RUNTIME ASSERTION CHECKING FOR JML ON THE ECLIPSE PLATFORM

USING AST MERGING

AMRITAM SARCAR

Department of Computer Science

APPROVED:

_____
Yoonsik Cheon, Chair, Ph.D.

_____
Nigel Ward, Ph.D.

_____
Bill Tseng, Ph.D.

_____
Patricia D. Witherspoon, Ph.D.
Dean of the Graduate School

to my

FAMILY

with love

RUNTIME ASSERTION CHECKING FOR JML ON THE ECLIPSE PLATFORM

USING AST MERGING

by

AMRITAM SARCAR, B.Tech.

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2009

Acknowledgements

I thank all those who helped me with various aspects of my research and the writing of this thesis. Dr. Yoonsik Cheon, my major professor and mentor, guided my graduate study at The University of Texas at El Paso. He stimulated my research interest in programming language and formal interface specification language. He spent numerous hours with me discussing many ideas and technical details that eventually led to this thesis and helped me with my writing. I also thank the other members of my thesis committee, Dr. Nigel Ward and Dr. Bill Tseng for their efforts and contributions to this work. I thank the JML developers and users for helping me develop the JML compiler on the Eclipse platform, especially Dr. Patrice Chalin, Dr. Robby, Dr. Zimmerman, Ghaith Haddad, Perry James, Jooyong Lee, and Dr. Garry Leavens of the JML group and Olivier Thomann of the Eclipse JDT team. I thank all the participants of Spring 2008 JML Winter School at UCF, Florida, for giving me the opportunity to get acquainted with the workings of JML2 and the Eclipse internal compiler. I thank the present and past members of our Software Specification and Verification Research Laboratory(SSVRL): Carmen Avila, Cessar Yeep, Luis De Haro, Fernando Cervantes, Begona Beorlieguie, Yong Wang, and Carlos Medrano. They contributed to my research in one way or another and made my graduate study at UTEP a pleasant experience. I also thank my friends, in particular, Shubhra, Avranil, Somdev'da, Jaime, and Cauhtemoc for their encouragements; and my roommates Bivas and Debarko for bearing me for almost two years! Finally I thank my family, in particular, my parents, for their love and support; my brother, for his encouragement and support; Manali, for always being there for me with love and patience; and finally "Ma", who has always been my constant support throughout my life.

Abstract

The Java Modeling Language (JML) is a formal behavioral interface specification language for Java. It is used for detailed design documentation of Java program modules such as classes and interfaces. JML has been used extensively by many researchers across various projects and has a large and varied spectrum of tool support. It extends from runtime assertion checking (RAC) to theorem proving.

Amongst these tools, RAC and ESC/Java has been used as a common tool for many research projects. RAC for JML is a tool that checks at runtime for possible violations of any specifications. However, lately there has been a problem for tool support. The problem lies in their ability to keep up with new features being introduced by Java. The inability to support Java 5 features such as generics has been reducing the user base, feedback and the impact of JML usage. Also, the JML2 compiler (jmlc) has a very slow compilation speed. On average, it is about nine times slower than a Java compiler such as javac. It is well understood that jmlc does more work than a Java compiler, hence it would be slower. The jmlc tool uses a double-round strategy for generating RAC code. The performance can be improved by optimizing compilation passes, in particular, by abandoning the double-round compilation strategy.

In this thesis I propose a technique for optimizing compilation speed using a technique known as *AST merging* with potential performance gain than its predecessor. The jmlc tool parses the source files twice. In the first pass, it parses the source file whereas in the second the source file is parsed again along with the generated RAC code. This affects the performance of the JML compiler, as parsing is one of the most costly tasks in compilation. In this thesis, I show how I solved this problem. Also, discussed in this thesis is how the new JML compiler (jml4c) was built on the Eclipse platform. To reduce maintainability issues with regards to code base for current and future versions of Java, Eclipse was chosen as the base compiler. The code base to support new Java language constructs can now be

then implicitly maintained by the Eclipse team, which is outside the JML community.

Most of Level 0 and Level 1 features of JML have been implemented. These are the features that are most commonly used in JML specifications. Almost 3500 JUnit test cases were newly created and run. Test cases and sample files from jmlc were incorporated to check completeness of the implementation. A subset of the DaCapo benchmark was used to test the correctness and performance of the new compiler. Almost 1.5 million lines of code was compiled across 350 packages generating about 5000 class files which shows that the compiler built can be used for practical and industrial purposes.

I observed that my proposed technique or the AST merging technique on an average is about about 1.6 times faster than the double-round strategy of jmlc;overall, jml4c was three times faster than jmlc. I also observed that this speedup increases with increase in lines of source code. As any industrial code or a sample code for which JML specification can be meaningfully used ranges in thousands of lines of code, our proposed technique will benefit from this. The implementation details showed the feasibility of the AST merging technique on the Eclipse platform which included front end support (lexing, parsing, type-checking) and back-end support (RAC generation and code generation). Several new features were also added that was either absent or partially implemented in jmlc. Some of them includes adding support for runtime specification inside inner classes, annotation for the enhanced-for loop, support for labeled statements, enhanced error messages and others.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, I first present an introduction to and the motivation for this thesis, which is followed by background material on the Java Modeling Language (JML), runtime assertion checking, Eclipse platform, incremental compilation, and abstract syntax trees. Then, I briefly summarize the problems that are addressed in this thesis. I also give an overview of my solutions to these problems. Finally, I summarize my contributions.

1.1 Background

The quality of software designs becomes improved by writing formal interface specifications of program modules such as classes and interfaces [Mey88] [Tan94]. The Java Modeling Language (JML) is one such specification language that helps us to write specifications for Java programming modules [LCC+05] [LBR06]. JML has a syntax that is easily understood by Java programmers, and yet provides many advanced features to facilitate writing abstract, precise, and complete behavioral descriptions of Java classes and interfaces [LBR99] [LCC+05] [LBR06]. However, formal specifications are seldom used in practice; it does not give any immediate tangible benefit. The benefits of using specifications are not obvious to programmers. The JML compiler [CL02] brings immediate and tangible benefits in terms of programming activities, such as debugging and runtime constraint validation. However, building such a compiler for a large practical language, such as Java [GJSB05] is a significant effort. Writing a runtime assertion checker for Java involves building or understanding and reusing a Java compiler and then extending that compiler with features of the formal specification language used to specify what to check dynamically. One of

the most important tool that is available for JML is its compiler (jmlc) [CL02]. However there are several problems with the current JML compiler. Its compilation speed is very slow compared to a Java compiler, it does not support any Java 5 features, and it contains implementation errors, amongst others.

In this thesis, I address the problems and issues related to the slowness of the JML compiler and show how my solution is implemented on the Eclipse platform. In the following subsections, I introduce briefly the Java Modeling Language, runtime assertion checking, the Eclipse platform, and concepts of incremental compilation and abstract syntax trees. I also outline very briefly the problem that I am solving in this thesis, my objectives, my approach, and contributions.

1.1.1   The Java Modeling Language

The Java Modeling Language (JML) [LPC+06] is a formal behavioral interface specification language (BISL) used to specify the behaviors of Java program modules. Unlike Java, JML assertions are not limited by Java's expressions; JML introduces a varied set of constructs including in-line assertions, quantifiers, invariants, history constraints, informal descriptions, model programs, and many more.

JML annotations are written inside multi-line comments like /*@ ... @*/ or single-line comments like //@ ... . For a Java compiler, the specifications are treated as comments and hence ignored, but in a JML compiler these comment-like specifications are translated into executable code. All features of JML are grouped into 4 levels, namely, levels 0, 1, 2 and X. Most of the features of JML are included in the first two levels.

Figure 1.1 shows a sample Java code annotated with JML specifications. The sample code would be used as a running example throughout this thesis, unless otherwise explicitly mentioned. In the code, line 6 is an example of type specification which is used to constraint the field `balance` to be always positive for all objects of class `BankAccount`. In line 5 the annotation `spec_public` is used to broaden the scope of the field (to *public*) inside specifications such that it can be referred by subclasses. The next few lines (8 – 12) shows

```
1  interface Account {
2    public long withdraw(long amt) throws TransactionException;
3  }
4  public class BankAccount implements Account {
5    private /*@ spec_public @*/ long balance; // more fields ...
6    //@ public invariant balance >= 0;
7
8    /*@ requires amt > 0 && amt <= balance;
9      @ assignable balance;
10     @ ensures balance == \old(balance - amt) && \result == balance;
11     @ signals (TransactionException) balance == \old(balance);
12     @*/
13   public long withdraw(long amt) throws TransactionException {
14   // method body ...
15   }
16   class TransactionException extends RuntimeException{
17     // class body ...
18   }
```

Figure 1.1: Example JML specification

what a typical method specification looks like. The specifications are written for the method `withdraw` which takes one parameter `amt`. The constraints that are to be satisfied by this method can be sub-divided into three method specifications: pre-condition, normal post-condition and exceptional post-condition. A pre-condition specifies what should be satisfied prior to entering this method. A normal post-condition specifies what are to be satisfied on exiting *normally* from this method. And an exceptional post-condition specifies what exceptions can be thrown and under what condition. In this example, the pre-condition specified by `requires` clause constraints that the parameter `amt` should be positive; and it should be less than the available `balance`. The normal post-condition specified by `ensures` clause specifies that `balance` after the transaction has completed must be equal to the old

value of `balance - amt` and the return value for this method should be equal to the current balance. The old value of `balance` is the pre-state value of `balance`. The exceptional post-condition specified by `signals` clause specifies that if the method invocation terminates abruptly by throwing an exception of type `TransactionException`, then `balance` must be equal to the old value of `balance`. In addition to this, the method should also satisfy the `assignable` clause i.e., of all the fields only `balance` can be assigned inside this method. The runtime assertion checker, a.k.a. JML2 *, is used to translate the above specifications into byte-code that can be evaluated at runtime.

In this thesis, I address the problems and issues related to runtime assertion checking compiler of JML.

1.1.2 Runtime Assertion Checking

Assertions are statements that are true at certain points of time in the program code [Hoa69]. These assertions in the program code are very useful for debugging purposes as well as proving program correctness [WK97]. They are also used for improving software testability [YB94]. RAC, for runtime assertion checking, is used to check at *runtime* whether assertions specified in the program holds or not. The JML compiler is used to generate byte-code for Java program modules that check at runtime whether any assertions has been violated or not.

Figure 1.2 shows a typical example of error reporting by the JML compiler to check for constraint validation. In this execution, the method `withdraw` is invoked with `amt = -100`. This violates pre-condition of the method specification. Hence `JMLInternalPreconditionError` is thrown with value as `amt: -100`.

---

*Due to historical reasons the current JML compiler is popularly known as JML2.

```
Exception in thread "main"
org.jmlspecs.jml4.rac.runtime.JMLInternalPreconditionError:
By method BankAccount.withdraw
Regarding specifications at
File "BankAccount.java", line 7, character 16
With values
amt: -100
    at BankAccount.withdraw(BankAccount.java:7)
    at BankAccount.internal$main(BankAccount.java:615)
    at BankAccount.main(BankAccount.java:9)
```

Figure 1.2: Output of JML under violation of a specification

1.1.3 Techniques to Validate Assertions

Constraint validation is one of the most important ways for a system to ensure integrity. Constraints are primarily stated using pre- and post-conditions. There are several ways to implement constraints that can either be validated statically or at runtime. Some of the approaches are handcrafted constraints [FGOG07], code instrumentation [Pay03] [Kra98b], compiler approach [LBR99], explicit constraint classes [FOG06], and interceptor mechanisms [WM05]. Another very efficient approach for generating runtime code is through incremental weaving [PK07]. However, one of the most popular variants is code instrumentation where it injects automatically generated code into the original code. There can be two variations to this, in-place code instrumentation, where the assertion checking code is placed within the original code and wrapper-based approach [TE03], where separate methods are generated for assertion checking.

1.1.4 The Eclipse Platform

Eclipse [Ecl] is a plug-in based application development platform for building rich client applications. 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. 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 1.3 shows the overview of the Eclipse architecture. The Eclipse Software Development Kit (SDK) is a combination of the Eclipse Platform, Java Development Tools (JDT), and the Plug-in Development Environment (PDE). As shown in the figure, the Eclipse Platform contains the functionality required to build an IDE. However, the Eclipse Platform is itself a subset of these components.
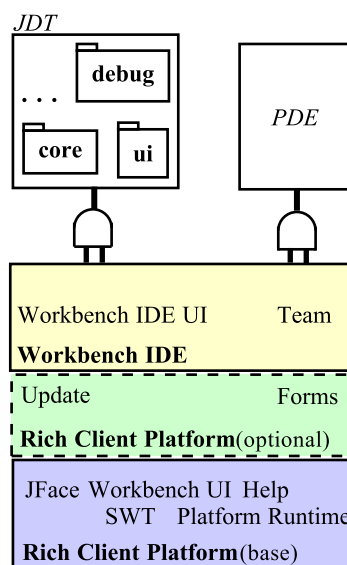
Figure 1.3: Eclipse plug-in architecture

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` i.e., all subpackages of `core`, can be considered part of the public API.

6

1.1.5 Incremental Compilation

Incremental compilation involves recompiling only that section of code that has been changed since the last compilation [Rei84]. For incremental compilation, the unit of incrementality is a very important concept. The unit of incrementality denotes the level at which re-compilation is done. Figure 1.4 illustrates the hierarchy of unit of incrementality in general. At the bottom is the compilation unit or the file that contains the source code. The compilation unit contains one or more types which may be class or interface. Each of these types are further subdivided into method level, followed by statement and expression levels. Various commercial compilers have different unit of incrementality i.e., a change at the statement level may trigger the compiler to compile only the changed statement, whereas in other cases it may trigger to compile the method in which the statement has been changed or in other cases the entire type is compiled again. In JML, most changes to the original source code (using the wrapper-method approach discussed in Section 3.1) happens at the method level, followed by statement and type levels.

Expression Level

Statement Level

Method Level

Type Level [class or interface]

Compilation Unit Level
[or, File]
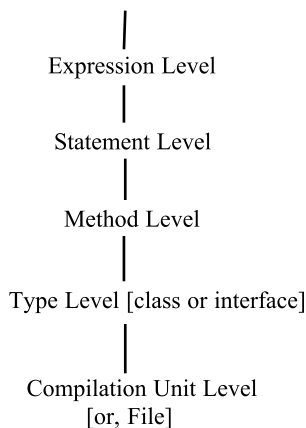
Figure 1.4: The hierarchy of unit of incrementality

1.1.6 Abstract Syntax Trees (AST)

An abstract syntax tree (AST) is a tree representation of the syntactic structure of a source code written in a certain programming language. Compilers use AST to represent programs under compilation. Each node of the tree denotes a construct occurring in the source code. Figure 1.5 represents a general form of an AST followed by a concrete example where the AST represents the source code listed in Figure 1.1. The left side of the figure shows the Java model as an AST where each node represents an element in the Java model. In Eclipse, a compilation unit represents a Java source file. Under each compilation unit, there are package-statements, import-statements and type declarations. The Java source file is entirely represented as a tree of AST nodes. Every node is specialized for an element of the Java programming language e.g., there are nodes for method declarations, variable declarations, assignments and so on. The bottom half of the figure shows a partial abstract syntax tree form of the source code enlisted in Figure 1.1.

An AST is just a tree-form representation of source code. Every element of the source code is mapped to a node or a subtree. However in Eclipse, every AST node is associated with an **id**, namely *ASTBits*. This id is a 32-bit integer value where each of the bits provide more information of the AST node including type-checked information, information regarding return values, and the type of node. It acts as a blue-print for the information contained in the AST node.

Since all the operations in Eclipse is done through AST nodes, it is interesting to know how these nodes are visited. Eclipse uses the visitor pattern [GHJV95] to traverse these nodes. It provides two operations to be performed on every node of an AST.

1.2 The Problem

The current JML compiler has several problems including the slow speed of the JML compiler compared to a modern Java compiler, lack of support for Java 5 features, lack of integration with an IDE, and unsupported features of JML. However the main problem

```
COMPILATION UNIT: BankAccount.java
   PACKAGE: null
   IMPORTS: null
        TYPES(3)
    Type Declaration
      > type name: Bank Account
      SUPERCLASS_TYPE: null
      SUPER_INTERFACE_TYPES (1)
       > type name: Account
      FIELD_DECLARATIONS (1)
      METHOD DECLARATIONS (1)
       MethodDeclaration
        > method name: BankAccount.withdraw(long)
        THROWN_EXCEPTIONS (1)
        STATEMENTS (2)
           .
           .
           .
```
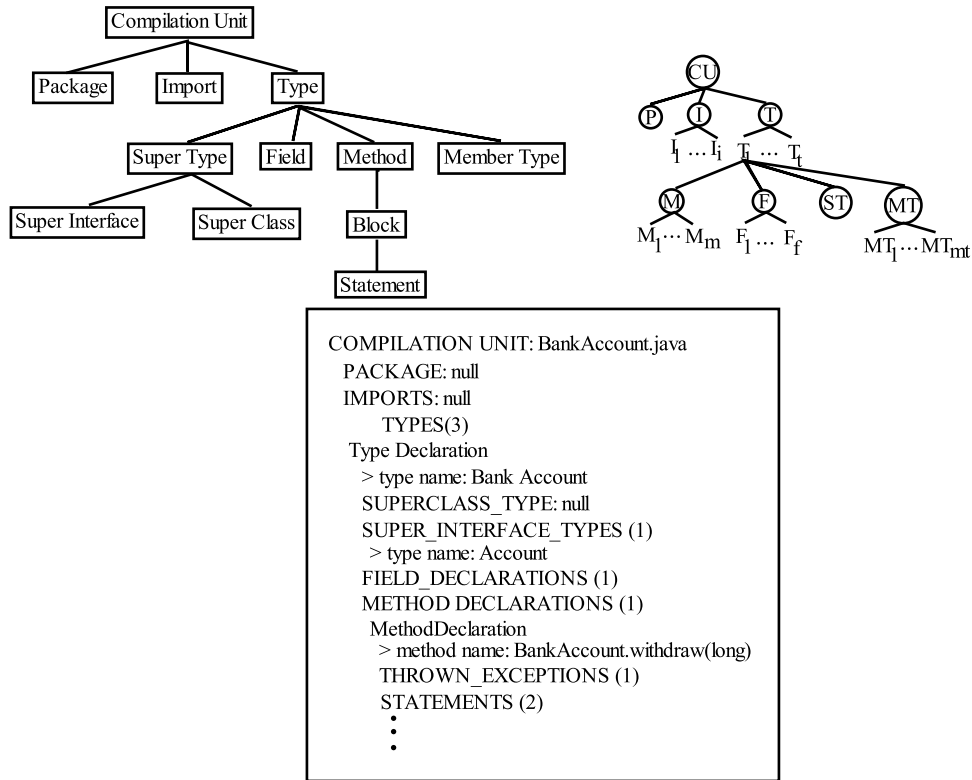
Figure 1.5: Abstract Syntax Tree Example

that I focus on this thesis is to develop a JML compiler that is faster than the previous JML compiler. I develop the JML compiler on the Eclipse platform for IDE integration and to support Java 5 features.

1.3 Objectives

My ultimate research objective is to show that my general approach towards faster generation of runtime assertion checking code is a feasible solution. I believe that a good way to do this is to show that the approach can be implemented on the Eclipse platform. My approach for achieving this is to develop an effective, extensible, and easily maintainable infrastructure. The techniques that I envision give an immediate and tangible results showing the performance gain over the existing JML compiler. The following summarizes my

9

specific research goals.

1. To develop a general AST merging approach which can be tailored for any formal specification language.

2. To develop a runtime assertion checker for JML on the Eclipse platform. This includes techniques to integrate our implementation with that of Eclipse.

3. To develop a framework such that there is a compilation speed up compared to the current JML compiler.

4. The implementation on the Eclipse platform should have minimal extension points so that it is easier to maintain and extend the framework.

In summary, the main goal of this thesis is to create a technique to generate automated runtime assertion checking code faster than the previous approach. The intention is to make the technique as general as possible.

In this thesis, I address most of JML's Level 0 and Level 1 features [LPC+06], but some of the advanced JML features such as model programs, refine statements, and others in Levels 2 and X features are left as future research topics.

1.4   Approach

In this section, I summarize my approach to the problems and challenges that were identified in the previous sections.

1. I introduce the notion of "AST merging" to merge specification checking code and original code such that the byte-code generated is used for checking runtime violation of any assertion. This approach is faster in compilation time than the double-round strategy of the compilation method.

2. I tailor or refine my general approach for JML.

3. I develop AST merging framework for JML on the Eclipse platform. I use it to integrate the new JML compiler to the IDE.

4. I refine the translation rules of jmlc (JML2) to support Java 5 features.

5. I test my framework and implementation using Junit test cases. Almost 40K test cases were tested including all the 35K test cases of the Eclipse compiler, test cases from jmlc, and newly written test cases to test the new framework. To test the effectiveness of the new approach, the approach was tested on the DaCapo benchmark [BGH$^+$06a][BGH$^+$06b].

1.5   Contributions

One of the most important contributions of this thesis is that it demonstrates and achieves a performance speed-up compared to the current JML compiler.

The second contribution is that it opens a new possibility in runtime assertion checking by successfully supporting AST merging technique.

The third contribution is that this thesis resolves many unsupported features of the current JML compiler, and resolves several existing known and unknown bugs in the JML compiler. Moreover bugs were newly discovered in the existing JML compiler.

The fourth contribution is that it supports Java 5 features which would broaden the scope of JML users (since the current compiler does not support Java 5 features and is reducing user code base).

