilation phases for the second time. This time it directly goes from static analysis to code generation as depicted using the red dotted lines.

7.1.1 Discussion

The main advantage of this approach is the separation of programming logic and contracts. It is the simplest approach for runtime assertion checking.

The main disadvantage of this approach is that the original program code is changed by the preprocessor, i.e., line numbers of compiler errors do not actually fit the line numbers of the program. The same problem arises with debugging or runtime exceptions. Another disadvantage from the performance point of view is that the existing/original source code undergoes the entire compilation path (excluding code generation) twice, thus increasing the compilation time.

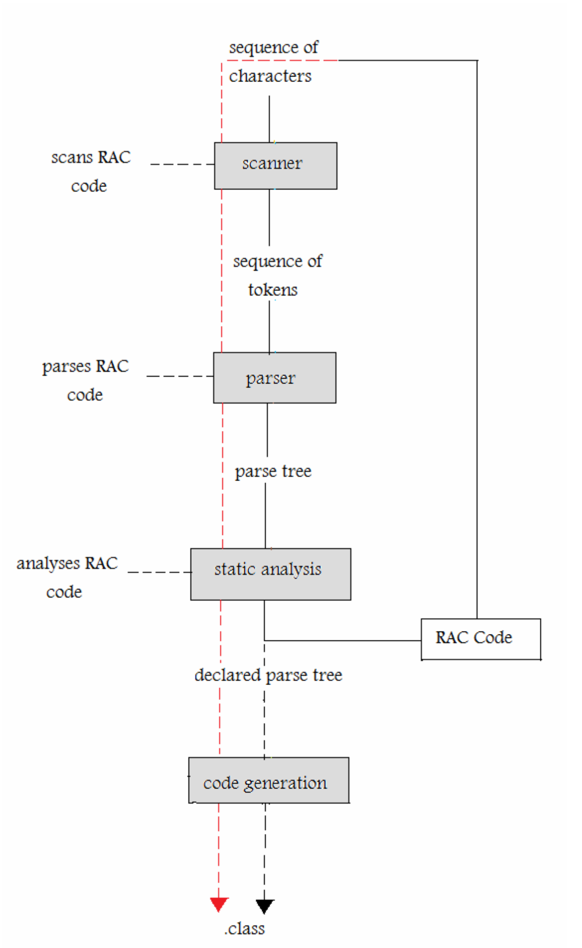


Figure 6. Double-round compilation in Eclipse

7.2 Incremental Approach

The next approach called the *Incremental Approach* works on the same fashion as the Double-round approach. However this time, the RAC Code Generator generates only the runtime assertion checking code and sends only the rac code to the scanner. It stores the context to which the new AST trees would be binded. This flow is depicted through the red dotted line in figure 6, where only new runtime assertion checking code would be scanned, parsed, analysed. After static analysis the new AST tree would be mutated to the original AST tree. This would give us a combined AST that in turn we can send to the code generation component for generating .class file. We could actually implement this approach by examining in closer detail how the different components interact with AST and how they deal with Type Bindings. A key component of this interaction is the separation of the formation of AST nodes and Binding them to their ParentAST node. The knowledge of building the types enables us to use the existing facility in Eclipse for combining a RAC AST into the original AST.

Currently, due to the complexity of Eclipse architecture, we have not yet figured it how to extend and refactor the existing architecture so that it still fulfills our goals and make this approach feasible. We must also say that the Eclipse framework does not compile snippets of code by itself. The unit of incrementation by Eclipse framework is a file.

7.2.1 Discussion

The main advantage of this approach is we still achieve to separate programming logic from contracts. In addition, we believe that the computation time would be greatly reduced. The reason behind this is that even though it goes through the compilation phases for a second time, it only compiles the added code. The only reason for an increase in time in this approach would be the time taken to merge the two ASTs.

However it still suffers similar to the previous method. The original program code is changed by the preprocessor, i.e., line numbers of compiler errors do not actually fit the line numbers of the program. The same problem arises with debugging or runtime exceptions. However, presently the most difficult part of this approach is how to merge the two ASTs.