Finally, it provides to the Java or JML community a runtime assertion checker, which is integrated with an IDE.

1.6   Outline

The rest of this thesis is structured as follows.

In Chapter 2, I give an overview of the current JML compiler, explaining its important concepts and underlying architecture. I focus on the problems of the current JML compiler, most importantly the inability to support Java 5 features and slow compilation speed.

In Chapter 3, I explain my proposed approach. I use AST merging technique to merge original source code with the runtime code for proper validation at runtime. I also show how this general approach can be tailored for JML to be implemented on the Eclipse framework.

In Chapter 4, I outline my evaluation strategy and demonstrate the practicality and effectiveness of my approach by applying it to specification-based representative test cases. The goal is to show that my approach is indeed faster than the double-round approach implemented in the current JML compiler. All the existing test cases of the Eclipse compiler were also tested to show that the new compiler does not break existing code and that all Java features are supported. Test cases from the DaCapo benchmark were also tested to show that the JML4c is able to compile *real applications*.

In Chapter 5, I conclude this thesis with a summary of my findings, followed by an outline of future research directions.

Chapter 2

The Current JML Compiler and Its Problems

In this chapter I give an overview of the current JML compiler, its underlying architecture, and the associated problems that are to be addressed in this thesis. I first show a top level view of the JML compiler and then explain informally the main architectural features of the compiler that are interesting from the perspective of runtime assertion checking. I also point out the problems of certain translation rules, as implemented in the current JML compiler. Also, mentioned are the problems of engineering these translation rules into Java programs by introducing new techniques and approaches. For complete description of JML, one should refer to JML documents such as the reference manual and design documents [LPC$^+$06] [LBR99] [LCC$^+$05] [LBR06] [CL02].

The following section discusses the compilation-based approach, the architecture of the current JML compiler, and the reasons behind performance degradation of JML2.

2.1   JML Compiler

A compilation-based approach was used for JML tool support including the JML compiler, as it is an intuitive and easy-to-use approach (see Figure 2.1). JML annotates its specification code inside special forms of comments, like (//@ ...). This has an advantage that Java or JML source files can be compiled with a Java compiler like `javac`. The JML compiler compiles Java source programs by translating JML annotations, if any, into runtime assertion checking code. It produces as output Java byte-code (`.class`) files, that

can be used in the same way as the output of Java compilers. The byte-code files may run on any Java Virtual Machines (JVMs) except that they may refer to JML-specific runtime classes. In summary, JML compiler is essentially a Java compiler with additional capability of translating JML specifications into automatic runtime checks.
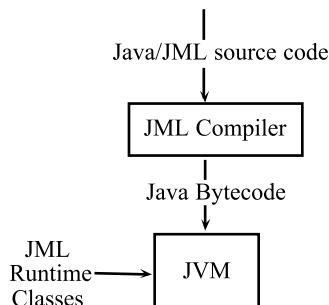
Figure 2.1: Compilation-based approach for JML Compiler

2.2 Double-Round Approach

The current JML compiler (*jmlc*) uses the double-round approach [CL02] to generate runtime assertion code. It uses the compilation-based strategy by using an underlying Java compiler to reuse already existing code of a Java compiler. The key idea behind the *jmlc* architecture is to introduce a new compilation pass that generates assertion code and then to rewire the whole compilation pass to generate single byte-code for the original and assertion code. Figure 2.2 shows the architecture of the current JML compiler, `jmlc`. The common code base for `jmlc` is an open-source Java compiler.

A new compilation pass called *RAC code generation* after the JML type-checking pass was added to implement so called the double-round approach. In this pass, runtime assertion checking code is generated from the type-checked abstract syntax tree. In this pass, the abstract syntax tree may be mutated to add special nodes for assertion code generation. If these added nodes are in the type-checked form, then compilation may proceed directly to the Java's code generation pass; this would be ideal in terms of the compilation speed.

14

However, the complexity of runtime assertion checking code makes it difficult to automate this process. To somewhat simulate this behavior, a new pass called the *RAC code printing* was added that writes the new abstract syntax tree to a temporary file, which ends the first pass compilation. In the second pass, the temporary file is compiled into byte-code by following the Java compilation passes.
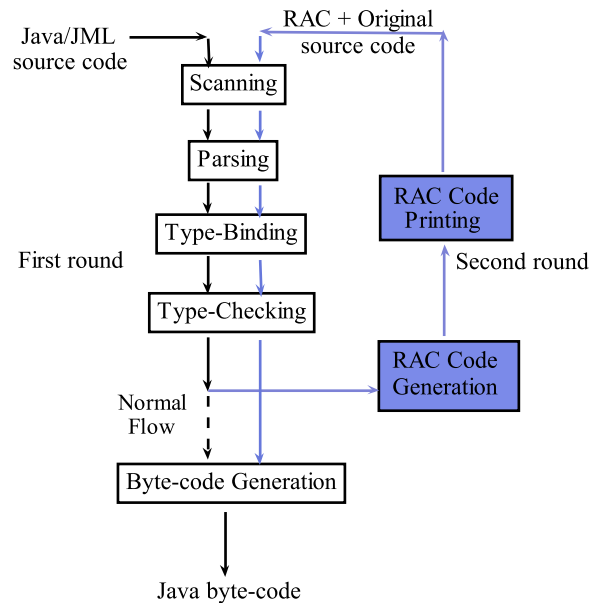
Figure 2.2: Double-round architecture for the current JML compiler (jmlc)

This architecture is called *double-round* because the original source code goes twice around the compilation path.

2.3 Problems

There are several problems associated with the current JML compiler. The problems ranges from issues related to performance of the JML compiler, deficiency in the existing translation rules, unsupported JML features, existing bugs in the JML compiler, and lack of support for Java 5 features like generics. The following subsections discusses them in detail.

2.3.1 Performance

A pressing problem of the current JML compiler is its performance. The existing JML compiler, `jmlc`, from the performance point of view is almost nine times slower than a Java compiler. The compilation time is huge compared to the compilation speed of a Java compiler like `javac` (see Figure 2.3)*. However, it is evident that since `jmlc` does more work than `javac`, it would take more time. There are several reasons for this slowness. Some of them are:

1. The `jmlc` tool does more work than `javac`. That is, using compilation-based approach [LBR99], it injects assertion-checking code into the original source-code for runtime evaluation.

2. The `jmlc` tool, being built on an open source Java compiler, MultiJava [CMLC06], results in decreasing its performance. The open source compiler is not as efficient as `javac`; it is not optimized for any kind of performance tuning unlike `javac` [Sun05].

3. Unlike Java compilers, the current JML type-checker parses the source files of all referenced types (for more information, refer to Section 2.4.) This affects the performance of the JML compiler, as parsing is one of the most costly tasks in compilation.

4. The compilation process of `jmlc` is double-round. That is, every type specification undergoes two time compilation which results in slower performance.

The programs that were test run for checking the compilation time for testing JML specifications were taken from the programs that were distributed as a part of the JML package, under the samples folder. A total of 15 sample programs were test run (see Table 2.1). They were taken from the distribution package JML2 version 5.6RC4. They are considered as standard test samples for a JML compiler.

---

*For more information refer to Appendix B.

Table 2.1: Characteristics of "sample" programs

Program	Types	Methods	Fields	Lines
AlarmClock	4	17	11	389
Purse	3	8	6	192
Digraph	**9**	64	**14**	900
DirObserver	5	13	3	189
PriorityQueue	3	13	3	101
DLList	8	66	**14**	1228
TwoWayNode	8	**70**	10	**1272**
Counter	3	6	3	103
LinearSearch	4	14	1	221
Proof	1	4	2	241
Reader	4	11	11	257
SetInterface	3	23	7	782
BoundedStack	5	33	11	573
UnboundedStack	5	21	5	223
Entry	4	22	6	299

Table 2.1 shows the characteristics of individual test samples in terms of number of types, methods, field declarations and total number of source code (number of executable lines, comments are ignored).

Figure 2.3 shows the compilation speed of jmlc and javac. From Figure 2.3, we can compute the average relative-slowness of the current JML compiler as:

$$rs_{avg} = \frac{\sum_{k=1}^{n} rs_k}{n} = \frac{\sum_{k=1}^{n} \frac{t_{jml}^k}{t_{javac}^k}}{n} \approx 8.5 \tag{2.1}$$

where $n = 15$, rs represents the relative-slowness and it is assumed that all programs are equally complex.

From the reasons cited above, obviously there's nothing that we can do about the first reason. Runtime assertion checker for JML is a tool that is used to specify program behaviors of modules. It adds more functionality to the Java compiler and thus does more work than a Java compiler. Regarding the second, there is work going on to build next generation tools on the Eclipse platform [CJK07] [KCJG08] [CJK08a], which is claimed to
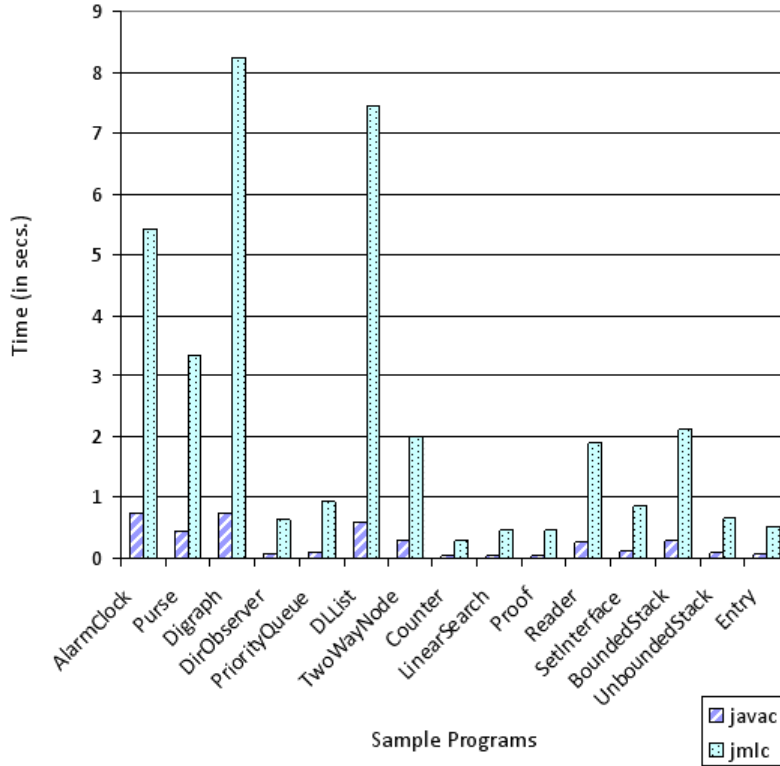
Figure 2.3: Relative-slowness of the jmlc tool compared to javac

be more efficient. The third issue is not addressed in this thesis. One solution would be to encode the signature information of JML specifications into byte-code or separate symbol files and to eliminate parsing of referenced types (see Section D, for further details.) The fourth is the main research question being addressed in this thesis.

2.3.2 Translation Rules

In loop annotation improper translation rule exists. If the loop annotation contain `continue` statements with an associated label (a Java language feature), it may result in compiler error in the second compilation pass. Another associated problem with loop annotations is that, if the loop has either a return statement or a throws clause inside the loop, the instrumented code results in compiler error.

```
       Source code                Instrumented code
                                                      Inserted by the
label:                       label:                   JML Compiler
 //@ maintaining true;        {
                               <check condition>
 for(...;...;...) {
 ...                           for(...;...;...) {
 if (<cond>)                    ...
    continue label;             if (<cond>)
 ...                              continue label;
}                                ...
                                 <check condition>
                                 }
                                }
```
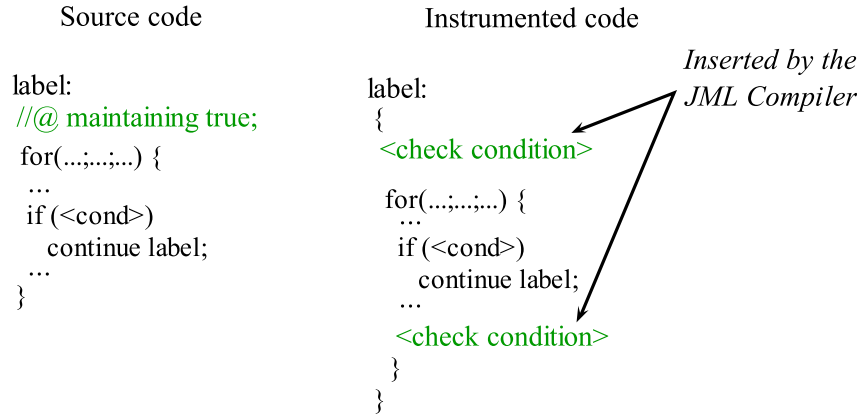
Figure 2.4: Problem in loop annotation: a synthetic example

Figure 2.4 explains the problem in detail. The source code contains a `for loop` which is annotated by a `maintaining` clause. The loop body contains a `continue` statement with an associated label. Even though there is JML annotation between the labeled statement and the start of the loop, a java compiler treats this statement as a comment line (since it starts with //) so there is no problem in the first pass. However after code generation by the JML compiler, assertion checking code is instrumented before the loop and at the end of loop. In the second pass, this results in a compiler error since the Java language feature does not allow statements in between labeled statement and the start of a loop where the label is referenced from a `continue` statement inside the loop.

2.3.3 Implementation Errors

There are several implementation errors in the current implementation of the JML compiler. Some of them are discussed below (for a complete list see Section 4.3.4).

Type invariants can refer to static or non-static fields. However proper contextual information is required for translating them. That is, amongst other things, it is important to know whether the field that is referred in the invariant clause is a static or non-static field. As per Java language specification (version 2.0), `this` operator cannot be used inside static methods. However in the current JML implementation, if the field referred in the

static invariant clause is a non-static field then `this` is used to reference the field, which results in a compilation error inside static methods.

Source code   Instrumented code

Inserted by the JML Compiler

```
public int field = 0;
//@ public static invariant field > 0;
...
```

```
...
public static void checkInv$static() {
    if (this.field > 0) ...

...
```

Figure 2.5: Problem in type invariants: a synthetic example

Figure 2.5 explains the problem in detail. The original code contains a field declaration that is non-static and has an associated static invariant clause. This is converted by the JML compiler into a static method where the predicate (`field > 0`) is checked. However since `field` is a non-static field declaration it is converted as `this.field` which results in compiler error.

2.3.4 Unsupported Features

Specifications in nested classes are not supported by the current JML compiler. That means, specifications written in nested classes are not checked at runtime even if the classes are compiled using the JML compiler and has JML annotations. There is no JML tool yet that supports the several new features of Java 5, most important is the introduction of generics. Since the MultiJava compiler is not being maintained, the JML project has been struggling to support the features of Java 5, especially generics. This is a major problem for the JML community since most source code in recent times is written in terms of generics for which the JML compiler cannot be used. Even support for Java 5's *enhanced for loop* is not available in the current release of the Common JML Tools, also known as JML2. Also, the current instrumented code is not Java 5 compatible, that results in several warnings from the second pass of compilation which is undesirable.

2.3.5 Extensibility

The current JML compiler (jmlc) does not support robustness [CJK07]. The code base of the open source compiler on top of which JML is built does not support extensibility, hence the maintenance of JML2 becomes extremely difficult. The implementation of the JML tools exposes various private API's and manipulates the internal architecture of the base version which further makes future extensions more difficult.

2.4 A Closer Look at Performance Degradation

In this section I take a closer look at the reasons for performance degradation of the current JML compiler. Here I discuss how each factor contributes to the slowness of the current JML compiler and the reasons behind them. I conclude by showing that why double-round strategy is an important problem to solve which is the focus for my thesis.

2.4.1 The Problem of Separate Compilation

The default behavior of the javac compiler is to compile other dependent or referenced files iff:

1. Only source code (.java) is available in class-path, or

2. Time-stamp of the source code is later than byte-code (when both are present), i.e., the contents of the source code is the latest.

That means, if the source code and byte-code have the same-time stamp then the javac compiler is not required to compile the source code, it reads the corresponding byte-code. However in the case of jmlc this is not true; it looks for the source code first (even when the byte-code is present and has the same time stamp), and if present recompiles the code to gather type information. That is, the JML compiler uses separate compilation for compiling dependent files.

For detailed description and reasons for such behavior in jmlc refer to Appendix D.

Figure 2.6 shows the relative slowness of the JML compiler if there is no separate compilation for referenced files.
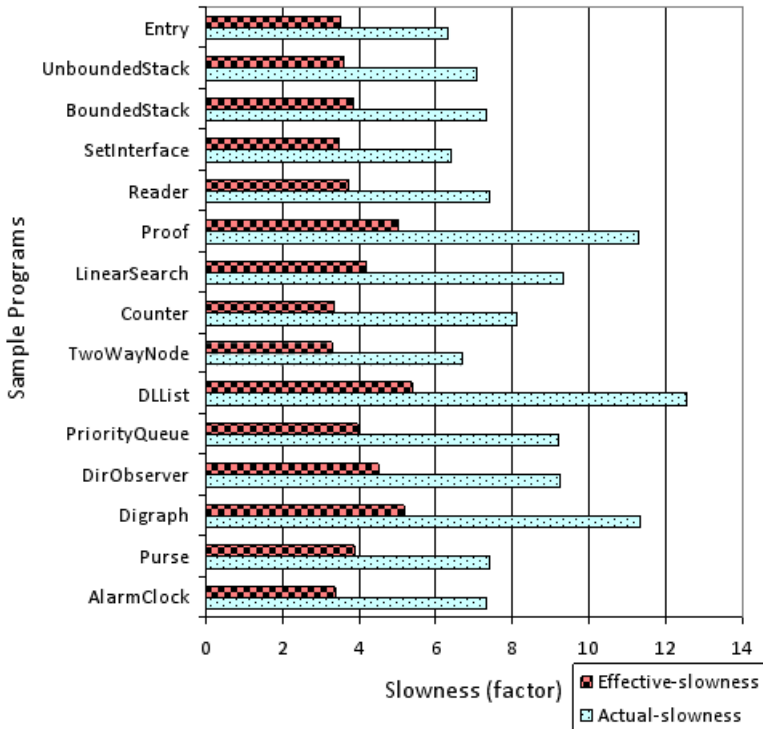
Figure 2.6: Relative-slowness of jmlc due to separate compilation

We can easily compute the slowness of the current JML compiler due to separate compilation of referenced files. It is given by:

$$rs_{avg}^{spcm} = \frac{\sum_{k=1}^{15} rs_k^{spcm}}{15} = \frac{\sum_{k=1}^{n} \frac{t_{spcm}^k}{t_{javac}^k}}{15} \approx 4.0 \qquad (2.2)$$

2.4.2 JML Compiler Built on Multi-Java Compiler

The current JML compiler has been built on Multi-Java [CMLC06], an open source compiler for Java. It would be interesting to see how much slower is Multi-Java with respect to the javac compiler. This is important because we require to know what is the *actual* relative-

22

slowness of JML w.r.t. javac. Figure 2.7(a) shows the relative slowness of Multi-Java to javac compiler.

Since JML is built upon Multi-Java we can write a very simple equation:

$$t_{JML} = t_{MJ} + t_{JML'} \tag{2.3}$$

where JML$'$ is the time taken to compile the source code due to the added components or code of JML. Now dividing equation 2.3 with $t_{javac}(> 0)$, we get:

$$\frac{t_{JML}}{t_{javac}} = \frac{t_{MJ}}{t_{javac}} + \frac{t_{JML'}}{t_{javac}} \tag{2.4}$$

From this equation we can plot a graph as shown in Figure 2.7(b) where `Actual-slowness` and `Effective-slowness` is denoted by $\frac{t_{JML}}{t_{javac}}, \frac{t_{JML'}}{t_{javac}}$ respectively.

We can easily compute the slowness of the current JML compiler incurred from Multi-Java from 2.4. It is given by:

$$rs_{avg}^{mj} \approx 1.5 \tag{2.5}$$

2.4.3   Double-round Architecture

The major bottleneck for this architecture is the double-round compilation undergone by the original source code. It is a well-known fact that in a compilation phase, most time is spent in the scanning phase since this requires interacting with a slower device like the hard-disk (see Figure 2.8). In this architecture, scanning and parsing is done twice for the original code which slows down the performance.

Figure 2.9 shows the effective slowness of the JML compiler due to the double-round strategy. It can be easily observed that the total time taken to compile in a JML compiler is still much slower than a javac compiler. This can be attributed to the fact that a JML compiler does more work than a Java compiler. In addition to this, a huge chunk of instrumented code is also added to the original code. However between Figures 2.3 and 2.9, the total time in case of *only double-round* is much faster than when separate compilation of referenced files is not present.