7.3 Byte Coding Weaving Approach

This approach is possibly one of the solutions to the above problem i.e. the difficulty in compiling only a portion of the code namely runtime assertion checking code in the context of the original source code. This approach takes totally a different path altogether. It does not go through the complexity of merging the AST nodes. In this approach, after static analysis, the flow branches out into two paths. One is the original path to code generation and other to a new component called "RAC code Generator". It takes declared parse tree as an input to this component. This component generates only runtime assertion checking source code and further sends this source code back to the compilation phases. Unlike, the JML2 approach, in this approach only the runtime assertion checking code is compiled. Having compiled, the code generator generates the corresponding bytecode. After this bytecode has been generated, we can use weaving technique to manipulate and merge the byte code of the runtime assertion check to the original source code.

7.3.1 Discussion

The advantage of this approach is that direct bytecode manipulation would alleviate the problem of merging ASTs. Our assumption is that bytecode manipulation would be easier to implement than merging two ASTs. In comparison to the incremental approach, less of code refactoring is done. This helps us to maintain the Eclipse JDT framework. Another distinct advantage over Double Round approach, is the time factor. We presume that even though in this approach we travel twice in the compilation path, the time taken for compilation would be less because in the second round only the new runtime assertion checking code would be compiled.

One major problem, is the feasibility of whether such an approach is actually possible. Currently we are doing a feasibility study. The main concern is how easy (from the implementation point of view) would be to actually manipulate byte code. Another concern, is whether it would be possible to resolve the different type-bindings between two such bytecodes.

7.4 AspectJ Approach

We have not yet started working for this approach. However we think that this approach would help us to achieve our design goals.

8. IMPLEMENTATION STRATEGY

The above approaches have been implemented into the Eclipse architecture taking care such that only public APIs are changed.

8.1 Double Round Approach

Following the discussions above, we can proceed to implement the JML2 approach as per the following steps –

• We require to make a call to `Preprocessing()` component which actually process the runtime assertion code and the original source code. This call is made before the actual generation of byte code. This method takes in the existing CU and returns a new CU which is in Intermediate Representation format, compliant for direct bytecode generation. Appendix A-1 illustrates how the method `Preprocessing()` is inserted into the existing `process()` method in Compiler.java.

• In the `Preprocessing()` method, the new compilation unit so generated would require to go through the same steps as the original source code had gone. In Appendix A-2 the body of the method is shown. This method calls another method in Parser.java class which actually creates the new runtime assertion checking code and merges with the previous source code to get a new source code.

• Another component needs to be added in the Parser.java class. This component named `addWrapperMethods()` actually injects new wrapper methods into the existing source code. For this time, we have hand-coded the generation of new wrapper methods. This would be automated in the later stages. However, this automation is possible, as shown in [11] dissertation thesis.

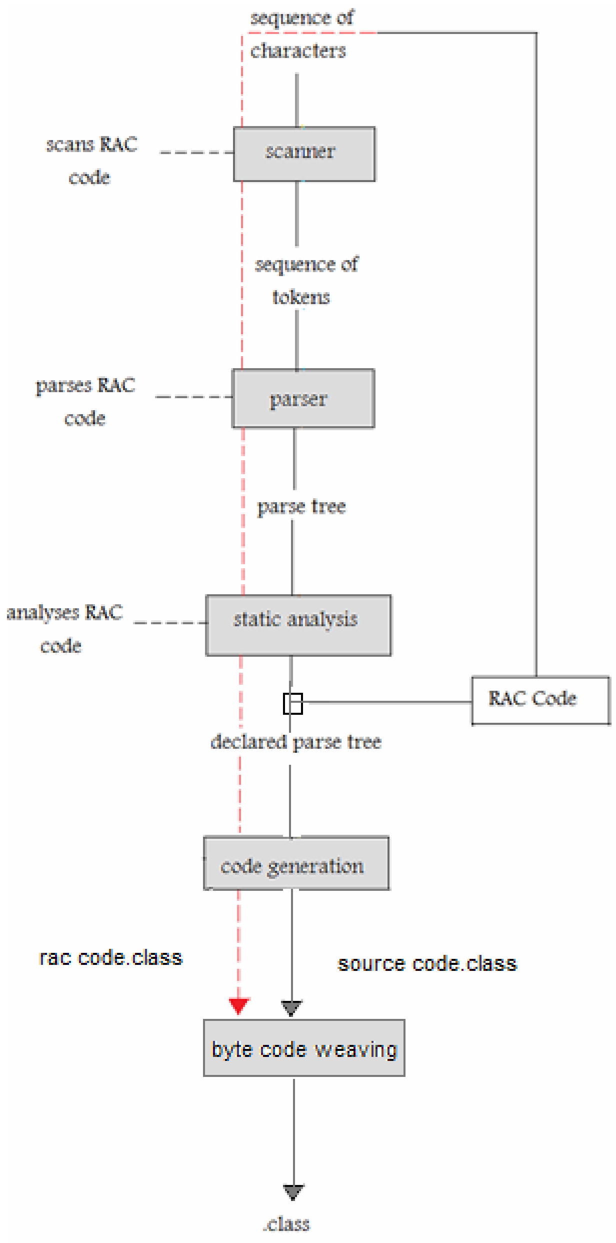


Figure 7. ByteCode weaving approach

8.1.1 Discussion

The changes that has been incorporated in this approach, into the Eclipse framework have been done in confirmation to the Eclipse community as well as the JML group. New code has been essentially written only in two packages –

• `org.eclipse.jdt.core/compiler/org/eclipse/jdt/internal/compiler`

• `org.eclipse.jdt.core/compiler/org/eclipse/jdt/internal/compiler/parser`

Another aspect was the addition of minimal extension points. In this approach we have added just one extension point.

8.1.2 Conformance to Design Goals

The newly added code very much adheres to the design goals. They are simple and modular. It is modular because the new components are very much separate from each other. That is, the component Preprocessing only preprocesses the new CU, while addWrapperMethods actually adds runtime assertion checking code into the original source code. It also conforms to the Proportionality goal – it required a small extension and we indeed did this by just adding 17 lines of code into the existing architecture. These 17 lines are responsible for the double-round compilation technique. Since the design was itself modular and the extension point has been kept to just 1, analysis can easily be done into the new architecture, as if no change has been done to the existing Eclipse framework at all.

8.2 Incremental Approach

Currently we have not been able to implement this approach and integrating it with the Eclipse JDT framework. As we had explained previously, that this approach primarily talks about compilation only of the RAC code (during the second compilation path). This means we require to parse, bind and static analysis this code on the context of the actual source code. The reason for our inability to implement this approach is that Eclipse JDT framework does not support incremental compilation at the method or more grainer level. Eclipse supports incremental compilation at the class level.

One approach towards finding a solution is to somehow (which we need to find) tell the ASTParser class under which context should it be typed into. This would help us in merging the two ASTs. Once we can get these two ASTs we can easily generate the bytecode for it using the `generate()` method provided by Eclipse framework itself.

8.3 Byte Coding Weaving Approach

Currently we are undergoing an extensive study on this approach, that would enable us to gather sufficient knowledge which would help us to implement this approach. Some current methods available are BCEL, ASM, etc. We are currently studying them, and in future we may also come up with our own Byte code Manipulation tool. This is primarily because the available tools may not suite our need completely, and by making our own tool we may not depend on a third party plugin tool.

9. BYTE CODE ARCHITECTURES

This section discusses about the several approaches through which we can actually finalize the internal format of runtime assertion checking code. Method specifications are translated into runtime assertion checking code following some steps. The last step is the attaching of assertion checking code to the original code. This step is for injecting the assertion checking code into the appropriate place of the original code.

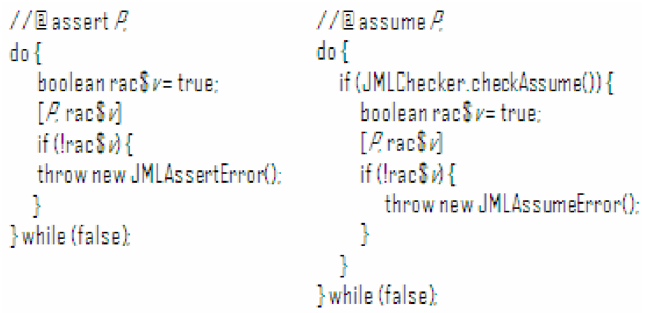
How is the assertion checking code injected into a method so that, for example, the method's pre- and post conditions are checked before and after the execution of the method body? There are three possibilities viz. in-line approach, a wrapper approach and a semi- wrapper approach. .