Similarly, we compute the slowness of the current JML compiler incurred due to double-round strategy. It is given by:

$$rs_{avg}^{dbl} \approx 2.5 \tag{2.6}$$

From equations 2.1, 2.2, 2.5 and 2.6 we observe that:

$$rs_{jml}^{total} = rs_{jml}^{rcrsv} + rs_{jml}^{mj} + rs_{jml}^{dbl} \approx 8.5 \tag{2.7}$$

2.5 Summary

In this chapter I discussed in detail the underlying architecture of the current JML compiler, jmlc. I also explained the problem of the current compiler. The problem was shown to have several reasons: separate compilation for referenced files, double-round, JML compiler does more work than a Java compiler, etc. To evaluate and gain more understanding of these factors, I conducted several experiments whose results has been discussed here. The jmlc tool was shown to use a double-round strategy for generating RAC code. This affects the performance of the JML compiler, as parsing is one of the most costly tasks in compilation.
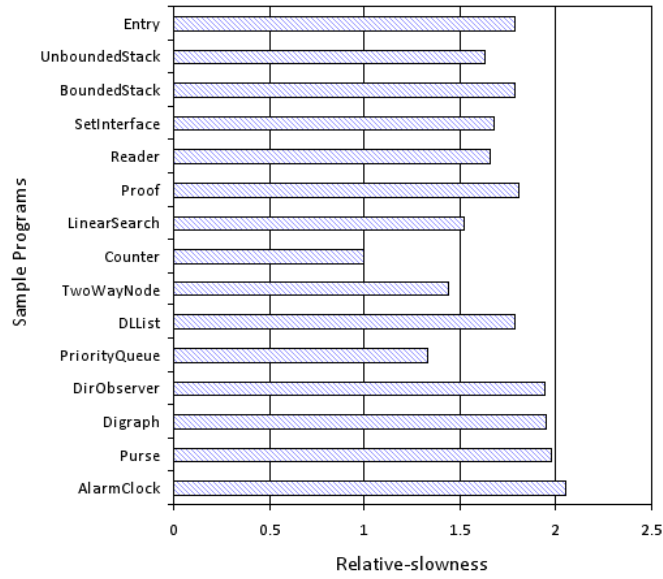
(a) Relative slowness of Multi-Java w.r.t. javac
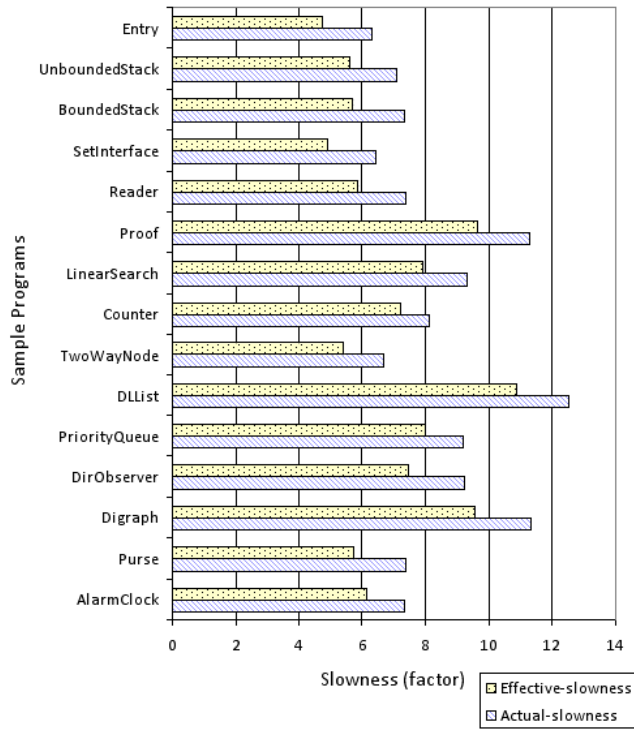
(b) Effective slowness of JML w.r.t. javac

Figure 2.7: Relative-slowness of jmlc
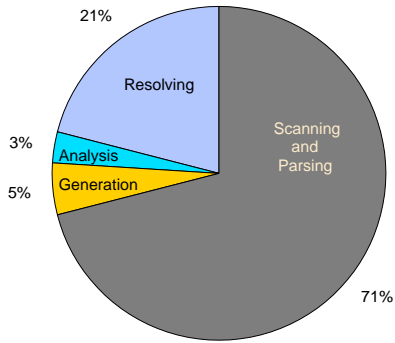
Figure 2.8: Distribution of compilation time for different phases
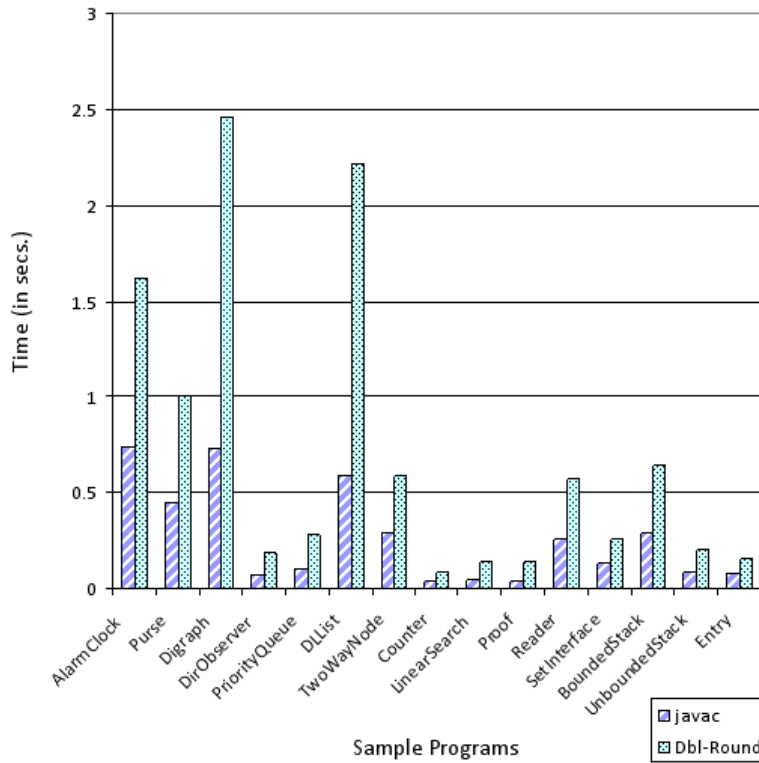
Figure 2.9: Relative-slowness of the jmlc tool due to double-round compared to javac

Chapter 3

Incremental Compilation Using AST Merging

In this chapter, I propose an incremental compilation using AST merging as a solution to the problem mentioned in the previous chapter. I explain in this chapter incremental compilation and AST merging in details, giving the outline of the general approach and showing how this approach can be tailored to JML on the Eclipse platform.

3.1   Approach

There are several approaches that can be used to translate assertions into executable code. Some of them are:

- One of the most popular approaches for translating assertions is preprocessing [BS03]. In this approach, the assertions are preprocessed which produces source code that contains both original and runtime assertion checking code.

- Another approach for translation is the compilation-based approach. This approach is used when assertions have built-in programming language features such that they can be directly translated by the native compiler.

- The third variation is the byte-code manipulation or weaving approach. This approach is limited in scope since it can be used only for languages based on virtual machines. The assertion checking code is embedded directly into the machine's byte-code [BH02].

In our approach, we use compiler based technique for generating JML specific code to check assertions at runtime. This approach is very similar to the approach outlined in [CL02]. It works on the same principle as that of the double-round architecture; that is, it consists of two compilation passes. Unlike the double-round architecture, in this technique only the JML specific code is sent to the second compilation pass. The steps involved in code generation using AST merging technique are illustrated in Figure 3.1. In the figure, steps 1–2 occur in the first compilation pass and the rest in second. In the first compilation pass, the original source code is parsed and type-checked. In this step, the assertions are also parsed and type-checked. This is shown in the figure where the input to this step is the source code and the output is a type-checked AST. For generating JML RAC code, type-checked information of the original AST (source code) is required. This is because, without knowing the type of a JML expression it is impossible to generate RAC code that is type safe [CL02]. With the type-checked information, JML RAC code (in source code format) is generated. Unlike in the double-round architecture, only JML RAC code is scanned and parsed which results in an untype-checked AST. This untype-checked AST is further type-checked and resolved as shown in step 4 of the figure. We thus now have two AST's, the original AST containing the original source code information (from the previous pass) and the second AST that contains only JML code information. A key component in this technique is the AST merging mechanism. The two AST's are then merged into one single AST containing both the original and JML code. This is shown in step 5 of the figure. On successful merging, the resulting AST is type-checked and is used for byte-code generation. This ends the second pass of compilation and also concludes the compilation path for generating byte-code to support JML specifications.

The steps involved to implement this technique can be summarized as:

1. In the first pass, parsing and type-checking the original source code, including JML annotations, is done.

2. Using this type-checked AST, JML RAC code is generated (in source code format).
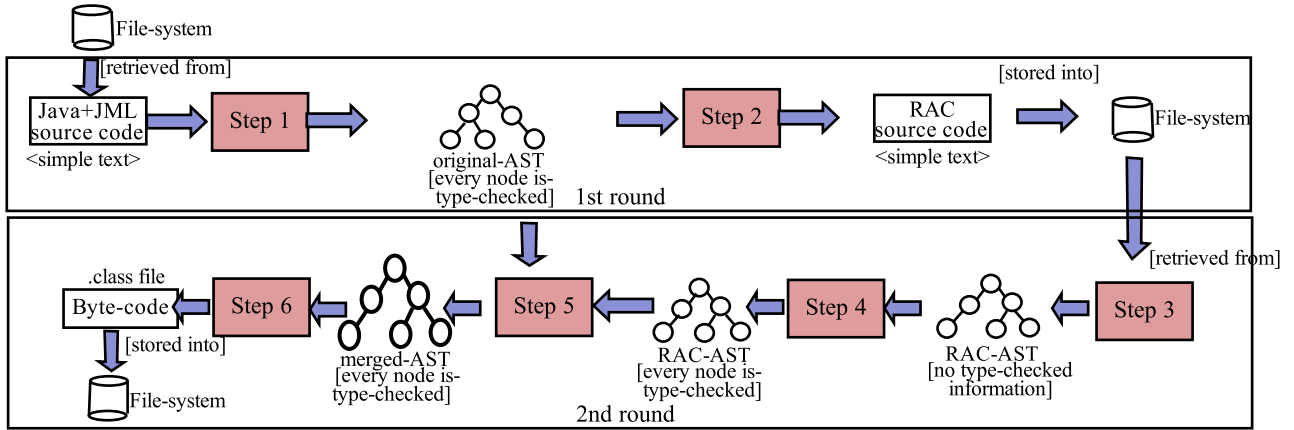
Figure 3.1: General Approach for generating RAC code

3. The JML RAC code is parsed. Parsing the JML code creates an initial AST.

4. This un-checked AST (JML AST) is type-checked and resolved.

5. The JML AST and the original AST are then merged together into a single AST.

6. The resulting merged AST is sent to code generation.

Before, I explain the AST merging technique in detail, let me explain the characteristics of the JML AST.

3.1.1 Characteristics of JML AST

The characteristics of a JML AST are discussed in this section, including compilation unit declaration, type declaration and method declaration.

A compilation unit consists of a package statement, import statements, and type declarations. In the JML translation approach, the package declaration remains the same between the ASTs. The import statements and type declarations may differ between the ASTs. The JML AST may contain either less, equal or more types than in the original AST. It may contain fewer types when the original AST contains types like `enum` that are

not implementable by the JML compiler. The JML AST can contain more type declarations if the original AST contains model types that needs to be translated to concrete types.

A type declaration consists of field declarations, super types (super classes and super interfaces), member types, and method declarations including constructor declarations. Usually the number of field and method declarations differ between the ASTs. The member type declarations * differ in cases when the enclosing type of the member type is an interface and is implementable, or the member type is a model type. In super types, the declaration of super classes remain the same, however super interfaces may change between the ASTs.

The JML compiler uses a wrapper-based approach for generating RAC code for type- and method-level specifications. In general, for every method present in the original AST, say $\mathcal{ME}_1$, the JML compiler generates four more instrumented methods, namely, $\mathcal{ME}_{pre1X}$, $\mathcal{ME}_{pst1X}$, $\mathcal{ME}_{xpst1X}$, $\mathcal{ME}_{in1X}$ where the subscripts represent pre, post, xpost, and internal methods for the original method $\mathcal{ME}_1$ in type $\mathcal{X}$ (see Figure 3.2). These methods are generated by the JML compiler to support method specifications. The merged AST contains all the five methods (four from JML AST and one from the original AST) merged in a *special manner* (which is explained further in the following subsections.) The merged AST contains other methods that are generated to support type specifications. It is also possible that for a particular type even if the original AST contains an empty type declaration, the corresponding JML AST may contain methods, fields, and even member types.

Table 3.1 enlists all the different symbols used in this algorithm with their associated meanings.

In addition, JML uses Hoare-style assertions for specifying pre- and post-condition. In addition to this, it can also specify type invariants, inline constraints, and abstract specifications using `ghost` and `model` keywords. The JML compiler uses both strategies of code instrumentation, inline and wrapper approach. Figure 3.2 shows a sample snippet of the JML generated code for the code listed in Figure 1.1. In this code, lines 1–12 are the

---

*also known as inner types

Table 3.1: Symbols and their meaning

Symbol	Meaning
OT	Original AST (AST version of the original source code)
JT	JML AST (AST version of the JML-specific code)
MT	Merged AST (AST version of the merged code)
*.name	Fully qualified name of the associated node of an AST
$*.\mathcal{T}$	Type declarations of the associated AST
$*.\mathcal{I}$	Import declarations of the associated AST
$*.\mathcal{F}$	Field declarations of the associated AST
$*.\mathcal{ST}$	Super type declarations of the associated AST
$*.\mathcal{MT}$	Member type declarations of the associated AST
$*.\mathcal{ME}$	Method declarations of the associated AST
$\mathcal{ME}$.SIG	Method signature that includes visibility modifier,
	return type, name, argument and throws clause for that method
$\mathcal{ME}$.SIGR	$\mathcal{ME}$.SIG with "*internal*" removed from the name of the method, if present.
M_{pre1X}	pre-condition checking method for M_1 in type X
M_{pst1X}	normal post-condition checking method for M_1 in type X
M_{in1X}	internal method for M_1 in type X
M_{xpst1X}	exception post-condition checking method for M_1 in type X

different wrapper methods that were generated to check method and type specifications. Every specification method is named in a predefined manner; containing three parts each delimited by '\$'. These sub parts represent the type of specification method (i.e., whether it is to check pre-condition, post-condition, etc.), the method for which the specifications is written (in our case it is the `withdraw` method), followed by the name of the type in which the method is present (class `BankAccount` in this case). Additionally, the original method body is replaced by delegation calls to different specification methods as shown in the diagram between lines 17–26. And the original method body itself is placed inside a new method (generated by the JML compiler) having the same signature as that of the `withdraw` method; the method name is prefixed by the name `internal`. This is shown in lines 13–16 of Figure 3.2.

```
1  /** Check pre-condition for method withdraw */
2  private boolean checkPre$withdraw$BankAccount(){
3    boolean pass = false;
4    // check the condition and return true or false accordingly
5    return pass;
6  }
7  /** Check post-condition for method withdraw */
8  private void checkPost$withdraw$BankAccount(){
9    // check the condition and return normally or throw exception
10 }
11 /** Check other conditions also like invariants, history
12 constraints, exceptional post conditions */
13 /** Original code goes here */
14 private int internal$withdraw$BankAccount() {
15   // Original code + assertion checking code for inline assertions
16 }
17 /** Original function from where all delegation happens */
18 public int withdraw() {
19   int result = 0;
20   // delegate calls to constraint checking methods
21   checkPre$withdraw$BankAccount();
22   try {
23     internal$withdraw$BankAccount(); checkPost$withdraw$BankAccount();
24   } catch(Exception e) { ... }
25   return result;
26 }
```

Figure 3.2: Wrapper-based instrumented code for validating assertions

3.1.2 AST Merging Algorithm

A key component of our approach is AST merging. This section explains the algorithm in detail. The entire merging mechanism can be subdivided into 3 major levels of merg-

ing: compilation unit level merging, type level merging, and method level merging. The following subsections discuss the approach for merging the ASTs at each level.

Compilation Unit Level Merging

Compilation Unit level Merging

Non-overlapping imports are appended*

*import declarations in RAC are generated from model imports
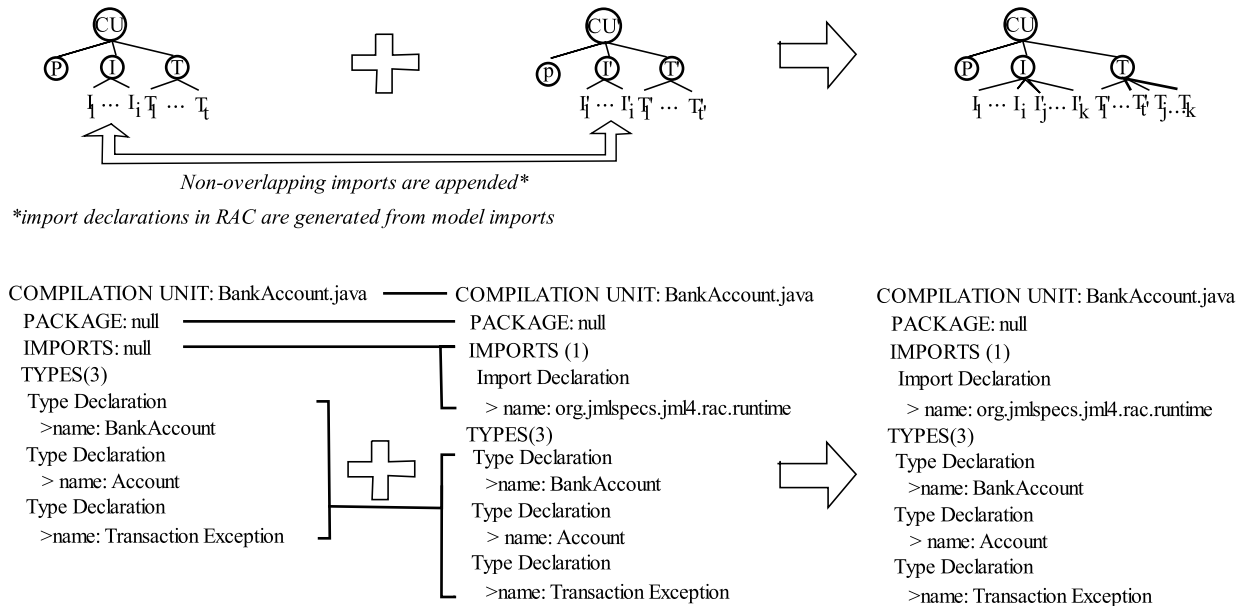
Figure 3.3: Compilation unit level merging

The compilation unit declaration is shown in Figure 3.3 where package statements, import statements, and type declarations are marked as P, I, and T, respectively. The general merging mechanism is illustrated here. Let us assume that type declarations and import declarations for the original AST ranges from $\{T_1$ to $T_t\}$ and $\{I_1$ to $I_i\}$ respectively and JML AST from $\{T_1'$ to $T_{t'}'\}$ and $\{I_1'$ to $I_{i'}'\}$. The merged AST would then contain type declarations that are in JML AST and remaining types that are declared in original AST but not in JML AST. This happens when the original AST contains types that are not implementable by the JML compiler. Conversely, for import declarations the merged AST

would contain declarations from the original AST and remaining import declarations in JML AST. This can be represented as follows:

$$OT.\mathcal{T} = \{T_1, T_2, ... ,T_t\}$$
$$JT.\mathcal{T} = \{T_1', T_2', ... ,T_{t'}'\}$$
$$MT.\mathcal{T} = T' \cup T = \{T_k \mid T_k \in T' \text{ or } T_k \in T\}.$$
$$OT.\mathcal{I} = \{I_1, I_2, ... ,I_i\}$$
$$JT.\mathcal{I} = \{I_1', I_2', ... ,I_{i'}'\}$$
$$MT.\mathcal{I} = I \cup I' = \{I_k \mid I_k \in I \text{ or } I_k \in I'\}.$$

Figure 3.3 illustrates the general algorithm in the upper-half of the diagram. The figure shows the merging mechanism using the general form of ASTs. Import statements from the original AST are first copied into the merged AST and then any other import declarations in JML AST that are not present in original AST, are appended. For merging type declarations it is reverse: JML AST type declarations are copied into the merged AST first and then any other types that are not present in JML AST but present in the original AST are appended. The bottom half of the figure shows a concrete example and how this merging is done. The original AST is the AST version of the code written in Figure 1.1 and the JML AST is the JML version for the same code. The original AST contains no package or import statements but contains three types namely `BankAccount`, `Account`, and `TransactionException`. The generated JML AST on the other hand contains no package statement, the same number of type declarations, and an additional import declaration than that of the original AST. The merged AST contains this import statement as can be seen from the figure.