9.1 In-line Approach

An in-line assertion, also called an intracondition, is an assertion that can be specified in the method body. In this approach, the assertion checking code is inserted directly into the body of the method being checked. For example, the precondition checking code becomes the first statement (or a block of code) of the method body. In JML, an in-line assertion is treated as a statement, and thus can appear where a Java statement is allowed. JML provides several kinds of in-line assertions, such as assert statements, assume statements, hence by statements, unreachable statements, set statements, and loop in-variant and variant statements.

9.1.1 Importance

JML statements like assert, assume, hence_by, etc. can be realized better using In-line approach. An assert statement is a specification statement containing a boolean expression that must hold when the control reaches the statement. An assume statement is a specification statement that specifies an assumption that the programmer makes on the program state when the control reaches the statement. In-line approach is best used for those JML statements which are statement specific. This approach is simple and efficient; it does not incur extra method calls for assertion checking. Another very important use of in-line assertions are to check the preservation of properties specified by type assertions from client-visible state. This is achieved by injecting assertion checking code directly into the client code for each reference of public fields.



```
//@assert P;
do {
    boolean rac$v= true;
    [P, rac$v]
    if (!rac$v) {
    throw new JMLAssertError();
   }
} while (false);
```

```
//@assume P;
do {
   if (JMLChecker.checkAssume()) {
      boolean rac$v= true;
      [P, rac$v]
      if (!rac$v) {
         throw new JMLAssumeError();
      }
   }
} while (false);
```

Figure 8. Translation of assert and assume JML statements

9.1.2 Short-comings

The in-line approach has two shortcomings. First, it is not trivial to inject assertion checking code of the post-state assertions such as normal and exceptional postconditions, invariants, and history constraints. The assertion checking code may not be added at the end, because the method body may have return statements. Second, the approach does not facilitate a modular way of implementing specification inheritance. The assertion checking code cannot be inherited by subclasses, as it is embedded into the method body. For subclasses, assertion checking code must be regenerated or textually copied down from superclasses and implemented interfaces (which may need renaming and other modifications).

9.2 Wrapper Approach

A wrapper approach is used to check method specifications. Each method is transformed into a private method, and instead a new wrapper method is generated with the same name and signature. As a result, all client calls to the original method now go to the wrapper method. The wrapper method is responsible for transparently checking method specifications. For this, the wrapper method delegates client calls to the original method wrapped with appropriate assertion checking. It calls pre-state assertion methods such as preconditions and pre-state invariants before delegating the method call; it calls post-state assertion methods such as postconditions, post-state invariants, and constraints, after delegating the method call. This new wrapper method is created as a separate class. From the client point of view, this has the effect of checking pre-state assertions in the pre-state and post-state assertions in the post-state.

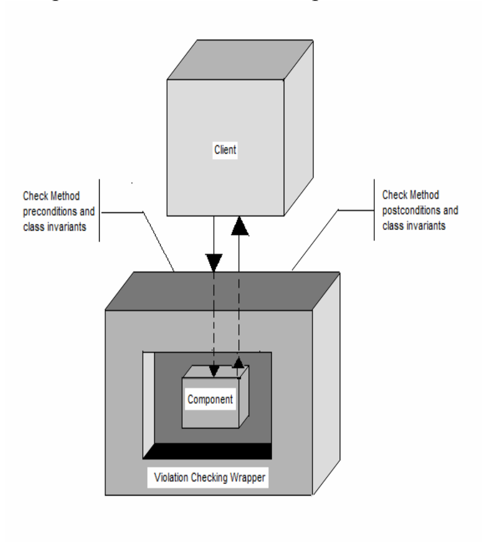

Client
Check Method preconditions and class invariants
Check Method postconditions and class invariants
Component
Violation Checking Wrapper

Figure 9. Conceptual understanding of Wrapper approach

9.2.1 *Importance*

The wrapper approach is better structured and organized as the instrumented code is modularized with wrapper methods and assertion checking methods. The approach also facilitates specification inheritance; a subclass can call the corresponding assertion checking methods of its superclasses to inherit specifications.

9.2.2 *Short-comings*

The only disadvantage of wrapper approach is its performance and space issues. Due to several calls of different wrapper methods, compilation-time of the code increases. Due to addition of our own wrapper methods and creating new .class files, the size also increases a lot, in comparison to the in-line approach. Another short coming of this approach is, the implementation of statement- specific JML statements like assert, assume, is very difficult.

9.3 Semi-Wrapper Approach

One major difficulty is to the synchronization between two or more class files, due to generation of neew class files for wrapper methods. From the above discussion we can very well conclude that implementing our JML RAC using any of the above two approaches would be difficult. Hence, we propose a third approach – a hybrid approach which implements wrapper approach in in-line style. In this approach we create new wrapper methods and embed them into the original source code itself. Thus we do not create a new class file for the wrapper methods.

9.3.1 *Importance*

Since this approach is possibly an improvement than the wrapper approach: it benefits from the pitfalls of it. Since we have been able to negate the extra overhead between different files this potentially increases the performance.

9.3.2 *Short-comings*

However it still falls short in coming up with an approach that would enable us to not only decrease computation time but also the compiled size code. This approach still has overhead which in reality increases the time complexity and compiled size code.

10. EVALUATION

We would be evaluating the proposed architectures against our design goals and the RAC features. We would be creating prototypes for the proposed architecture. We should also create exhaustive test cases that we can test against the individual prototypes. The results from the test runs and static analysis would help us to come up with one architecture which would help us to achieve our Designed Goals.

11. CONCLUSION

In this paper, we have outlined a strategy for extending eclipse framework to incorporate JML. In particular use runtime assertion checking on eclipse platform. This strategy is not without challenges, however. Two of the more troublesome are finding the right extension points and minimal change in the actual eclipse source code. The architecture that would be eventually chosen must adhere to certain specific criteria.

Acknowlegements

We gratefully acknowledge the support extended from Dr.Yoonsik Cheon, research advisor who has always guided us whenever we were stuck with a problem.

12. REFERENCE

[1] Patrice Chalin, Perry R. James, and George Karabotsos. An Integrated Verification Environment for JML: Architecture and Early Results. *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 47-53, September 2007.

[2] G. T. Leavens, “The Java Modeling Language (JML)”: http://www.jmlspecs.org, 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] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576{583, October 1969.

[5] Jurgen F.H. Winkler and Stefan Kauer. Proving assertions is also useful. *ACM SIGPLAN Notices*, 32(3):38{41, March 1997.
[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cli®s, N.J., 1978.

[7] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Co., Reading, Mass., 1990.

[8] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.
[9] Bertrand Meyer. Applying \design by contract". *Computer*, 25(10):40{51, October 1992.

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

[11] Yoonsik Cheon. A Runtime Assertion Checker for the Java Modeling Language. Department of Computer Science, Iowa State University, TR #03-09, April 2003.

[12] R.P.Tan, S.H.Edwards. An Assertion Checking Wrapper Classes for Java. SAVCBS' 03 Helsinki, Finland.

[13] F. Oquendo. Formally modeling software architectures with the UML 2.0 profiles for π-ADL, *ACM SIGSOFT Software Engineering Notes,* 31(1):1-13, January 2006.

[14] A framework for software maintenance metrics Pfleeger, S.L.; Bohner, S.A. Software Maintenance, 1990., Proceedings., Conference on Volume , Issue , 26-29 Nov 1990 Page(s):320 - 327

[15] Dijkstra, E. W. Self-stabilization in spite of distributed control. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, New York, 1982, pp. 41–46.
[16] 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.
[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] Robert Bruce Findler and Matthias Felleisen. Contract soundness for object-oriented languages. In *OOPSLA '01 Conference Proceedings, Object-Oriented Programming, Systems, Languages, and Applications, October 14-18, 2001, Tampa Bay, Florida, USA*, pages 1-15, October 2001.
[19] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison-Wesley Professional, 2005.