Type Level Merging

Type declaration containing field declarations, super types, member type declarations and methods are shown in Figure 3.4 which are marked as F, ST, MT, and M, respectively. The

Type level Merging

The merged AST contains super types and member types from RAC

merging fields

Ti

F M ST MT

$F_1 ... F_f$ $M_1 ... M_m$ $MT_1 ... MT_{mt}$

merging methods

Ti

F' M ST MT

$F'_1 ... F'_f$ $M'_1 ... M'_{m'}$ $MT'_1 ... MT'_{mt'}$

Ti

M F ST MT

$F_1 ... F_f$ $F'_1 ... F'_f$ $M_1 ... M_m M'_1 ... M'_{m'} MT'_1 ... MT'_{mt'}$

Member types are recursively merged

COMPILATION UNIT: BankAccount.java
> type name: BankAccount
SUPERCLASS_TYPE: null
SUPER_INTERFACE_TYPES (1)
> name: Account
FIELD_DECLARATIONS (1)
METHOD DECLARATIONS (1)

COMPILATION UNIT: BankAccount.java
> type name: BankAccount
SUPERCLASS_TYPE: null
SUPER_INTERFACE_TYPES (2)
> name: Account
> name: org.jmlspecs.jml4.rac.runtime.JMLCheckable
FIELD_DECLARATIONS (5)
METHOD DECLARATIONS (29)

COMPILATION UNIT: BankAccount.java
> type name: BankAccount
SUPERCLASS_TYPE: null
SUPER_INTERFACE_TYPES (2)
> name: Account
> name: org.jmlspecs.jml4.rac.runtime.JMLCheckable
FIELD_DECLARATIONS (6)
METHOD DECLARATIONS (30)

Figure 3.4: Type-level merging

general merging mechanism is illustrated here. Let us assume that the original AST has several types $\mathcal{T} = \{T_1, ... ,T_t\}$ and each type contains field declarations $\mathcal{F} = \{F_1, ... ,F_f\}$, super types $\mathcal{ST}$, member types $\mathcal{MT} = \{M_1, ... ,M_{mt}\}^{\dagger}$ and method declarations $\mathcal{ME} = \{ME_1, ... ,ME_m\}$. In case of JML AST, they are represented as $\mathcal{T}'$ and each type contains $\mathcal{F}'$, $\mathcal{ST}'$, $\mathcal{MT}'$ and $\mathcal{ME}'$. These are shown in the upper portion of Figure 3.4. During JML code generation, several fields may be instrumented for its purpose. These fields must be merged into the final AST. Hence, the merged AST contains both original AST and JML AST's field declarations. This is shown in the figure where all the fields from the RAC is appended to the original AST. In case of super types and member types, it is possible for JML to have more super types, and member types than that of the original. Moreover, it is also possible that one of the member types is not implementable by JML like `enums` and `annotation type`. The manner in which member type declarations are merged is by appending the non-overlapping types between the original AST and JML AST (i.e., types that are declared in the original AST but not in JML AST) to JML AST's member type declarations. For merging methods between original and JML AST, special considerations

---

†Each type in $\mathcal{MT}$ is again another type declaration

are made (which is explained later in the following subsection). However, at the top level, the merged methods should contain all the methods of original and JML. The merging mechanism at the type level is represented as:

$$MT.\mathcal{F} = F \cup F' = \{F_k \mid F_k \in F \text{ or } F_k \in F'\}.$$
$$MT.\mathcal{ST} = \{ST'_1, ST'_2, \dots, ST'_{st'}\}$$
$$MT.\mathcal{MT} = MT' \cup MT = \{MT_k \mid MT_k \in MT' \text{ or } MT_k \in MT\}.$$
$$MT.\mathcal{ME} = ME \cup ME' = \{ME_k \mid ME_k \in ME \text{ or } ME_k \in ME'\}.$$

Figure 3.3 illustrates this general scheme in the upper-half of the diagram. The figure shows the merging mechanism using the general form of ASTs. Field declarations from both ASTs are copied into the merged AST, super type declarations from only JML AST are copied into the merged AST. Member type declarations of JML AST are first copied and then any non-overlapping types from original AST are appended to the merged AST. The bottom half of the figure shows a concrete example and how this merging is done. The original AST is the AST version of the code written in Figure 1.1 and the JML AST is the JML version for the same code. The original AST contains three types of which one is an interface. In the original AST, as shown in the figure, `BankAccount` contains no super classes, one super interface, one field declaration and one method declaration and no member types. However, in JML AST, it contains two super interfaces, five field declarations, and 29 method declarations. Using the merging mechanism discussed above, the merged AST contains no super classes, two super interfaces, six field declarations (one from original AST and the rest from JML AST), and 30 method declarations. For type `Account` which is an interface, even though the original AST contains no member type declarations, the JML AST and hence the merged AST contains one member type declaration. This member type is required to be generated by JML to make the interface `Account` implementable since an interface in Java is not implementable. This is required because an interface may contain specifications that are to be satisfied by the runtime

36

checker, in such a case the JML compiler adopts a strategy by creating a concrete class that extends `JMLSurrogate` (a special class in the jml runtime package) and implements the corresponding interface and all specifications are checked in this inner class.

Method Level Merging
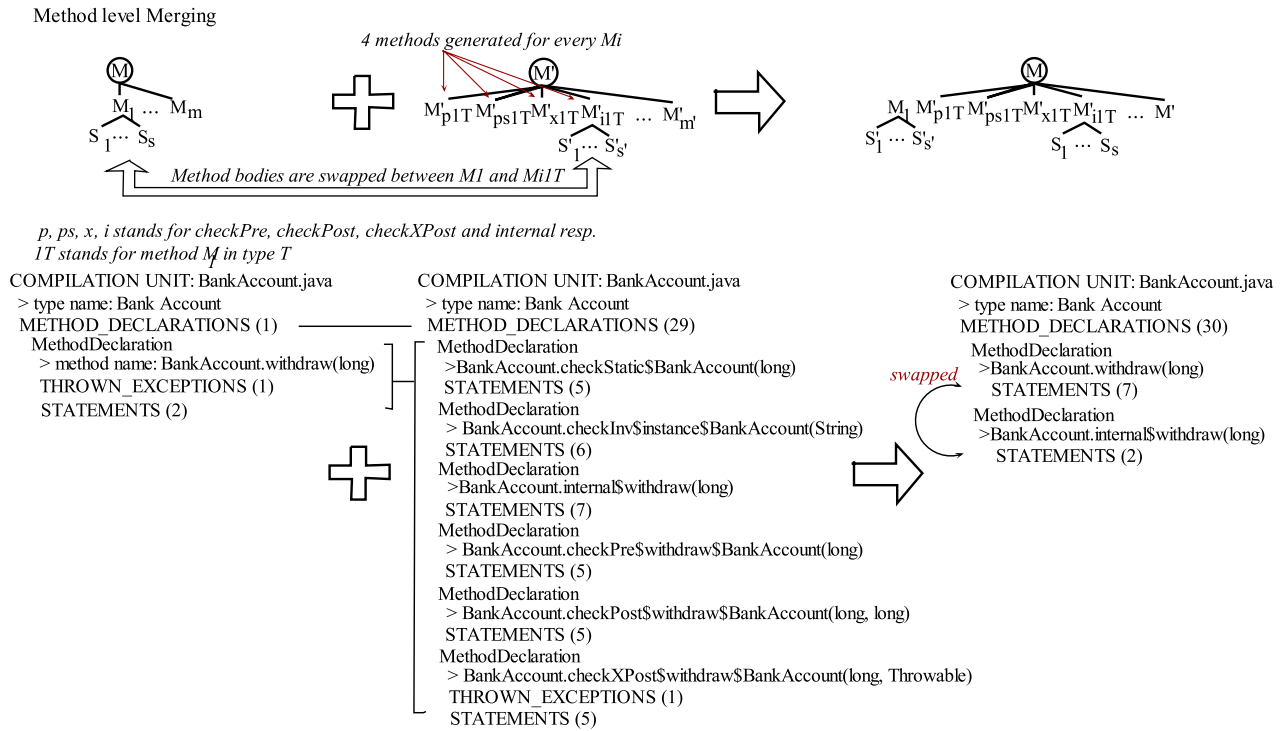
Figure 3.5: Method level merging

Figure 3.5 shows the merging mechanism, where all the five methods are appended to the merged AST and the method bodies between the internal method generated by JML $\mathcal{ME}_{in1X}$ and the original method $\mathcal{ME}_1$ are swapped (see Section 3.2.2). Using pattern matching, we identify those methods in JML which have the same method name as in original AST prefixed by the pattern `internal` (see Figure 1.1). On correct identification, we swap the method bodies between the JML AST with that of the original AST. This is

in conformance with the wrapper-based approach where the original method is replaced by delegation calls and the original method body itself is placed inside a new method. The swapping of method bodies is required because the calls to different specification methods is instrumented inside "internal" method rather than the original method and vice-versa. they This is further explained in Algorithm 1. The merging mechanism at the method level is represented as:

$OT.\mathcal{ME} = \{ME_1, ME_2, ... ,ME_m \}$

$JT.\mathcal{ME} = \{ME'_1, ME'_2, ... ,ME'_{m'} \}$ where for each i in ME_i

there are four methods in ME' (ME'_{pre1}, ME'_{pst1}, ME'_{xpst1}, ME'_{in1})

and there are other methods in ME' to support type specifications

$MT.\mathcal{ME} = ME \cup ME' = \{ME_1, ME_2, ..., ME_m, ME'_1, ... ,ME'_{m'}\}$

where the method bodies of ME_1 and ME'_{in1} are swapped.

Figure 3.5 illustrates this general scheme in the upper-half of the diagram. The figure shows the merging mechanism using the general form of ASTs. Method declarations from the JML AST and the original AST are copied into the merged AST, where for every method ME_i in the original AST its body is swapped with that of ME'_{in1}. The bottom half of the figure shows a concrete example and how this merging is done. The original AST is the AST version of the code written in Figure 1.1 and the JML AST is the JML version for the same code. The original AST contains a single method for type `BankAccount`. In the JML AST it contains 29 method declarations where some of them has been shown in the figure. The first two methods namely `checkInv$static$` and `checkInv$instance$BankAccount` are methods for supporting type specifications. The other methods shown in the figure are to support the method specification of the `withdraw` method. The merged AST as shown in the figure contains 30 method declarations where the method bodies between `withdraw` and `internal$withdraw` has been swapped.

Algorithms 1 and 2 gives the general steps to merge two ASTs namely the original AST,

parsed and type-checked from the source code, and the JML AST generated by the JML compiler from the source code.

Algorithm 1 Merge OT with JT

Input: OT and JT

Output: MT

    ***MergeASTs*(OT, JT)**

 1: $MT.\mathbf{P} = OT.\mathbf{P}$

 2: $MT.\mathbf{I} = OT.\mathbf{I} \cup JT.\mathbf{I}$

 3: **for all** $sT \in OT.\mathbf{T}$ and $cT \in JT.\mathbf{T}$ such that $sT.\mathbf{name} = cT.\mathbf{name}$ **do**

 4:     **Merge** type-level constructs that includes fields, super types, member types to **cT**.

 5:     ***MergeMethods*(sT, cT)**

 6: **end for**

 7: $MT.\mathbf{T} = JT.\mathbf{T} \cup OT.\mathbf{T}$

Algorithm 2 Merging Methods of JT and OT

Input: OT and JT

Output: MT

    ***MergeMethods*(OT, JT)**

 1: **for all** $oM \in OT.\mathbf{ME}$ and $rM \in JT.\mathbf{ME}$ such that $oM.\mathbf{SIG} = rM.\mathbf{SIGR}$ **do**

 2:     **if** $rM.\mathbf{SIG}$ contains 'internal' **then**

 3:       Swap $oM.\mathbf{S}$ and $rM.\mathbf{S}$

 4:     **end if**

 5: **end for**

 6: $MT.\mathbf{ME} = JT.\mathbf{ME} \cup OT.\mathbf{ME}$

3.2 Application: A JML Compiler on the Eclipse Platform

In this section, we discuss the motivations behind developing a JML compiler on the Eclipse platform. We also discuss the necessity to alter the general approach to our specific need due to the constraints set by the Eclipse framework and how we make use of our proposed approach to build a JML compiler on Eclipse platform. In the following sub-section, we outline the reasons for selecting Eclipse as the base compiler onto which a JML compiler is built. The Eclipse framework presents certain logistic and architectural constraints which required us to slightly change our approach. The following sections discuss the changes in the approach, the reasons for it and the implementation details.

3.2.1 Why Build JML on the Eclipse Platform?

The JML community is targeting mainstream industrial software developers as the key end users. Since JML is essentially a superset of Java, most JML tools will require, at a minimum, the capabilities of a Java compiler front end. Some tools (e.g., the RAC) would benefit from compiler back-end support as well. One of the important challenges faced by the JML community is keeping up with the accelerated pace of the evolution of Java. As researchers of JML community, 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. 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 is an important criteria for our effort. Chalin, James, and Karabotsos describes the importance

of this problem and discusses a possible solution of building front-end support for JML on top of the Eclipse platform [CJK08b].

3.2.2 Challenges

The AST merging technique proposed in Section 3 has some implementation constraints. Some of them are:

- JML uses the wrapper-based approach to develop the framework for validating assertions at runtime. This means, several new specification methods are created wherein individual specifications are checked, original method bodies are placed inside new methods having the same signature to that of the original method (except they are prefixed with `internal`), and the original method body is replaced by calls to different specification methods. During JML code instrumentation these calls to different specification checking methods are instrumented inside new methods that are prefixed by the name `internal`. In the final version, these statements are embedded inside the original method body (see Figure 3.2). Type-checking them inside a particular method and using the type-checked code inside another method causes a violation in the byte-code format. Since the method body was type-checked under a different method it results in an inconsistent byte-code format. Hence they are merged before type-checking and then they are type-checked. Figure 3.6 illustrates this fact. Four methods are generated by the JML compiler for a single method in the original code. One of the methods namely `internal$m$X` contains delegation code. For minimizing code duplication, the delegation code are generated by the JML compiler inside the `internal` method along with other specification methods. In the final version, as shown in figure, the merged AST has method bodies of `m` and `internal` swapped. Thus type-checking the code before merging results in incompatible byte-code.

- For inline assertions, the constraint checking code may have dependency with the existing code inside the method body. For example in Figure 3.7, we have an inline

<div align="center">41</div>
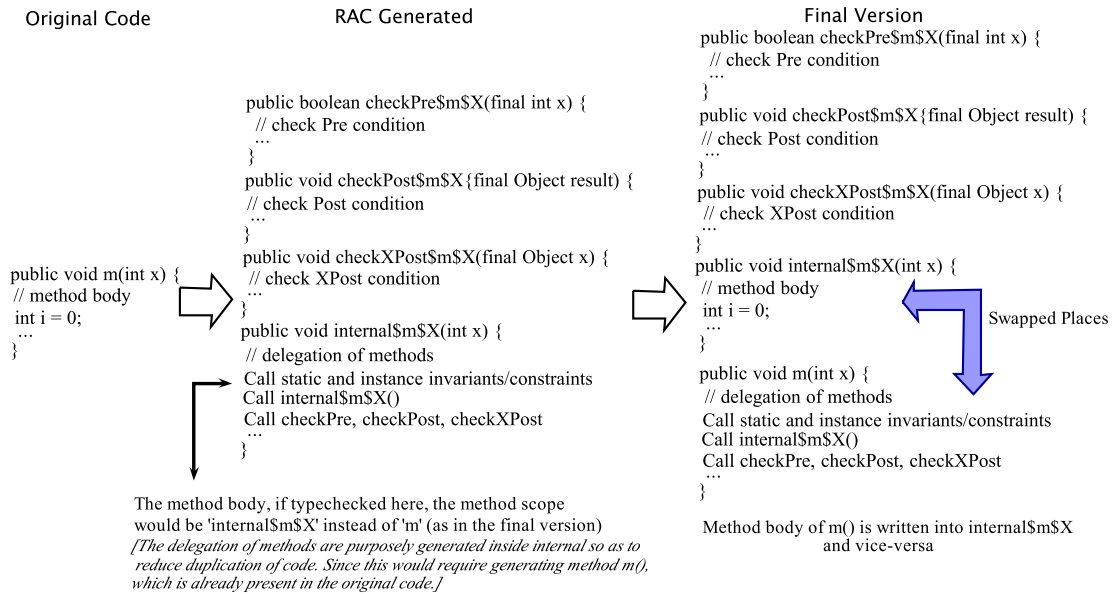
Figure 3.6: Difference in source code between RAC and final version.

assertion `//@ assert i > 0;` where `i` is a local variable. In this case, the generated JML code shown in the figure contains references to `i`. This is highlighted in the figure inside a box. If type-checking is done *only* on the JML code, it would obviously give a compilation error, `i cannot be resolved`, since `i` is not declared locally inside the JML generated code. Further, having a field declaration with the same name `i` in the same type, may complicate matters, as the local `i` would be wrongly type-checked to the field `i` (since the local `i` is not present or visible inside JML code). One simplistic approach to this problem would be to add a temporary variable in the JML code (for `i`), however this may result in code duplication. Therefore we, merge the JML code with the original code prior type-checking. In the figure, the generated JML code does not contain any local declaration of `i` since it is already declared in the original code. However, the final version of the code after merging contains both the declaration of the the local variable and the assertion checking code.

- A major architectural problem in Eclipse is that it does not support incremental compilation at type, at method or at statement levels. Most traditional compilers like

42
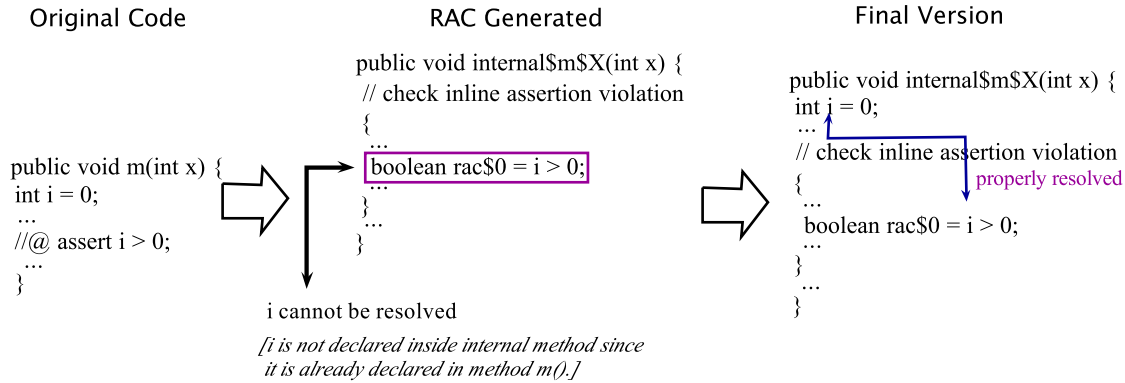
Figure 3.7: Problem of type-checking the RAC code, in presence of inline assertions

the Eclipse Java compiler, assumes that the different phases like scanning, parsing, resolving, and byte-code generation are visited *once* per compilation unit. This is not the case for incremental compilation. Moreover, most incremental compilers like the C# compiler of Microsoft Visual Studio® provides some kind of temporary data structure, like `output_file_name.extension.incr` to take advantage of incremental compilation. The data structure stores information about the status of compilation: its lookup tables, internal AST, and others. The Eclipse framework does not provide these features. Hence incremental compilation is not possible in Eclipse at a level lower than the compilation unit.

- The Eclipse framework is designed so that type-checking starts from the top level construct, i.e., compilation unit. If a particular type is type-checked then it abandons the type-checking of the specific type and moves onto the next. Eclipse uses the visitor pattern to visit all the nodes. During type-checking, all the children of a particular node are visited (type-checked) first and then the node itself is type-checked. On successful completion, the node is marked type-checked by setting the AST bits to *type-checked.* For better performance, children of a particular node are visited only if the node is marked not type-checked. Thus if a compilation unit node is marked type-checked then all the children of this node are not visited as it is assumed to be

43

type-checked. In JML, this strategy is not applicable since instrumented methods, instrumented types, and instrumented statements are added onto existing type and method declarations in the second pass. Thus even though if the top-level node is type-checked (from the first pass), some of its children may not be type-checked which would result in compiler error. Similar behavior is seen when it type-checks for a method; the method body is visited only if the method header is not type-checked. This is also true at the statement level.

3.2.3 Modified Approach on the Eclipse Platform

The changes to our modified approach as compared to the general approach are:

1. In the first pass parsing and type checking of original source code is done.

2. Using this type-checked AST, instrumented JML code is generated in source code format, which is further saved in a temporary file.

3. The RAC code is parsed; parsing the JML code creates an initial AST.

4. This untype-checked AST (JML AST) is merged to the original, un-checked AST (original-AST). *Note that merging is done between untype-checked ASTs and not type-checked ASTs as mentioned in the general approach.*

5. Type-binding, type-checking, and flow-analysis are done on this merged AST. *Note that this is similar to the general approach except in the modified approach all the nodes in the merged AST are visited.*

6. The resulting AST is sent for code generation.

The Eclipse framework does not provide us with any API that we can take advantage of for the incremental approach. The unit of increment in Eclipse is a compilation unit. However, for the JML compiler the unit of increment is a sequence of Java statements. A

sequence of instrumented Java statements cannot be simply parsed and type-checked and then merged with the original AST in the Eclipse framework. This is because the Eclipse framework does not contain APIs to support parsing and type-checking at the statement level, a major reason for our inability to implement the general approach. Thus, we propose an alternative solution. We merge the two AST's before type-checking phase in the second compilation pass. In our approach, we merge the two ASTs as not-type-checked objects (even though of course the original AST was type-checked in the first pass) rather than type-checked ones.

3.2.4   Implementation Details
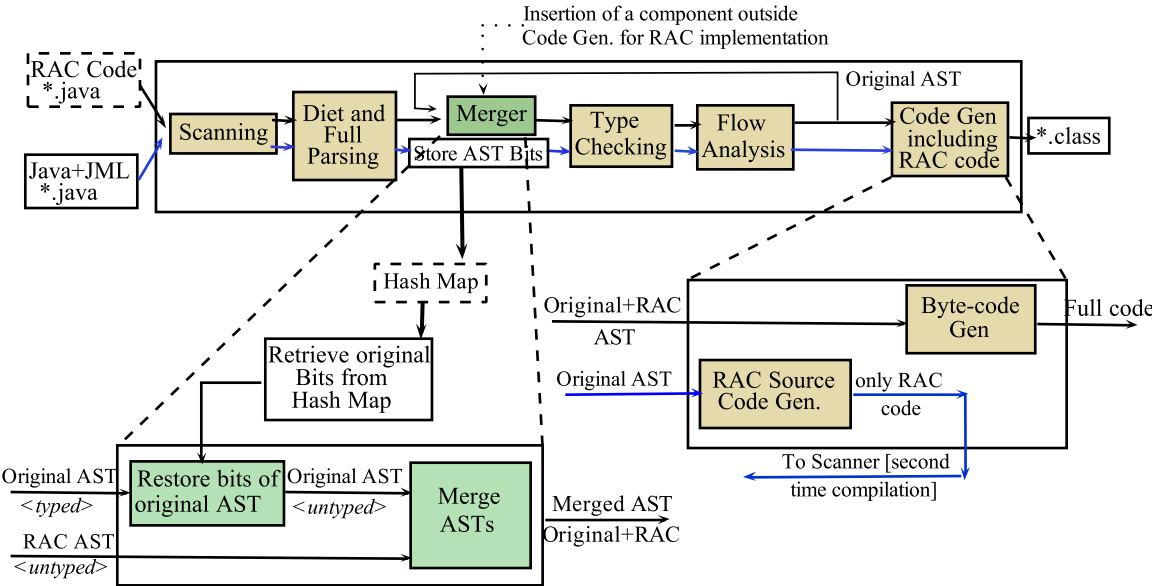
Figure 3.8: Proposed architecture designed on the Eclipse Platform

Figure 3.8 shows how the JML compiler has been implemented on the Eclipse platform. Scanning, diet and full parsing, type checking, flow analysis, and a major portion of code generation form a part of the Eclipse JDT architecture (refer to Section A, for more detailed explanation). Merger, store AST bits and a part of code generation are parts of the back-end support for the JML compiler implementation. All the phases have two inputs as

45

shown in the diagram. They correspond to the first pass and second pass of compilation respectively. The bottom arrows correspond to the first pass of compilation. In the first pass, a Java file is scanned, parsed, type-checked and flow analyzed. Prior to type-checking, the bit pattern for each AST node (see Section 1.1.5) that is present in the original AST is stored in a hash map for later retrieval. In the code generation phase, the JML source is generated from the original AST and only the JML code is sent for the second pass of compilation. In the second pass, the input to the scanner is the JML code which is scanned and parsed. Between parsing and type-checking, merging is done. The inputs to the Merger phase, as shown in the figure, is the original AST that is type-checked from the first pass and the JML AST that is not type-checked. Inside the merging phase two things happen to bring the merged AST into a consistent state. First, the original AST is converted from type-checked to untype-checked AST. This is done by reverting the current AST bits with that of the bits prior to type-checking for the original AST. The next step that follows, is merging the two ASTs. The merged AST is type-checked, flow analyzed and code generated.

Storing and Retrieving of AST Bits

As discussed in Section 1.1.5, the AST bits act as a blue-print for all the information contained in the AST node. To facilitate merging of untype-checked ASTs, the type-checked original AST needs to be transformed into an untype-checked AST. This is required because the bits corresponding to each node of the original AST signifies that the node has already been type-checked. In such a case, the children for these nodes would not be visited. Under normal conditions this always holds true. However, since the merged AST contain both JML AST and original AST nodes, some of them would be type-checked and others not. To force visiting all the nodes in the merged AST , retrieval of AST bits for the original AST is done. This is done by visiting each of the AST nodes before type-checking phase in the first pass, such that we collect the information of the bits of each AST node and save it in a HashMap where the key is the AST visited and the value is the bit pattern corresponding

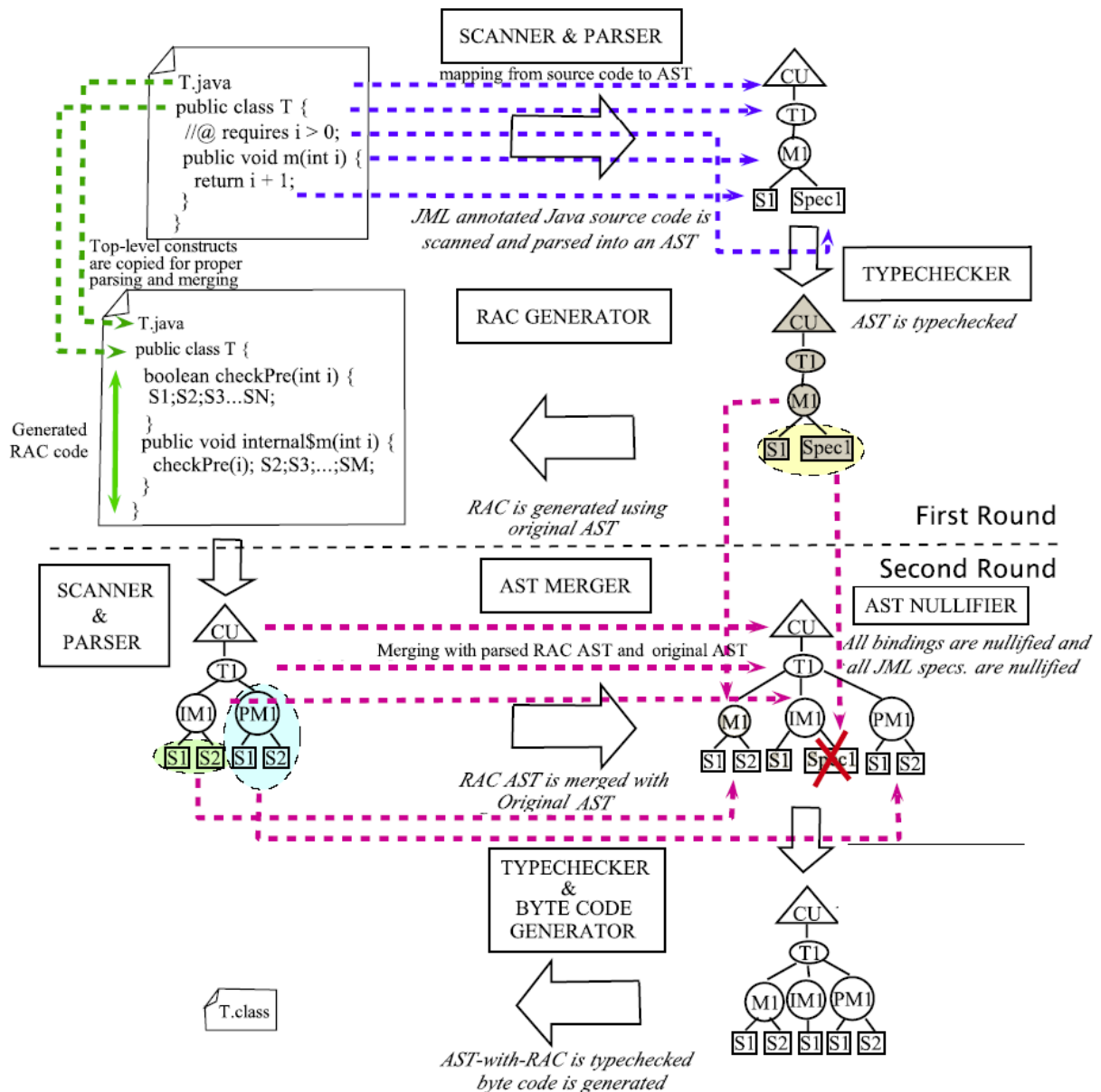Figure 3.9: Steps involved for generating RAC code

to the AST node. To implement this scheme, the visitor pattern was adopted, where every AST was visited and the corresponding bit pattern was stored. This process as shown in Figure 3.8 happens in the first pass prior to type-checking the original AST nodes. In the second pass, while merging the original AST with the JML AST, all the nodes of

47

the original AST are again visited. For every node visited, its current bit pattern (which signifies that the node was type-checked) is now overwritten by the initial bit pattern (prior to type-checking) which is stored in the hash table. Furthermore since type-checking on a node results into two side-effects: the AST bit pattern being changed and the resolved type being set; we also require to nullify the resolved type of the node. In the second pass, while we visit each of the nodes to re-set the AST bit patterns, the corresponding resolved type is also nullified since type-checking the same node twice may have problems.

Figure 3.9 illustrates the modified approach in a step-by-step fashion. Initially there is a Java source code that is annotated with a JML specification. For simplicity, the Java source code contains a single method `void m(int i)` having a simple JML pre-condition. As a first step, this file is scanned and parsed into an AST. The figure gives a one-to-one relationship between the code and the AST generated. The different nodes are shown differently in the figure to represent the different Java models. This AST is further type-checked. In the figure the type-checked nodes are shaded in gray. This AST is used to generate the JML code in source code format. Top-levels constructs like the compilation unit, type declarations are copied from the original AST. The source code is scanned and parsed to generate JML AST. An important phase following this is AST merging. Inside this phase, an important step occurs. The AST Nullifier that nullifies the resolved type of the nodes as well as re-set the AST bit pattern is done. JML annotation nodes are also nullified in this step. The statements between the the internal method IM_1 and the original method from the original AST `M` are swapped. The resultant AST is then further type-checked and is forwarded to byte-code generation.

The overall approach towards incremental compilation on the Eclipse platform can be outlined as follows:

1. In the first compilation pass, before type-checking and resolving, the bit pattern for each AST node is stored.

2. The first compilation pass ends by generating JML RAC code and pretty-printing to

a temporary file. This file contains only JML RAC code.

3. The second pass starts by scanning and parsing only JML RAC code. This results in an AST.

4. The bit pattern for the original AST is restored back by visiting each of the original nodes.

5. The two ASTs are then merged into one single AST.

6. This AST proceeds to the type-checking and resolving phase.

7. The type-checked AST then proceeds for the final byte-code generation phase and completes the second compilation pass.

The Relationship between Incremental Compilation and AST Merging

Figure 3.10 shows how our proposed technique is related to incremental compilation[‡] on the Eclipse framework. The upper half of the figure shows the steps of manual addition, deletion, and necessary edits on the original source code in a step-by-step fashion showing the incremental changes necessary to convert the original code into JML-specific code. The final version of the original code results in a edited code where some of the statements are already type-checked and others are not. The ones that require type-checking are the added or edited code. In the figure, steps 1 and 2 correspond to the deletion of method specification, that is removing the JML annotation and adding necessary methods into the source code for constraint checking purposes. In step 3 a new method namely `internal$m` is created. In step 4, the inline annotation is removed and its corresponding source code is written. As a final step for the method `m`, the method bodies between `m` and `internal$m$` are swapped.

[‡]Incremental compilation at the method, statement level

49
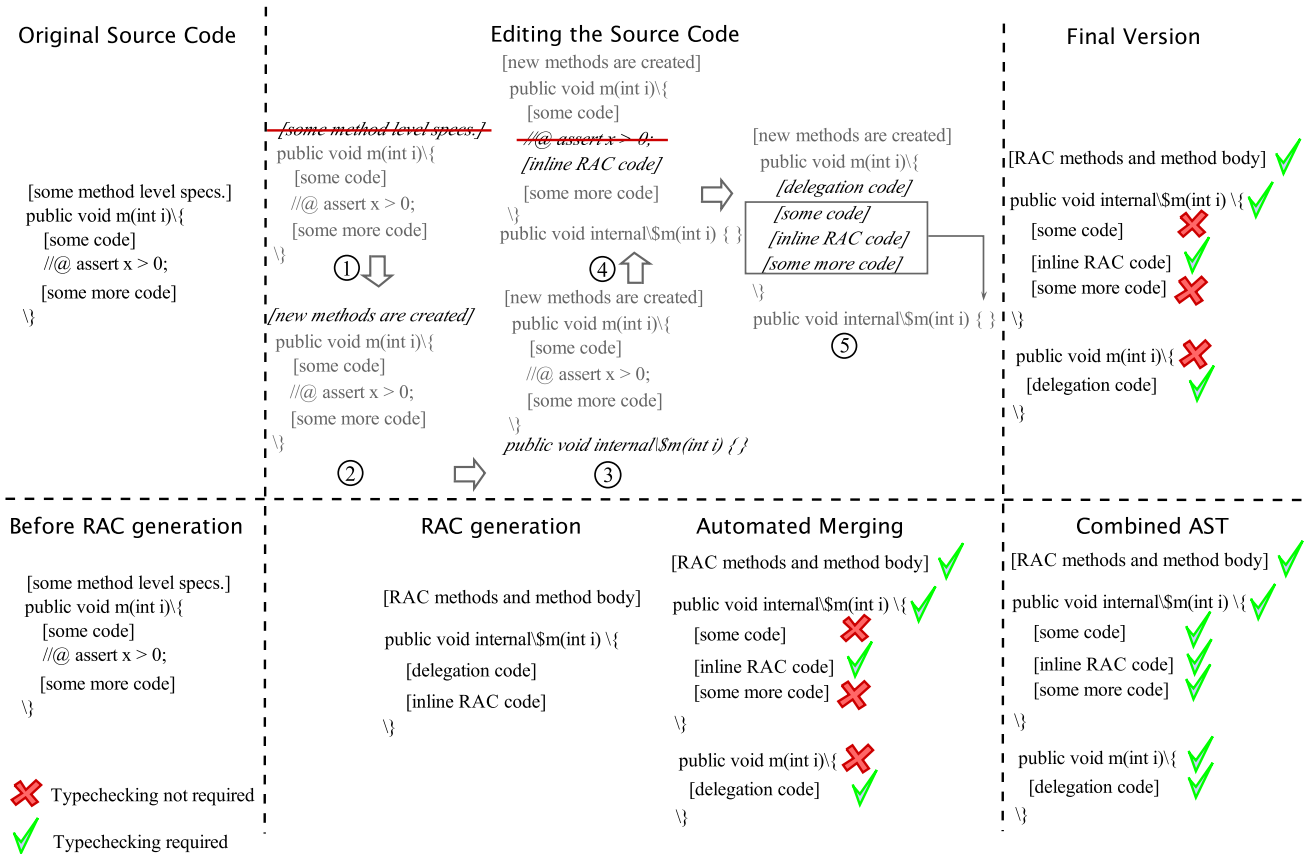
Figure 3.10: Comparison between proposed approach and incremental approach

The bottom half of the figure depicts our proposed technique showing how the compiler generates and merges the code automatically in a similar fashion [§]. Steps 1 through 4 in our approach is taken care by the JML compiler, where runtime assertion code is generated. Step 5 and parts of step 4 are simulated in the automated merging process. The resultant code is similar to that of the final version of the code shown in upper half of the figure. However due to problems in the Eclipse framework, in order to type-check the method body, we require to nullify the types and re-set AST bit pattern of the already type-checked code (refer to Section 3.2.2). Hence, nullification is done which results in the final version where

---

[§]This technique is slightly different from our proposed approach where merging is done prior nullification. In the original approach, nullification is done before merging.

all the nodes are to be type-checked.

3.3 Related Work

3.3.1 Incremental Compilation

Most incremental compilers has been built on systems that provide data structures to store intermediate results and tightly integrate with all the components. Fritzson [Fri83b] in his paper demonstrated incremental compilation at the statement level. He also showed [Fri83a] that to develop such a system extra information is required which is not the case in a traditional batch compiler. Earley and Caizergues [EC72] presented an incremental compiler for Algol and PL/I that make use of an internal data structure in order to do incremental compilation. Crowe, et.al [CNHM85] in their paper titled, "On converting a compiler into an incremental compiler", showed that a global storage data structure and an incremental parser is required to be built for an integrated programming environment. It is interesting to note that in all the approaches stated above "extra" data structures have been used to develop incremental compilers. In my approach, AST merging technique was used to show the effectiveness of developing an incremental compiler on top of a batch compiler without the help of any "special" data structures. Unlike a traditional incremental compiler, whose final output is always up-to-date, in our case, the intermediate representation is always kept up-to-date.

Incremental compilation techniques has been successfully used in developing interactive environment. The earliest known effort was in developing Magpie, an interactive system for Pascal which used incremental compilation to specify debugging actions in Pascal [SDB84] [DMS84]. Recently incremental compilation techniques has been revived with the growing popularity of integrated development environments (IDEs). Some of the popular ones are

Visual Studio® IDE¶, IntelliJ® IDEA‖, Eclipse** which support incremental compilation in some form or the other. Montana is another open, extensible integrated programming environment for C++ that supports incremental compilation by using a persistent code cache that serves as a central source of information for compiling, browsing, and debugging [Kar98]. In this thesis, I show that it is possible to develop an incremental compiler that is fully integrated with an IDE.

3.3.2   AST Merging

In Premkumar's work [Dev92] and [HGM00] applications of AST merging has been successfully implemented in source code analyzers and program analysis tools. More recently, Angyal et.al [ALC07] showed abstract syntax tree (AST) differencing and merging techniques were applied on model-driven software development. Here synchronization between a platform independent model (PIM) and a platform specific model (PSM) were achieved using three way AST merging. Rosenblum [Ros92] in his paper describes APP as a standard preprocessor pass of cc which uses limited capability of merging algorithm at the source-code level. All of the above work focusses primarily on source code analysis instead of compilation techniques. Most of the efforts in AST merging has been concentrated on pre-processors or source code analyzers whose input is source code and output is also source code. Since the output to these AST merging techniques was primarily source code, mutating the nodes of the AST was easily possible. They use "lazy" AST merging approach that fails to take care of "consistent" merging and stresses on pretty-printing only.

In this thesis I explore new directions by using AST merging technique during compilation phase for byte code generation. My problem domain is extended beyond the standard because unlike others who have used it to generate source code, my approach is based on creating byte code. In addition to this problem extension we also adopt a method which

---

¶Visual Studio® is a registered product of Microsoft Corporations

‖IntelliJ® IDEA is a registered product of JetBrains

**Eclipse is an open-source product

is different from the "lazy" merging approach since our focus is on generating byte code instead of source code. This is because the problem requires consistent mutation of the AST nodes resulting in correct byte code format.

3.3.3   Runtime Assertion Checking

Multiple contract based Java systems exist that suport assertions. These include but are not limited to ContractJava [FF01], iContract[Kra98a], Jass - Java with assertions [BFMW01] and Jcontract [Par] that uses pre-processor techniques where they primarily use double-round strategy for generating assertion checking code. A significant contribution of this thesis involves developing a technique that can be used in such tools and pre-processors leading to performance gains.

3.4   Summary

In this chapter I outlined a general approach to solve the problem of slow compilation speed of the current JML compiler. I also tailored this approach to implement the JML compiler on the Eclipse framework. Developing the compiler was not without challenges. I required to make multiple changes in the general approach such that it can be implemented in the Eclipse framework. I also established the relationship between AST merging and incremental compilation; elaborating on how it can be built on top of a batch compiler.

Chapter 4

Evaluation

In this chapter, I describe the evaluation strategy and its corresponding results for the evaluation of our proposed architecture. Most of Levels 0 and 1 features enlisted in the JML reference manual [LPC$^+$06] have been implemented. The features that fall under these levels are most commonly used in JML specifications. I briefly give the overview of the test cases that were run to test the completeness of the compiler construction and also the test cases to test the performance of the proposed approach. Also discussed in this section are my other contributions in my thesis and how the current implementation solves the problems of JML2.

4.1 Test Cases

To test whether the proposed approach is indeed a feasible solution and can be used to compile sufficiently large and complicated real applications, we tested our compiler with all the 35K test cases of the Eclipse compiler, together with almost 3K other test cases for JML alone. Table 4.1 shows the break down of the test cases.

4.1.1 JML Test Cases

Several existing JML test cases were used to test the current framework of JML on the Eclipse framework and some more new test cases were also written especially for the new approach. Here we focus only on the backend implementation (which is the focus of this thesis). About 2100 new Junit test cases were written to test the new framework. They are packaged in `jml4rac.RunRACTests`. All of them are grouped into different features of

Table 4.1: Distribution of test cases across top-level feature tests

Features to Test	Packages	Versions*	Test cases
Integration	org.eclipse.jdt.core.tests.RunBuilderTests	1.3	195
	org.eclipse.jdt.core.tests.RunDOMTests	1.1	3111
	org.eclipse.jdt.core.tests.RunFormatterTests	1.5	1623
Compiler	org.eclipse.jdt.core.tests.RunCompilerTests	1.1	23346
	org.eclipse.jdt.core.tests.RunModelTests	1.1	6138
JML support	org.jmlspecs.eclipse.jdt.core.tests.RunJML4Tests	422	560
RAC support	org.jmlspecs.eclipse.jdt.core.tests.jml4rac.RunRACTests	423	2072
	org.jmlspecs.eclipse.jdt.core.tests.jml2rac.RunRACTests	401	501
JML Samples	http://www.jmlspecs.org/samples		15
Benchmarks	`antlr`, `jython`, `jfreechart`, `xalan`, `fop`, `hsqldb`		8

JML. Figure 4.1 shows the different features that were tested.

RAC Tests
- ASTMerger Test
- ClassLevelSpecification Test
- ConstructorLevelSpecification Test
- DesugarSpecification Test
- Expression Test
- GhostOrModel Test
- GenericType Test
- Inheritance Test
- Inline Test
- InnerClass Test
- Interface Test
- MethodSpecification Test
- QuantifiedExpression Test
- SignalsClause Test
- ViolationReporting Test
- WrapperMethodGenerator Test

Figure 4.1: RAC Test classes grouped as per JML features

Amongst others test classes, an important test class is the `ASTMergerTest` which focusses on testing the new AST merging technique that was proposed in this thesis. This class tests almost all the Java features with and without JML annotations to check whether proper merging and correct code generation is possible using the proposed approach. An illustrative example of an annonymous class is shown in Figure 4.2. By default, the Eclipse Java compiler adds a synthetic method `X()` to every anonymous class declaration. Since

this is a synthetic method [†] it does not result in any compilation error. The method `X` is treated as a synthetic constructor by the compiler. However in the second pass, the method `X` is interpreted as a simple method having no return type. This generates a compiler error since only constructors have no return types. A possible solution that I suggest and implement in this thesis is to have a nullifier visitor class that would visit every AST node to nullify some important data structure which were generated in the first pass (see Section 3.2.4) such that the method `X` is removed.