APPENDIX A-1

```
/**
 * Process a compilation unit already parsed and build.
 */
public void process(CompilationUnitDeclaration unit, int i) {

     this.lookupEnvironment.unitBeingCompleted = unit;

     this.parser.getMethodBodies(unit);

     // fault in fields & methods
     if (unit.scope != null)
          unit.scope.faultInTypes();

     // verify inherited methods
     if (unit.scope != null)
          unit.scope.verifyMethods(lookupEnvironment.methodVerifier());

     // type checking
     unit.resolve();

     // <jml-start id="6" />
     if (this.options.jmlEnabled) {
          this.jmlSourceLookup.mergeWithSourceAndSpec(unit);
     }
     // <jml-end id="6" />
     // flow analysis
      unit.analyseCode();

     // Second time-processing for RAC
     Unit = Preprocessing(unit);

     // <jml-start id="extension" />
```

```
          if (this.options.jmlEnabled)
               CompilerExtension.preCodeGeneration(this, unit);
           // <jml-end id="extension" />

          // code generation
          unit.generateCode();

          // <jml-start id="6" />
          if (this.options.jmlEnabled && this.options.jmlNullityCountsEnabled) {
               CompilationUnitScope scope = null; //new CompilationUnitScope(unit,
this.lookupEnvironment));
               unit.traverse(new ReferenceCounterVisitor(this.problemReporter), scope);
          }
          // <jml-end id="6" />
          // reference info
          if (options.produceReferenceInfo && unit.scope != null)
               unit.scope.storeDependencyInfo();

          // finalize problems (suppressWarnings)
          unit.finalizeProblems();

          // refresh the total number of units known at this stage
          unit.compilationResult.totalUnitsKnown = totalUnits;

          this.lookupEnvironment.unitBeingCompleted = null;
     }
```

APPENDIX A-2

```
     /**
      * Process a new compilation unit for second time processing for RAC
      */

     public CompilationUnitDeclaration Preprocessing(CompilationUnitDeclaration
currentUnit){

          CompilationUnitDeclaration newUnit = null;
          newUnit = this.parser.addWrapperMethods(currentUnit);

          lookupEnvironment.buildTypeBindings(newUnit, null);
          lookupEnvironment.completeTypeBindings();

          // Update in the lookupEnvironment that the unit being completed is the new
Unit
          this.lookupEnvironment.unitBeingCompleted = newUnit;

          this.parser.getMethodBodies(newUnit);

          // fault in fields & methods
          if (newUnit.scope != null)
               newUnit.scope.faultInTypes();

          // verify inherited methods
          if (newUnit.scope != null)
               newUnit.scope.verifyMethods(lookupEnvironment.methodVerifier());

          // type checking
          newUnit.resolve();

          // <jml-start id="6" />
          if (options.jmlEnabled) {
               jmlSourceLookup.mergeWithSourceAndSpec(newUnit);
          }
          // <jml-end id="6" />

          // flow analysis
          newUnit.analyseCode();

          // replace the new Unit from the current unit
          this.unitsToProcess[0] = newUnit;

          return newUnit;
     }

/*
 * Adds new Wrapper methods for implementation of RAC
 */
public CompilationUnitDeclaration addWrapperMethods(CompilationUnitDeclaration 
sourceUnit){
     CompilationUnitDeclaration unit;
     CompilationResult compilationResult = sourceUnit.compilationResult;
     try {
          /* automaton initialization */
          initialize(true);
          goForCompilationUnit();

          // unit creation
          this.referenceContext =
               this.compilationUnit =
                    // <jml-start id="nnts" />
                    // TODO: this might go away when default nullities are moved to type 
declarations
                    new JmlCompilationUnitDeclaration(
                    // <jml-end id="nnts" />
               this.problemReporter,
               compilationResult,
               0);
/* scanners initialization */
char[] contents;
try {
     contents = compilationResult.compilationUnit.getContents();

     // convert to string format for easy manipulation
     String sourcecode = CharOperation.charToString(contents);
     int cut = sourcecode.lastIndexOf("}");
     String substring = sourcecode.substring(0, cut);
     String newString =  "public void checkPre$m(int x){\n" +
                         "if(!(x>0)){\n" +
                         "\t throw new Error(\"Precondition Failure\");}" +
                         "}\n" +
                         "public void checkPost$m(int x){\n" +
                         "if(!(x>5)){\n" +
                         "\t throw new Error(\"Postcondition Failure\");}" +
                         "}\n" +
                         "public int $m(int x){\n" +
                         "\t checkPre$m(x);\n" +
                         "\t int retValue = m(x);\n" +
                         "\t checkPost$m(x);\n" +
                         "\t return retValue;}\n" +
                         "}"; // for the main class
     substring = substring + newString;

     //Replacing some characters
     substring = substring.replaceAll("public int m", "private int m");
     substring = substring.replaceAll("jmlObject.m", "jmlObject.\\$m");
     System.out.println(substring);
     contents = substring.toCharArray();
} catch(AbortCompilationUnit abortException) {
```

```
                this.problemReporter().cannotReadSource(this.compilationUnit,
abortException, this.options.verbose);
                contents = CharOperation.NO_CHAR; // pretend empty from thereon
            }
            this.scanner.setSource(contents);
            /* run automaton */
            parse();
        } finally {
            unit = this.compilationUnit;
            this.compilationUnit = null; // reset parser
            // tag unit has having read bodies
            if (!this.diet) unit.bits |= ASTNode.HasAllMethodBodies;
            }
        return unit;
}
```