Other test classes that tests some new features in JML that were not previously supported by the current JML compiler are `GenericType Test`, `InnerClass Test`, `ViolationReporting Test` and some parts of `Inline Test`. All of them test the new features added in this thesis, including supporting JML specifications for generic types, specifications for nested classes, enhanced violation reporting mechanism and support for inline specifications in member classes, quantified expressions in inline assertions and supporting new Java constructs such as the `enhanced for loop`.
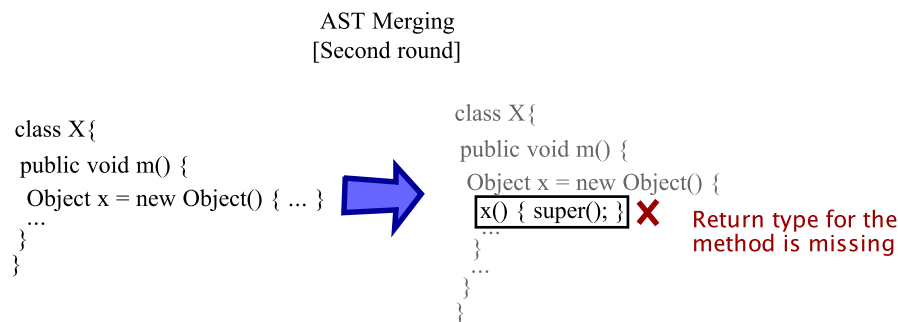

Figure 4.2: Why proper merging and nullifying must be done prior to code generation

---

[†]The compiler generates this code and is thus not required to be parsed or type-checked; it is assumed to error-free.

56

4.2 Performance Measure

Performance of a compiler is a measure of its compilation time, memory usage during compilation, runtime speed and many others. Program correctness is also one of the most important criteria for a compiler being built. In terms of JML, the compilation time is an interesting problem. It is affecting users from using the compiler for any practical purposes since it is inherently very slow than a Java compiler. In our evaluation strategy, we try to measure the compilation time and the memory used for compiling Java classes annotated with JML specifications. The runtime speed is not measured since it is not a focus for my thesis.

To do this, Eclipse framework provides us with a tool to measure the performance. The class `CompilerStats.java` contains a data-structure to measure the parse time, resolve time, analyze time and generate time. We make use of this data-structure to measure our compilation time performance. To measure program correctness, we plot graphs showing the percentage of success to failure of our approach with that of the current JML compiler across all test cases. The results obtained are discussed in the follow sections.

4.3 Test Results

This section discusses the different test experiments and their corresponding results that were carried out on the two different compilers of JML, current and new.

4.3.1 Performance Testing

Section 2.3.5 discussed that the current JML compiler is very slow than a Java compiler because of the nature of compilation in JML: *separate compilation* and double-round strategy. To compare the performance between AST merging and double-round it is important not to include the slowness due to separate compilation. The number of test cases that were test run was about 3000. Due to space constraints, table 4.2 tabulates only the total

Table 4.2: JML Test results

Approach	Total time (secs)	Average (ms)
Double-round (full)	179	6.5
AST merging (full)	127	4.6
Double-round (second pass)	125	4.55
AST merging (second pass)	76	2.75

and average time [‡] of the two approaches. It can be observed that on an average basis AST merging technique is 142% faster than the double-round technique. The main reason for this difference is because of the variations in total compilation times of the two approaches. The main speedup occurs in the second pass of JML code generation that is 160% faster in the former case.

JML Sample Specifications

The JML distribution contains several sample specification packages in order to test and verify new JML tools for executability and completeness of the tools [Lea]. Each of these samples are heavily annotated with JML specifications. For a detailed description of the packages and their characteristics see Table 2.1. In Figure 4.3 the results of the raw compilation time of AST merging approach, double-round approach, the current JML compiler and javac are shown. In Figure 4.4 the graph shows the relative-slowness of the two approaches compared to javac. It can be noticed that in all cases, the *proposed approach is faster than the double-round by a factor of 1.4*. This also shows that the proposed approach is faster when tested against the JML test suite.

Benchmark tests on JML Compiler

To test the performance of the new JML compiler using AST merging and also to test the program correctness of `jml4c`, the compiler was tested on some of the test suites of the DaCapo benchmark. The DaCapo benchmark is a set of open-source client-side Java

---

[‡]The compilation time of the failing test cases were not included
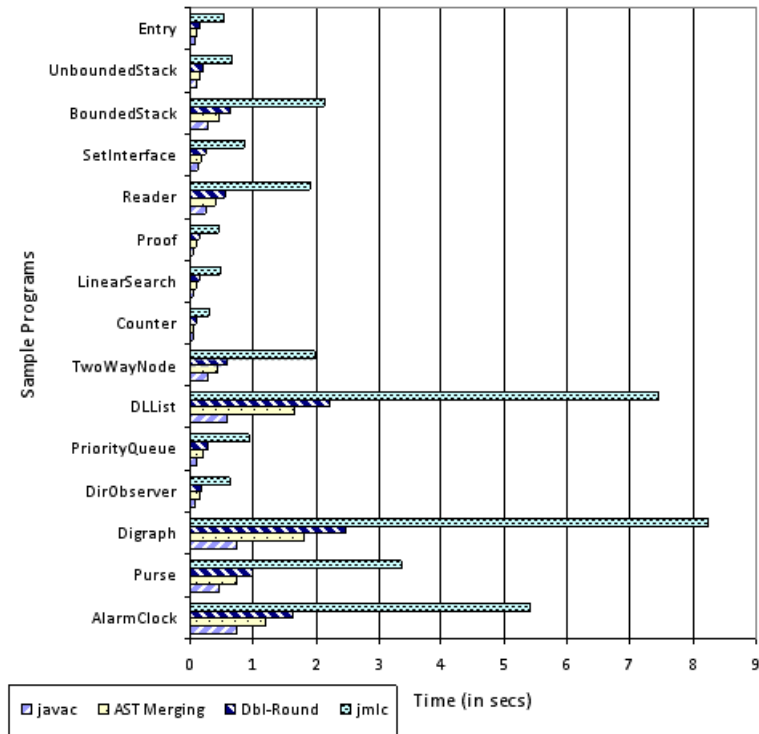
Figure 4.3: Compilation time of all approaches

benchmarks. The motivation behind testing the compiler against these benchmarks was to re-assure that the compiler can be used in "real" applications. The following subset of the DaCapo benchmark was tested:

1. **antlr** A parser generator and translator generator.

2. **chart** A graph plotting toolkit and pdf renderer.

3. **fop** An output-independent print formatter.

4. **hsqldb** An SQL relational database engine written in Java.

5. **jython** A python interpreter written in Java.

6. **xalan** An XSLT processor for transforming XML documents.
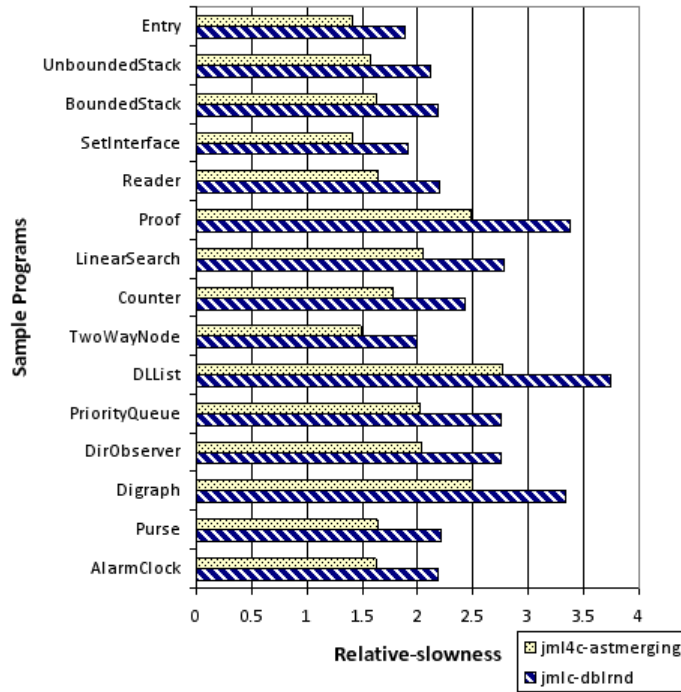
Figure 4.4: Relative slowness of AST merging and double-round approach to javac

The reasons for using these benchmarks over the SPEC-benchmark is because they are open-source and their complexity is higher than those in SPEC-benchmark [BGH+06a] [BGH+06b].

The results of running these benchmark on the Eclipse Java Compiler, jmlc and jml4c has been tabulated in Table 4.3. Each benchmark is followed by a number that signifies the number of packages for that benchmark e.g., **antlr** has 90 packages. The last two rows tabulates the arithmetic mean and geometric mean of KLOC, number of class files and compilation time of all the benchmark. Figure 4.5 shows the graph of the compilation time of each of three compilers and 4.6 shows how jml4c and jmlc are compared with respect to the Eclipse Java compiler. Figure 4.7 gives the characteristics of the benchmark in terms of lines of code and class files generated. Note that the number of lines and class files generated is different between the Eclipse compiler and jml4c. This is because jml4c instruments runtime assertion code into the source code and adds an inner class for every

60

interface.

From the table, it can computed that the speed-up of AST merging on double-round is the following:

$$S_{am} = \frac{\frac{\sum_{k=1}^{6} t_k^{jmlc}}{6}}{\frac{\sum_{k=1}^{6} t_k^{jml4c}}{6}} \approx 1.59 \qquad (4.1)$$

$$S_{gm} = \frac{\sqrt[6]{\prod_{k=1}^{6} t_k^{jmlc}}}{\sqrt[6]{\prod_{k=1}^{6} t_k^{jml4c}}} \approx 1.54 \qquad (4.2)$$
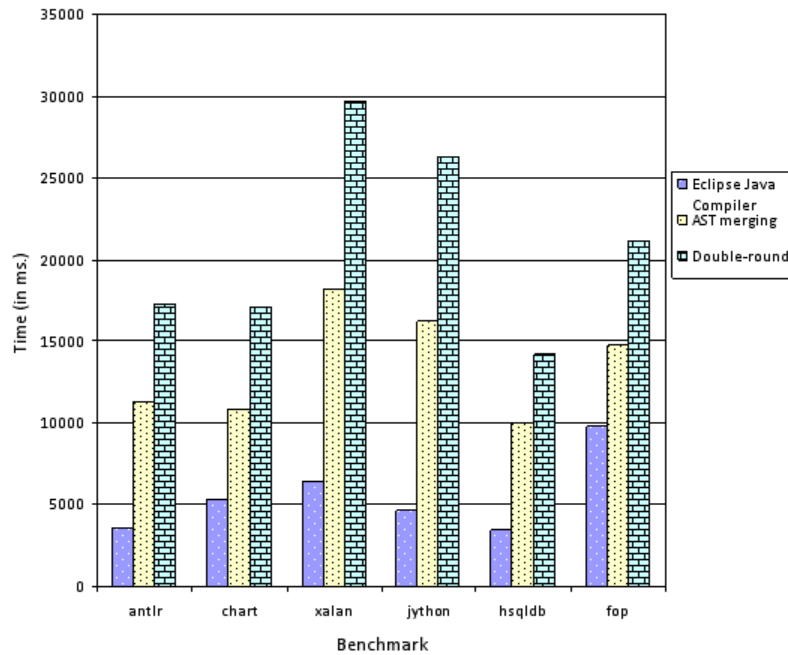
Figure 4.5: Compilation time of three compilers

Overhead of JML4c compared to the Eclipse Java Compiler

To study the overhead of the new JML compiler with respect to the Eclipse base compiler we performed some tests. The same test cases that were used to test JML specifications were used to test the overhead of JML4c. Before running the test cases, it was made sure

Table 4.3: Benchmark results

DaCapo				
Benchmark	Compilers	KLOC	No. of class Files	Compilation Time (in ms.)
antlr - 90	Eclipse Java Compiler	67.1	660	3531
	JML4c	1002.3	722	11299
	jmlc	1002.3	722	17302
chart - 65	Eclipse Java Compiler	304.3	1025	5346
	JML4c	2142.2	1128	10827
	jmlc	2142.2	1128	17107
xalan - 41	Eclipse Java Compiler	338.85	1202	6457
	JML4c	1996.5	1268	18222
	jmlc	1996.5	1268	29702
jython - 55	Eclipse Java Compiler	240.9	906	4641
	JML4c	3808.3	941	16243
	jmlc	3808.3	941	26313
hsqldb - 28	Eclipse Java Compiler	247.7	651	3453
	JML4c	1582.2	702	9958
	jmlc	1582.2	702	14224
fop - 51	Eclipse Java Compiler	237.9	660	9843
	JML4c	2420.4	1939	14764
	jmlc	2420.4	1939	21163
min	Eclipse Java Compiler	67.1	660	3453
	JML4c	1002.3	702	9958
	jmlc	1002.3	702	14224
max	Eclipse Java Compiler	338.85	1789	9843
	JML4c	3808.3	1939	18222
	jmlc	3808.3	1939	29702
arithmean	Eclipse Java Compiler	239.46	1038.83	5545.17
	JML4c	2158.65	1116.67	13152.57
	jmlc	2158.65	1116.67	20968.5
geomean	Eclipse Java Compiler	214.8	974.8	5175.9
	JML4c	1992.2	1047.72	13213.9
	jmlc	1992.2	1047.72	20283.5
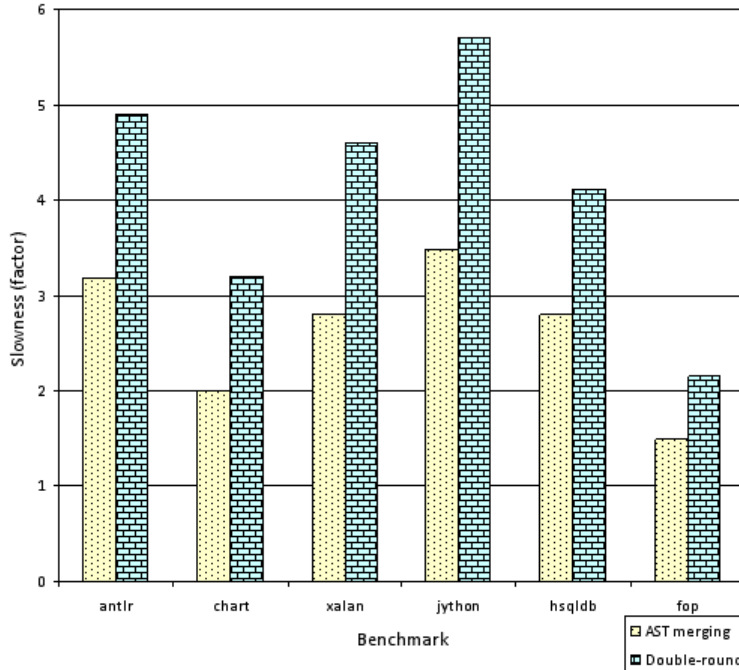
Figure 4.6: Slowness-factor of two approaches w.r.t. Eclipse Java Compiler

that the JML annotations were converted to comment lines. Figure 4.8 shows the overhead
of the new compiler w.r.t. the Eclipse Java compiler. On average, JML4c is 1.37 times
slower than the Eclipse Java compiler.

Overall compilation speed-up

To compare the overall performance speed-up of the new compiler, jml4c with respect to
jmlc, we tested them on two different sets of test cases. One set of test cases contained JML
annotations and the other set contained no such annotations. We tested the two compilers
using the test cases of the JML sample package: first with JML annotations enabled and
then by disabling them.

It can be observed from the Figure 4.10 that jml4c on average is about three times
faster than jmlc in presence of JML annotations and about 4.5 times faster when there is
no jml annotations.

4.3.2 Testing jml4c with respect to hand-crafted code

Since runtime assertion checker instruments additional code into the original source code, it is interesting to study the performance of the new JML compiler in comparison to hand-crafted code. Here we study the performance difference, if any, between jml4c that automates the generation of RAC code and javac. We feed hand-crafted code to the javac compiler and compare the time difference, if any with jml4c. We use the JML sample package as the test cases. On average, jml4c is 1.51 times slower than javac.

4.3.3 Performance Analysis

To further understand and analyze the test results obtained and to figure out any bottlenecks of the proposed approach we performed another set of tests. In these tests we tried to find how each Java language feature affect our proposed approach. We limited our feature set to type declarations (class, interface, abstract class, inner class, local class, anonymous class), method declarations, field declarations, inheritance (through interfaces and classes), method body, JML specifications for methods and interfaces and certain advanced features like ghost and model fields and model methods. Table 4.4 lists the proportion by which each of these features affect the proposed approach. We may note that not only do types, methods, and specifications affect compilation speed differently in case of JML but also their different types affect a lot. Amongst the different types, a single class has the highest contribution and a local class has the least. For methods, the length of the body plays an important role; the higher the number more faster would it be for JML4c compared to jmlc. Specifications at different levels do not change much since JML uses the wrapper-based approach where more compilation time is used to build the framework i.e., the different specification based methods.

Table 4.4: Java and JML features affecting JML4c

Feature	Delta-Time (in ms.) $\Delta T = t_{jml4c} - t_{javac}$
Class	20.3
Interface	14.0
Abstract class	17.2
Inner class	14.0
Local class	3.2
Field decls.	1.5
Inheritance in interface	3.5
Inheritance in class	3.4
Method decl.	10. 9
Method body	4.7
Method specs.	3.4
Type specs.	3.1
Inline specs.	2.4
Ghost field	3.1
Model field	3.2
Model method	1.6

4.3.4   Testing Compiler Correctness

Almost 40K test cases were run on both the current compiler and the new JML compiler built on the Eclipse platform. The test results are shown in Figure 4.11 showing the percentage of success across all the test cases. For getting a better understanding for the failing test cases, they have been grouped into different feature sets. Figure 4.11(a) shows the result of testing the Eclipse framework itself. It shows how successful we were on building the JML compiler on Eclipse. The small number of failing test cases are due to certain changes to the configuration and other supporting files. Figure 4.11(b) shows the result of testing the new compiler alongwith the current JML compiler against all the test cases of JML and Eclipse. It shows that the success percentage for JML2 in JML test cases is higher than that of JML4c. The reason behind this is exemplified in Figure 4.11(c) and 4.11(d). Many of the failing test cases are due to test cases that contain level 2 or higher features that are not supported by JML4c yet. Another important observation is that the

percentage of success for number of known bugs is higher than JML2. The small number of test cases which fails for levels 0 and 1 features has been further categorized and shown in Figure 4.11(d) as sub-features that are supported or not supported. In summary, the new compiler, JML4c is 99.46 % successful compared to 88.37 % of the current JML compiler.

4.4   Enhancement to the Existing JML Compiler

Apart from performance speed up, support for Java 5 features, several translation rules, unsupported features and failing test cases were fixed in JML4c. In this section, I highlight my contributions by discussing features of the JML compiler that were either added or fixed.

4.4.1   Support for Java 5 features

No JML tool including the JML compiler supports the features that were introduced in Java 5. The most important ones are the introduction of generics and `enhancedforloop`. Since most new code that is written these days contain generics, it is impossible for the current JML compiler to compile these code. Hence, due to the inability to use JML there is a reduction in user code base. In my thesis, the new JML compiler supports Java 5 features. And claims that this would revive the lost user code base. However, JML specifications itself do not yet support Java generics i.e., generics inside specifications.

Support for Non-implementable Types

There are certain types that are not implementable by the JML compiler. Assertions written for these types are not checked at runtime. The types that fall under this category are `enum` and `annotation` types. The current JML compiler, JML2 is not able to parse these types. However, in our case (the JML compiler designed on the Eclipse platform) compilation units containing such types are properly parsed and type-checked. However if JML annotations are present for such types they are ignored i.e., they are not implemented.

66

Table 4.5 shows sample programs that were run and their corresponding outputs.

Table 4.5: Sample programs and their outputs

Enum Type	

`public enum DayOfWeek {`
` SUNDAY, MONDAY, TUESDAY, WEDNESDAY,`
` THURSDAY, FRIDAY, SATURDAY`
`}`
	JML2 Compiler
	`jmlc -Q DayOfWeek.java`
	Syntax error: unexpected token: enum
	JML4c Compiler
	`java -jar jml4rac.jar DayOfWeek.java`
	no problems

Annotation Type	

`public @interface Worker {`
` int id();`
` String engineer() default "NA";`
`}`
	JML2 Compiler
	`jmlc -Q Worker.java`
	Syntax error: unexpected token: public
	JML4c Compiler
	`java -jar jml4rac.jar Worker.java`
	no problems

Support for enhanced for loop

In Java 5, a new form of the for loop was introduced. It is popularly known as `enhanced for loop`. The current JML compiler does not support this form of writing for loop. Moreover, the JML Reference Manual also does not have translation rules defined for such statements and hence loop annotations for the `enhanced for loop` are ignored. In my thesis, I define the translation rule for the `enhanced for loop` and also implemented it which is shown in Table 4.6

Instrumented Code Is Java 5 Compatible

The instrumented code that is generated by the current JML compiler is not Java 5 compatible. Compiling the original code with the JML compiler results in several unnecessary warnings. Most of these warnings comes from the fact that the instrumented code is not Java 5 compatible. In the current implementation all of the instrumented code has been properly translated such that it is compatible with Java 5.

Table 4.6: Translation of enhanced for statement

Original code	Translated code (T_1 is iterable)	Translated code (T_1 is arrayType)
``//@ maintaining I;
//@ decreasing V ;
[L:] for (E_1: T_1)
S```	```{
[L:] while (true) {
[[checkInvii]]
[[checkVarii]]
if (!(B)) {
break;
}
E_1 = rac$jml.next();
S
}
[[checkInvii]]
}
B ≡ rac$jml.hasNext()```	```{
boolean rac$f = true;
T rac$jml = T_1;
T rac$incr = [[init_value]];
[L:] while (true) {
if (rac$f) {
rac$f = false;
} else {
rac$incr++;
}
[[checkInvii]]
[[checkVarii]]
if (!(B)) {
break;
}
E_1 = rac$jml[rac$incr];
S
}
[[checkInvii]]
}
B ≡ rac$incr ¡ rac$jml.length()```

4.4.2 More Features are supported in JML4c

This section discuss the several new features that has been supported in the new JML compiler.

Support for Non-Executable JML Clauses

Every Java class compiled with the JML compiler contains not only its normal content (as would be generated by, e.g., `javac`), but also an embedding of its specification and how to verify it at runtime. Instrumentation code is generated on a per type, per method, per field, and per assertion basis [CL02]. The first version of the JML compiler [CL02] attempted to implement assertion semantics based on classical two-valued logic causing the instrumented

code to be much larger than the source. In [CR08] the expression evaluation of the JML compiler was redesigned. In the current JML compiler the evaluation of expressions became quite straightforward.

However in the current JML compiler the expression evaluation scheme which was redesigned in [CR08] had one serious flaw; possible assertion violations at times were unreported. This was because as a side-effect of the new design, every sub-expression appearing in JML specification were checked at compile-time whether it was executable or not. By executability we mean, can RAC verify at runtime whether the assertions hold or not? If the expression is not executable by the JML compiler (i.e., its value at runtime is undefined) then the entire expression or clause was dropped and no code corresponding to the expression was generated.

```
1  /*@ ensures \result * \result <= i &&
2    @ (* \result is approximately equal to square root of i *);
3    @*/
4  public double sqrt(int i) {
5    // purposely the method body is wrongly implemented
6    return i + 1;
7  }
```

The code-snippet shown above is a motivating example to show the problem of unreported violation in the current JML compiler. Under the old semantics, any call to method `sqrt` would result in `JMLPostconditionError` since for no value of `i` the post condition holds. However this would result in a huge amount of JML specific code. In the current JML compiler, since the right-hand side of the `and` expression of the JML ensures clause is not executable, the entire clause is dropped and hence no violation is reported.

In my thesis, care has been taken such that all violations are reported (even in case of non-executability) and the size of the translated code is also not increased. The approach towards doing this is simple and intuitive. It is based on the concept that: *If a JML clause is not executable then the entire clause is not dropped; rather corresponding code is generated that decides the outcome of the validation of the clause at runtime.* Every non-executable ex-

pression is replaced by `JMLNonExecutableUtil.throwNonExecExceptionForXXX` method where XXX is the type of expression, i.e., boolean, byte, char, double, float, int, long, short. If the expression's type is reference type XXX is replaced by `Object`. The class `JMLNonExecutableUtil` is a utility class and is placed inside the JML runtime package along with other JML error classes. For example the translated code for the `ensures` clause is:

```
...
try {
 rac$b0 = ((rac$pre0 * rac$pre0 <= i) &&
 (JMLNonExecutableUtil.throwNonExecExceptionForBoolean()));
} catch(JMLNonExecutableException rac$e$nonExec) {
 rac$b0 = true;
} ...
```

A normal post-condition violation would be reported since the first sub-expression in the `and` expression violates the post-condition.

JML Quantified Expressions in Inline Assertions

The current JML compiler does not support quantified expressions in inline assertions, however JML4c supports it. In order to reuse the existing quantifier evaluation package while implementing the direct expression evaluation approach, the output of the quantifier translator is wrapped into an inner class that is used in the evaluation of the assertion.

Support for JML Specification for Inner Classes

A class defined within another class is called a nested class. There can be four different types of nested classes in Java: A static inner class, a non-static inner class, local class and anonymous class. The current JML compiler does not support specifications for any

of the nested classes. However in jml4c, support for JML specifications for inner classes is present. This makes the JML compiler more apt to catch a larger set of violations.

4.4.3 Implementation Problems in JML2

This section discusses some of the implementation problems that exist in jmlc.

Problem in ghost field

The code-snippet illustrated below contains two such examples. The code contains a class named `AlarmClock` that contains three fields and some methods, one of which is `setAlarm` method. This method checks if the parameter that is passed into this method is equal to the `alarmTime`, then the ghost variable `alarmRing` is set as true and the method returns a true value otherwise, false. The class contains a static invariant for the *non-static field* `alarmTime`.

```
1  public class AlarmClock {
2    public int alarmTime = 0;
3    public int min = 0, hour = 0;
4    //@ public static invariant alarmTime == min*60 + hour*60*60;
5
6    //@ public ghost boolean alarmRing = false;
7    public static boolean setAlarm(int now)  {
8      if (alarmTime == now) {
9        //@ set alarmRing = true;
10       return true;
11     }
12     //@ set alarmRing = false;
13     return false;
14   }
15   // ...
16 }
```

The code when compiled by the current JML compiler gives the following output:

```
File "AlarmClock.java", line 4, character 41 error: In this static

context "this" is not accessible [JLS 15.7.2] File

"AlarmClock.java", line 9, character 27 error: In this static

context "this" is not accessible [JLS 15.7.2] File

"AlarmClock.java", line 12, character 27 error: In this static

context "this" is not accessible [JLS 15.7.2]
```

The problem with the current JML compiler is that proper translation rule does not exist for translating non-static fields in static scope. The example shown here illustrates two problems: non-static field accessed in static invariant and setting a non-static ghost field inside a static method. These problems have been taken care of in the new implementation of the JML compiler where proper rules exist for translating such types.

Problem in exception reporting

Another problem that exists in the current JML compiler is that if an exception is thrown while executing a specification clause it is not thrown back to the client. An example of such a problem is shown below. In this example, `null` is being passed to the method `product`. The pre-condition for this method is that the length of the parameter `a` is positive. However as can be seen from the code, the method `count` would throw a `NullPointerException` which should be reported back to the client. However this is not done in case of the current JML compiler.

```
1  public class Demo {
2
3    public static void main(String[] args){
4      new Demo().product(null);
5    }
6
7    //@ requires count(a) > 0;
8    public long product(int[] a){
9      long product = 1;
```

```
10    for (int i:a) {
11      product *= i;
12    }
13    return product;
14  }
15
16  public /*@ pure @*/ int count(int[] a){
17    return a.length;
18  }
19 }
```

Current JML Compiler	New JML Compiler
`jmlc -Q Demo.java`	`java -jar jml4rac.jar Demo.java`
`jmlrac Demo`	`java -cp ".:org.jmlspecs.jml4.rac.runtime" Demo`
No Problems	*Exception in thread "main" JMLEvaluationError:*

Problem in Loop Annotations

In JML a loop statement can be annotated with one or more loop annotations. The code-snippet shown below is an illustrative example of JML annotation. It contains a method that returns the sum of all odd numbers contained in the variable `a`. This example has two loop annotations for the while loop, one followed by the keyword `maintaining` and another that follows the keyword `decreasing`. They are written above the loop itself. The first loop annotation describes the range that the variable `i` can take and the second checks whether the variable `i` is decreased by one after each iteration.

```
1 public class Sum {
2  public static long sumOfEvenArray(int[] a) {
3    long sum = 0; int i = a.length;
4    out:
5     /*@ maintaining -1 <= i && i <= a.length;
6       @ decreasing i;
7       @*/
```

```
 8      while (--i >= 0) {
 9       if (a[i]%2 == 0)
10        continue out;
11       sum += a[i];
12      }
13    return sum;
14   }
15 }
```

Due to improper translation rules, the code when compiled by the current JML compiler results in a compilation error as shown in Table 4.7. In Java, a labeled statement must precede the block that contains a labeled continue statement i.e., there should be no statement between the labeled statement and block. However the current JML compiler generates instrumented code that violates this feature. In this thesis, the translation rule has been changed to take care of this problem.

Table 4.7: Compilation result of JML annotation with label statement

JML2 Compiler	JML4c Compiler
`jmlc -Q Sum.java`	`java -jar jml4rac.jar Sum.java`
Target of continue statement is not continuable	*no problems*

In Java a loop statement can either be a `for loop`, `while loop` or a `do-while loop`. The current JML compiler contains incorrect translation rules for translating `do-while` loops that are annotated with JML annotations. Table 4.8 shows the translation rule that exists for the `do-while` loop for the current JML compiler. It also shows the new translation rule where statements `S` is embedded inside an if block. This small change is necessary because if `S` is either a `return` statement, a `throws clause`, a `break` or `continue` statement then the next few lines in the translated code becomes unreachable issuing `Statement is unreachable` compiler error. This can easily be handled if we embed statements `S` inside an if block with the condition set to `true`.

Table 4.8: Old and new translation rules for do-while loop

JML2 Translation rule	New Translation rule
| ```
[[do S while (B)]] =
 {
  <<varDecl>>
  while (true) {
   <<checkInv>>
   <<checkVar>>
   S
   if (!(B)) {
    break;
   }
  }
  <<checkInv>>
 }
``` | ```
[[do S while (B)]] =
 {
  <<varDecl>>
  while (true) {
   <<checkInv>>
   <<checkVar>>
   if (true) { S }
   if (!(B)) {
    break;
   }
  }
  <<checkInv>>
 }
``` |

4.5 Summary

In this chapter, I showed my evaluation strategy and discussed the test results that were obtained from the experiments. The test cases that were used to check the completeness of my compiler consisted of 40K test cases of Eclipse, about 3K test cases for JML, JML sample packages and also on real applications. I also used the test suites from the DaCapo benchmark to test and benchmark the new compiler with the current JML compiler. In this chapter I also discussed some of my other contributions in terms of adding support to Java 5 features by using Eclipse as the base compiler. From the results, it can be concluded that the proposed approach is faster by 160% than the old approach. Overall, I obtained a compilation speed-up of three times than the previous JML compiler, jmlc. It was also shown that jml4c is only 1.5 times slower than hand-crafted code implemented on javac.

(b) Number of class files generated by Eclipse and jml4c

Figure 4.7: Characteristics of benchmark on different compilers

76

Figure 4.8: Overhead of JML4c w.r.t. Eclipse Java compiler

Figure 4.9: Compilation time of jmlc and jml4c with and without annotations

77

Figure 4.10: Compilation time of jmlc and javac in presence of handcrafted code

(a) Test results of integration testing

(b) Percentage of success of two approaches across the different sets of test

(c) Failure tests grouped into feature sets

(d) Percentage of failure tests of levels 0 and 1 that are supported by jml4c

Figure 4.11: Test results of the two approaches

79

Chapter 5

Conclusion

5.1 Future Work

There are several natural extensions to the work presented in this thesis. They include supporting more features of the JML language, improving performance and usability, establishing that the approach can be used for other specification languages and application to other tools.

The JML compiler presented in this thesis does not yet support such JML features as those that are in Level 2 and beyond such as refinement, model programs, and non-functional properties. In JML, the behavior of a method is written in abstract code, called model programs, in a notation similar to the specification statements. JML also has several specification constructs to specify non-functional properties like time and space requirements.
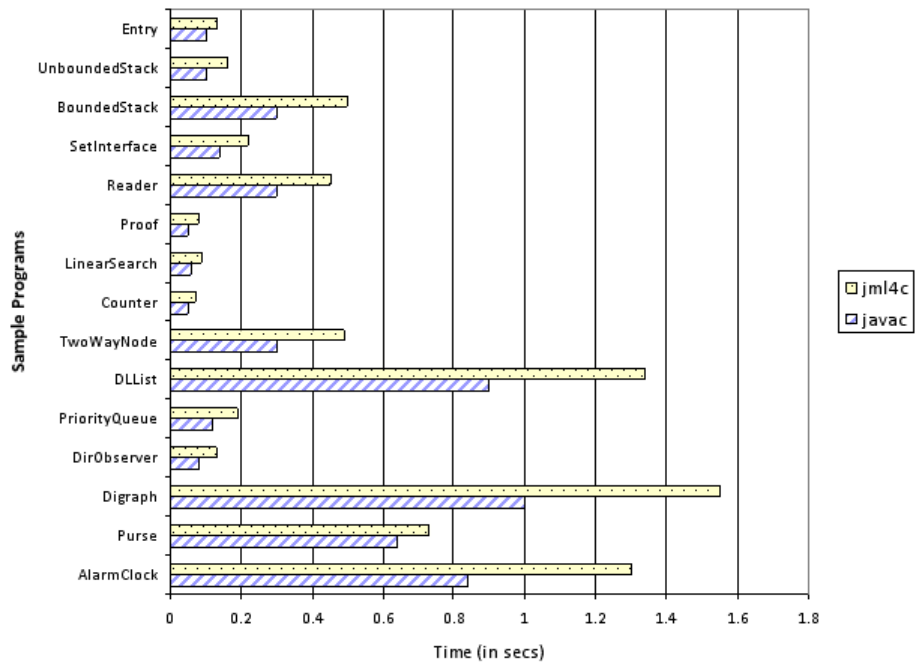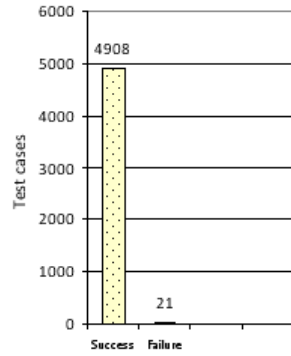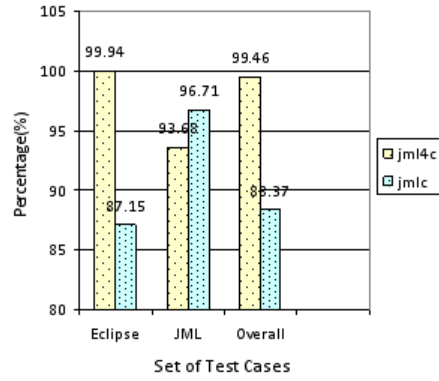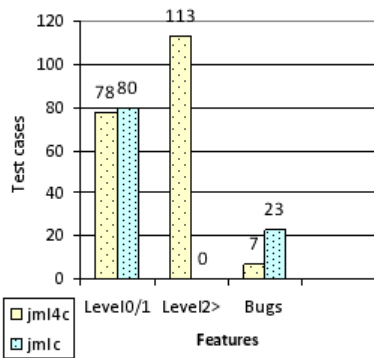
There are two distinct areas of future work: performance of the JML compiler and using AST merging strategy in preprocessors. Unlike Java compilers, a JML compiler parses the source files of all referenced types. This affects the performance of the JML compiler, as parsing is one of the most costly tasks in compilation. This can be taken care by encoding the signature information of JML specifications into byte-code or separate symbol files and by eliminating parsing of referenced types. The JML compiler can also be improved further by optimizing compilation passes, in particular, by abandoning the double-round strategy altogether and generating runtime assertion code directly in byte-code format. Some possible approaches would be to generate parse trees in the type-checked form instead of source code.

In this thesis, I focused my effort on investigating the problems and solution of the slowness of the compilation time of the JML compiler, and use AST merging as a practical and efficient way to merge between source codes. I also relied on the current JML compiler's code base to claim that the JML compiler is faithful to the semantics of JML specifications. However, a formal treatment would be appreciated for the JML compiler to be viewed as providing the correct semantics for JML.

5.2 Summary

The work reported in this thesis was motivated by the lack of practical use of the current JML compiler due to its slowness, lack of integration with an IDE and inability to support Java 5 generics. One can use BISLs to write detailed design documents of program modules and such specifications allow one to clarify and critically evaluate the roles of program modules. In addition, I strongly believe that BISLs can be used in daily programming tasks, such as debugging and testing and when contained within an IDE, it can increase user base and bring more productivity into the development process. I have presented in my thesis, a working JML compiler that has been integrated with a popular IDE.

In this thesis, I have presented detailed approaches for writing the JML compiler on the Eclipse platform. The JML compiler presented in this thesis represents a significant advance over the state of the art in runtime assertion checking.

In Chapter 2, I defined the problem in the current JML compiler that I solved in my thesis. The problem was shown to have several reasons: separate compilation for referenced files, double-round, JML compiler does more work than a Java compiler, etc. In this chapter I explained the architecture behind the current JML compiler especially the double-round architecture.

In Chapter 3, I proposed a general approach towards solving this problem due to the double-round approach. Then I showed how would this be implemented on the Eclipse platform leading to a revised version of the general approach. Eclipse itself is not an

incremental compiler, however it was showed in this chapter that it is possible to use incremental compilation in my approach.

In Chapter 4, I showed my evaluation strategy and discussed the test results that were obtained from the experiments. The test cases that were used to check the completeness of my compiler consisted of 40K test cases of Eclipse, about 3K test cases for JML, JML sample packages and also on real applications. I also used the test suites from the DaCapo benchmark to test and benchmark the new compiler with the current JML compiler. In this chapter I also discussed some of my other contributions in terms of adding support to Java 5 features by using Eclipse as the base compiler. From the results, it can be concluded that the proposed approach is faster by 160% than the old approach. Overall, I obtained a compilation speed-up of 300% than the previous JML compiler, jmlc.

In this thesis I also explored new directions by using AST merging technique during compilation phase for byte code generation. My problem domain is extended beyond the standard because unlike others who have used it to generate source code, my approach is based on creating byte code. In addition to this problem extension I also adopted a method which is different from the "lazy" merging approach since my focus is on generating byte code instead of source code. This is because the problem requires consistent mutation of the AST nodes resulting in correct byte code format.

The successful implementation of the tool using AST merging also provides a partial proof that the approach can be used as an effective framework for developing other tools like preprocessors. I believe that the techniques and approaches developed in this thesis namely AST merging are applicable to other object-oriented programming languages and other tools.

References

[ALC07]   László Angyal, László Lengyel, and Hassan Charaf. Novel Techniques For Model-Code Synchronization. *ECEASST*, 8, 2007.

[BCC+05]   Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

[Bec99]   Kent Beck. *Extreme Programming Explained: Embrace Change.* 1999.

[BFMW01]   Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with Assertions. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[BGH+06a]   S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[BGH+06b]   S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis (Extended Version). Technical Report TR-CS-06-01, 2006. http://www.dacapobench.org.

[BH02]   Jason Baker and Wilson Hsieh. Runtime aspect weaving through metapro-
           gramming. In *AOSD '02: Proceedings of the 1st international conference on
           Aspect-oriented software development*, pages 86–95, New York, NY, USA, 2002.
           ACM.

[Bre06]   Cees-Bart Breunesse. *On JML: Topics in Tool-assisted Verification of Java
           Programs*. PhD thesis, Radboud University of Nijmegen, 2006.

[BS03]   Mike Barnett and Wolfram Schulte. Runtime verification of .NET contracts.
           J. Syst. Softw., 65(3):199–208, 2003.

[CJK07]   Patrice Chalin, Perry R. James, and George Karabotsos. An integrated ver-
           ification environment for JML: architecture and early results. In *SAVCBS
           '07: Proceedings of the 2007 conference on Specification and verification of
           component-based systems*, pages 47–53, New York, NY, USA, 2007. ACM.

[CJK08a]   Patrice Chalin, Perry R. James, and George Karabotsos. JML4: Towards
           an Industrial Grade IVE for Java and Next Generation Research Platform
           for JML. ENCS-CSE-TR 2008-01x, Concordia University, Montreal, Canada,
           2008.

[CJK08b]   Patrice Chalin, Perry R. James, and George Karabotsos. JML4: Towards an
           Industrial Grade IVE for Java and Next Generation Research Platform for
           JML. In *VSTTE '08: Proceedings of the 2nd international conference on Ver-
           ified Software: Theories, Tools, Experiments*, pages 70–83, Berlin, Heidelberg,
           2008. Springer-Verlag.

[CK04]   David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML.
           Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-
           R0413.

[CL02]   Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the
             Java Modeling Language (JML). In *Proceedings of the International Confer-
             ence on Software Engineering Research and Practice (SERP 02), Las Vegas*,
             pages 322–328. CSREA Press, 2002.

[CLSE05]   Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards.
             Model variables: cleanly supporting abstraction in design by contract: Re-
             search Articles. *Softw. Pract. Exper.*, 35(6):583–599, 2005.

[CMLC06]   Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. Mul-
             tiJava: Design rationale, compiler implementation, and applications. *ACM
             Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[CNHM85]   Malcolm Crowe, Clark Nicol, Michael Hughes, and David Mackay. On convert-
             ing a compiler into an incremental compiler. *SIGPLAN Not.*, 20(10):14–22,
             1985.

[Cor]   Microsoft   Corporation.   Enable   Incremental   Compilation.
             http://msdn.microsoft.com/en-us/library/aa691231(VS.71).aspx.   (Date
             retrieved: October 1, 2009).

[CR08]   Patrice Chalin and Frédéric Rioux. JML Runtime Assertion Checking: Im-
             proved Error Reporting and Efficiency Using Strong Validity. In *Proceedings
             of the 15th International Symposium on Formal Methods (FM '08)*, pages 246–
             261, 2008.

[Dev92]   Premkumar T. Devanbu. GENOA: a customizable language- and front-end
             independent code analyzer. In *ICSE '92: Proceedings of the 14th international
             conference on Software engineering*, pages 307–317, New York, NY, USA, 1992.
             ACM.

[DMS84]   Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz. Viewing a
           programming environment as a single tool. *SIGPLAN Not.*, 19(5):49–56, 1984.

[EC72]   Jay Earley and Paul Caizergues. A method for incrementally compiling lan-
           guages with nested statement structure. *Commun. ACM*, 15(12):1040–1044,
           1972.

[Ecl]   Eclipse.org. Explore the Eclipse Universe. http://www.eclipse.org/. (Date
           retrieved: October 1, 2009).

[Ecl03]   Eclipse Platform: Technical Overview, 2003.

[FF01]   Robert Bruce Findler and Matthias Felleisen. Contract Soundness for Object-
           Oriented Languages. In *OOPSLA*, pages 1–15, 2001.