APPENDIX A-3

SPECIFICATIONS	DETAILED DESCRIPTION	JML STATEMENTS	JML2
Heavyweight & Lightweight Specifications		behavior	Y
		normal behavior	Y
		exceptional behaviour	Y
Privacy of Method Specifications		public	Y
		protected	Y
		package-visible	Y
		private	Y
Privacy of Type Assertions		public	Y
		protected	Y
		package-visible	Y
		private	Y
Preconditions		requires	Y
Normal Post condition		ensures	Y
Exceptional Post condition		signals	Y
Frame Conditions		assignable	Y
Redundancy		_redundantly	Y
Syntactic Sugars	Specification Case	also	Y
	Nested Specification		Y
	Desugaring Specification		Y
Undefinedness Problem	Demonic		Y
	Angelic		Y
Quantified Expression	Universal Quantifiers	for all	Y
	Existential Quantifiers	exists	Y
	Generalized Quantifiers	sum, product, min, max	Y
	Numeric Quantifiers	num of	Y
	Set Comprehension	new T	Y
Inline Assertions	Assertions, Assumptions	assert, assume, hence_by	Y
	Unreachable Statements	unreachable	Y
	Set Statements	sets	Y
	Loop Invariants	maintaining	Y
	Loop Variants	decreasing	Y
Constructors	Implicit		Y
	Explicit		N
Finalizers			Y
Helper Methods			Y
Type Invariants	Static		L
	Instance		L
Type Constraints	Static		L
	Instance		L
	Old Expression		L
	Nested Method Calls		L
	Universal Constraint		L
	Method Specific Constraints		L
Specifications for Interfaces			Y
Inheritance of Specifications	Strong Behavioral Subtyping		Y
	Weak Behavioral Subtyping		Y
	Multiple Inheritance		Y
Inheritance of Instance Invariants			Y
Inheritance of Instance Constraints			Y
Inheritance of Interface Specification	Propagating Assertion calls to Super Interfaces		Y
Model Fields	Inheritance		Y
	Interface Model Fields		Y
	Interface Model Fields Inheritance		Y
Ghost Field	Inheritance		Y
	Interface Ghost Fields		Y
	Interface Ghost Fields Inheritance		Y
Model Methods			Y
Refinement		refine	N
Model Program			N
Non Functional Properties	Time and Space reqs. in concurrent programs		N
		duration	N
		working space	N
Concurrency Aspects of Programs	Synchronization	when	N
Subclassing		accessible	N
		callable	N
Example Specification			N
Termination			N
Initializers			N
Model Classes			N
Model Interface			N