[FGOG07]   Lorenz Froihofer, Gerhard Glos, Johannes Osrael, and Karl M. Göschka.
           Overview and Evaluation of Constraint Validation Approaches in Java. In
           ICSE, pages 313–322. IEEE Computer Society, 2007.

[FOG06]   Lorenz Froihofer, Johannes Osrael, and Karl M. Göschka. Trading Integrity
           for Availability by Means of Explicit Runtime Constraints. In *COMPSAC*,
           pages 14–17. IEEE Computer Society, 2006.

[Fri83a]   Peter Fritzson. A systematic approach to advanced debugging: incremental
           compilation. *SIGSOFT Softw. Eng. Notes*, 8(4):130–139, 1983.

[Fri83b]   Peter Fritzson. Symbolic debugging through incremental compilation in an
           integrated environment. *Journal of Systems and Software*, 3(4):285–294, 1983.

[GHJV95]   Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design pat-
           terns: elements of reusable object-oriented software.* Addison-Wesley Longman
           Publishing Co., Inc., Boston, MA, USA, 1995.

[GJSB05]   James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java^{TM} Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[HGM00]   James Hayes, William G. Griswold, and Stuart Moskovics. Component design of retargetable program analysis tools that reuse intermediate representations. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 356–365, New York, NY, USA, 2000. ACM.

[Hoa69]   C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[JUn]   JUnit.org. JUnit.org Resources for Test Driven Development. http://www.junit.org/. (Date retrieved: October 1, 2009).

[Kar98]   Michael Karasick. The architecture of montana: an open and extensible programming environment with an incremental C++ compiler. *SIGSOFT Softw. Eng. Notes*, 23(6):131–142, 1998.

[KCJG08]   George Karabotsos, Patrice Chalin, Perry R. James, and Leveda Giannas. Total Correctness of Recursive Functions using JML4 FSPV. In *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, number CS-TR-08-07 in Technical Report, pages 19–26, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362, 2008. School of EECS, UCF.

[Kra98a]   R. Kramer. iContract - the Java^{TM} Design by Contract^{TM} Tool. *Technology of Object-Oriented Languages, International Conference on*, 0:295, 1998.

[Kra98b]   Reto Kramer. iContract – the Java® Design by Contract® Tool. In *TOOLS 26: Technology of Object-Oriented Languages and Systems*, pages 295–307. IEEE Computer Society Press, August 1998.

[LBR99]   Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[LBR06]   Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[LC05]   Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. Draft, available from jmlspecs.org., 2005.

[LCC+05]   Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[Lea]   Gary   T.   Leavens.   The   Java   Modeling   Language. http://www.eecs.ucf.edu/ leavens/JML/.   (Date   retrieved: October   1, 2009).

[LG86]   Barbara Liskov and John Guttag. *Abstraction and specification in program development.* MIT Press, Cambridge, MA, USA, 1986.

[LPC+06]   Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*, May 2006. Draft revision 1.193.

[Mey88]   Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[Mey92]   Bertrand Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992.

[Par]   Parasoft.   Automatic Java Software and Component Testing: Using
            Jtest to Automate Unit Testing and Coding Standard Enforcement.
            http://www.parasoft.com/jsp/products/article.jsp?articleId=839&product=Jtest.
            (Date retrieved: October 1, 2009).

[Par79]   D. L. Parnas. On the criteria to be used in decomposing systems into modules.
            pages 139–150, 1979.

[Pay03]   Mary F Payne.   Automating instrumentation: Identifying instrumentation
            points for monitoring constraints at runtime, January 01 2003.

[PK07]   Mario Pukall and Martin Kuhlemann.   Characteristics of Runtime Pro-
            gram Evolution. In Walter Cazzola, Shigeru Chiba, Yvonne Coady, Stphane
            Ducasse, Gnter Kniesel, Manuel Oriol, and Gunter Saake, editors, *RAM-SE*,
            pages 51–58. Fakultät für Informatik, Universität Magdeburg, 2007.

[Rei84]   Steven P. Reiss. An approach to incremental compilation. In *SIGPLAN '84:
            Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages
            144–156, New York, NY, USA, 1984. ACM.

[Ros92]   David S. Rosenblum.   Towards a method of programming with assertions.
            In *ICSE '92: Proceedings of the 14th international conference on Software
            engineering*, pages 92–104, New York, NY, USA, 1992. ACM.

[SDB84]   Mayer D. Schwartz, Norman M. Delisle, and Vimal S. Begwani. Incremental
            compilation in Magpie. *SIGPLAN Not.*, 19(6):122–131, 1984.

[Sun05]   Sun. Java Tuning White Paper. December 2005.

[Tan94]   Yang Meng Tan. *Formal specification techniques for promoting software mod-
            ularity, enhancing documentation, and testing specifications.* PhD thesis, Mas-
            sachusetts Institute of Technology, Cambridge, MA, USA, 1994.

[TE03]   Roy Patrick Tan and Stephen H. Edwards. An Assertion Checking Wrapper Design for Java, August 15 2003.

[WK97]   Jürgen F. H. Winkler and Stefan Kauer. Proving assertions is also useful. *SIGPLAN Not.*, 32(3):38–41, 1997.

[WM05]   Qianxiang Wang and Aditya P. Mathur. Interceptor Based Constraint Violation Detection. In *ECBS*, pages 457–464. IEEE Computer Society, 2005.

[YB94]   Hwei Yin and James M. Bieman. Improving Software Testability with Assertion Insertion. In *Proceedings of the IEEE International Test Conference on TEST: The Next 25 Years*, pages 831–839, Washington, DC, USA, 1994. IEEE Computer Society.

Appendix A

Compilation Phases Overview of the Eclipse Platform

The main steps of the compilation process performed by JDT are illustrated in Figure A.1. In the Eclipse JDT (and also in JML4), there are two types of parsing: in addition to a standard full parse, there is 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, the corresponding source (*.java) file is searched. 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.

The most important components of the JDT compilation phases are:

1. *Scanning.* Scanning of source code is done at a much later stage. 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 or bounded to the CU.

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.

3. *Type Checking.* Type checking is performed by invoking the `resolve` method on a compilation unit.

4. *Flow Analysis.* Flow analysis is performed by the `analyseCode` method on a compilation unit.
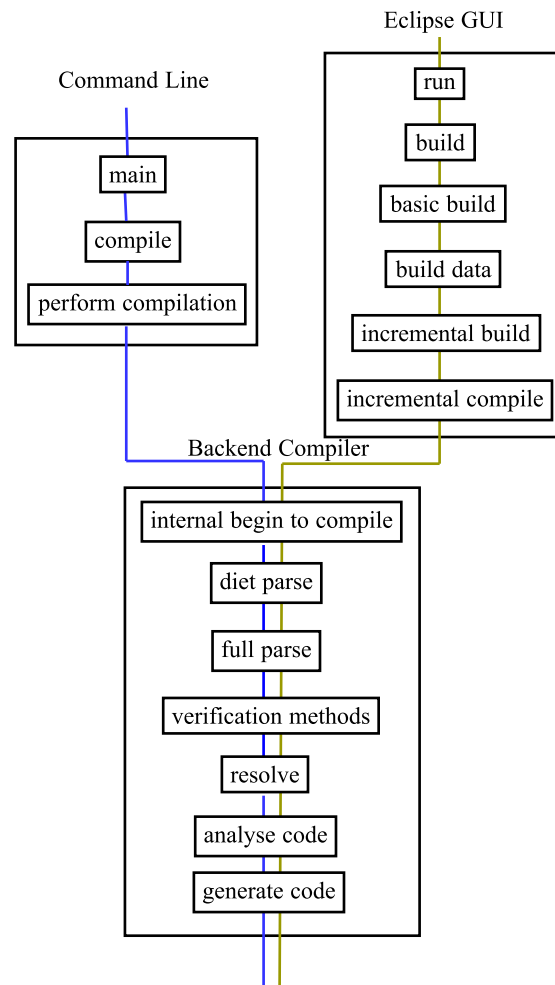

Figure A.1: Interaction between command-line tools, GUI and the Eclipse Java compiler

Contrary to popular belief, the Eclipse framework has just one back-end compiler support. Here we discuss the overall interaction between the command-line tools, GUI and the

compiler. Figure A.1 illustrates the interactivity between the different APIs. The command line API (at an abstraction level) contains 3 methods. 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, which initializes the Batch.Compiler to its default settings and passes the Compilation Units denoted as CUs to `internalBeginToCompile` method, which is the starting point of the compiler API. In GUI case, on invoking the Eclipse framework, several threads concurrently start, 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 or 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 that identifies which CU is to be built. And then sends this to the `jdt.compiler` class.

Appendix B

Testing on the Eclipse Platform

B.1 Experimental Setup

All the experiments and the results that are tabulated for this thesis has been carried out on a Dell Optiplex 760 Intel $^{®}$ Core ^{TM} 2 Duo CPU E7400 @ 2.8 GHz with 3.25 GB memory, running Microsoft Windows XP Professional Version 2002 SP3. All the tests were executed using Java hotspot Client VM version 1.6.0-13-b03.

The tests were timed using the system clock. The compilation time was computed as the difference between the start and stop time.

B.2 Compilers Used

javac: javac is the standard compiler in Sun JDK and also serves as a reference implementation for the Java Programming Language. The version used was 1.6.0.05.

jmlc: jmlc is the standard compiler for Java Modeling Language. The version used was JML2 version 5.6 RC4.

Eclipse Java Compiler: The Eclipse SDK that was used for testing purposes was version number 3.4.1.

Jml4c: The new JML compiler that has been built on top of the Eclipse Java compiler versioned 3.4.1 v-874.

B.3 Testing the JML Compiler inside Eclipse

Unit testing is a labor-intensive activity, hence it is often not done as an integral part of programming. However, it is a practical approach to increasing the correctness and quality of software; for example, the Extreme Programming approach [Bec99] relies on frequent unit testing. This section demonstrates how by using JML and the JUnit testing framework we automated the writing of unit test oracles. Thus, we show that the compiler so designed can be used seamlessly in Eclipse and can be used to compile "real" programs.

We used JUnit [JUn] a popular framework that automates some of the details of running tests. It is a simple yet practical testing framework for Java classes; it encourages the close integration of testing with development by allowing a test suite be built incrementally.

B.3.1 Testing Framework

The Eclipse framework supports JUnit testing framework. In essence, all the 35K test cases for Java or Eclipse compiler have been written around the JUnit framework. The main class from which all the different test classes inherit their methods is the `AbstractRegression Test` class. However for our purposes, we created a new test class that inherits the abstract class and overrides some of the methods, namely `runConformTest`. Creating this new test class served several purposes, namely:

1. For testing purposes, we required to change some of the default compiler options (at times dynamically *).

2. Adding `/org.eclipse.jdt.core/bin` and `/org.jmlspecs.annotation/bin` to the default class path.

3. We needed to override methods in the abstract class for testing the JML compiler such that the test code was successfully compiled by the new JML compiler integrated into Eclipse instead of using the default Eclipse compiler.

---

*For checking and testing non-null type systems.

4. Added a new method namely `runConformTestThrowingError` which helped to evaluate automatically that whether the thrown error is of the expected type.

5. Added new methods, `compileAndRun`, which made it easier for the users to write similar test cases. For most test cases, the body for `public static void main(String args[])` remained the same. We factored this out and made users only put the test code without the main method. This was automatically added into the test code later during the execution. An example of such a test case is shown in Figure B.1. In the figure, it is shown that the test code is embedded as a parameter to the method `compileAndRun`. The method shown contains three parameters: name of the class that is tested, the test code, method body of the main function. Another variation is to have another parameter for the method that gives the expected error that is to be thrown.

```
1  public void test_synchronized_statement(){
2    compileAndRun(
3      "X.java",
4      "public class X {\n" +
5      " public void m() {\n" +
6      "   synchronized(this) {\n" +
7      "     int i = -1;\n" +
8      "   }\n" +
9      " }\n" +
10     "}\n",
11     "new X().m()");
12 }
```

Figure B.1: An example of a test case being tested using JUnit framework inside Eclipse

B.3.2 Deciding Test Outcomes

A test case can be thought of a three-tuple containing the following form:

$\mathcal{Q} = (\mathcal{C}, \mathcal{E}, \mathcal{A})$ where

$\mathcal{Q}$ = the test outcome,

$\mathcal{C}$ = the test code that is tested,

$\mathcal{E}$ = expected output or error,

$\mathcal{A}$ = actual output or error.

In our framework, a test is assumed to be successful if $\mathcal{E} = \mathcal{A}$; otherwise, the test is assumed to have failed. More specifically, if the call to the test code terminates normally, i.e., no exception is thrown, then the test succeeds. Similarly, if the call results in an exception that is not an assertion violation error, then the test succeeds *iff* the error was expected. With JUnit, however, such an exception must be caught by the test method, because all exceptions are interpreted by the JUnit framework as signaling test failures; we required to change this behavior. If the test code throws an assertion violation error, we require to find whether the assertion violation error is of the expected error given by $\mathcal{E}$.

B.3.3 Executing the Test Cases

A test suite namely `RunJMLTestCases` which runs all the test cases together was created. The suite method is like a main method that is specialized to run tests. In order to run all the test cases, it was enough to run this method; this would internally call and execute all the test classes. Figure B.2 shows a screen shot of how a failure and a successful test run looks. Notice that the failure trace shows why the test run failed.
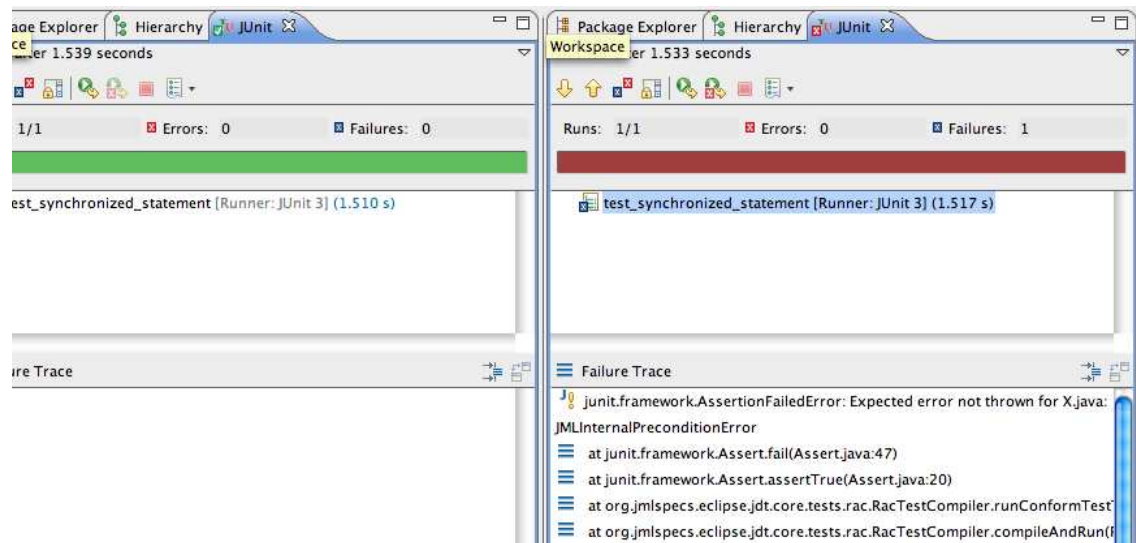
Figure B.2: Screenshot of a successful and a failed test run.

Appendix C

Front-End support for JML on the Eclipse Platform

In this chapter, I present very briefly the most important parts of the processes that are involved for front-end support for JML on the Eclipse platform. This chapter explains the key concepts behind grammar file, JikesPG, the parser and scanner source code. Most of the front-end support towards JML has been largely carried out by Patrice Chalin and his team [CJK08b] of Concordia University, Canada, Robby and his team from Kansas University and others from the JML group.

C.1 Introduction

A very important principle in the development process is to make minimal changes to the source code from Eclipse, to allow for convenient and easy merging between Eclipse and our project. Currently almost all JML Level 0 and 1 features has been supported by the front-end.

C.2 Grammar Files

The grammar file (`java.g`) is located in `org.eclipse.jdt.core package`. It is used as an input to an LALR parser to generate the corresponding code. The changes that are made to the grammar file are all contained in between lines `<jml-startid=''jml.feature-name" />` and `<jml-end id=''jml.feature-name" />`.

The grammar file (java.g) has 7 different sections. They are:

1. Options - Contains options or flags that is used for parsing by the JikePG generator.

2. Define - Translation rule for common macros that are used through out the grammar file like `/.\$putCase`

3. Terminals - Identifiers or keywords that are terminals.

4. Alias - Common aliases used for writing production rules.

5. Start - Points to the start from where the generator starts generating code for the parser.

6. Rules - Production rules.

7. Names - Readable names that is used in error messages.

In the grammar, each production rule is followed by one or two additional lines. Every production rule contains a semantic action and a readable name for a nonterminal. The semantic actions are required for productions that uses `::=` and looks like `/.\$putCaseconsumeXXX();\$break ./`. The method body in `consumeXXX()` does the associated semantic action. And the second line looks like `/:\$readableName XXX:/` which is used for error messages. If the rule does not contain any semantic action, then `->` is used instead of `::=`.

The new JML specific keywords are added to the `java.g` grammar file in the `$Terminals` section.

C.3   JikesPG Generator

The parser files and resources are automatically generated using the Jikespg parser generator. For the JML front-end support, a slightly customized version of the JikesPG is required. Here are the instructions for customizing it: Under `src/common.h` replace

`#define PRINT_LINE_SIZE 80` with `#define PRINT_LINE_SIZE 300`. More instructions can be found at http://sourceforge.net/apps/trac/jmlspecs/wiki/JML4

The input to the parser generator is a grammar file that is LALR(1).

C.4   Parser File

Different types of stacks are maintained by the parser. The more important ones deal with identifiers, expressions, and ASTs. There's also an `intStack` for token positions. The identifier stacks store particular names and keywords. It contains three parts which is used to store the position of the name or keyword. An AST for an expression will be contained inside an expression stack whereas the statement as a whole gets pushed into the AST stack. Terminal symbols by convention are enclosed inside single quotes, like 'requires'.

C.5   Scanner File

Keywords are added onto the static initializer for the Map `JML_KEYWORD_TO_TOKEN_ID_MAP` such that the scanner is able to recognize these names as keywords and interpret them differently. The keywords that are added in scanner, are also added in `Parser.consumeToken(int type)`.

C.6   Type Checker and Flow Analyser

Type checking is performed by invoking the `resolve` method on a compilation unit. Similarly, flow analysis is performed by the `analyseCode` method.

C.7 Merging External Specification

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.

Appendix D

Separate Compilation in the Current JML Compiler

Here we discuss the problem of separate compilation of jmlc and a possible solution to it. This is an important problem to solve as it amounts to almost 40% of the compilation time. Java is a strongly-typed language, i.e., at compile-time it ensures that all assignments or method calls are type-compatible. The necessity for referencing dependent files is to gather type information. In Java, all type information is stored inside the byte-code. Hence if the byte-code is current then necessary information can be gathered from the byte-code itself. However in jmlc, this may or may not be possible. JML provides abstract specifications i.e, to write specifications with abstract values instead of directly using Java variables and data structures. There are several advantages of using abstract specifications. By using abstract values, the specification does not have to be changed when the particular data structure used in the program is changed [LG86] [Mey88] [Mey92] [Par79]. Also, it allows the specification to be written even when there are no implementation data structures. Abstract specifications in JML is written using the JML modifier `model`. JML allows declaration of model methods, fields, and even types.

Figure D.1 shows a sample snippet using model fields. A model field can be thought to be an abstraction of a set of concrete fields. A `represents` clause is used to specify the connection between the concrete fields and such model fields. Since model fields are specification-only fields, they are translated by the JML compiler into access methods [CLSE05] [CL02]. Several scenarios are possible to reason about the behavior of jmlc's separate compilation. The java file `I1` may be pre-compiled by javac and `C1` is required to

be compiled by JML compiler. In this case, since `val1` is a model field, the information is not present in the `I1`'s byte-code and hence it is to be re-compiled by JML compiler to gather the type information of `val1` from `I1`. More generally, since on compiling `I1`, the type information of the model field `val1` cannot be gathered directly from the byte-code, it is always re-compiled.

```java
1  // I1.java
2  public interface I1 {
3    //@ public model instance int val1;
4
5    //@ requires val1 > 0;
6    void m1();
7  }
8
9  // C1.java
10 public class C1 implements I1 {
11   public int x = 1;
12   //@ public represents val1 <- x;
13
14   //@ requires val1 < x + 5;
15   void m1() { /* ... */ }
16 }
```

Figure D.1: Example of model fields in specifications

D.1 A Possible Solution

For correct compilation the JML compiler requires to recompile the referenced source code. Here we propose four variants which would solve the problem of separate compilation. They are:

Encoding type signature inside .class file by extending byte-code format

The `.class` file generated using jmlc does not contain any information about type signature. So an immediate solution to this problem is, encoding this type signature information into the byte-code. In order to do so, we require to extend the existing byte-code format so that this type-checked information can be encoded into it. However, we should also make note that by extending the byte-code format, we would do no harm to the actual execution of `.class` file. Since `javac` would ignore the extended format as it does for the annotations.

Writing type signature into a separate symbol file

Another alternative is to write the runtime specification into a separate symbol file. This would free ourselves from extending the current byte-code format. In the type-checking phase rather than checking source code in the class-path, we would require to find a specific file. This file would be different for different source files. Hence we require a naming convention, say the keyword `symbol` prefixed with the filename itself (for which the type signature information is being generated) Eg. for a filename `Simple.java` we would have `Simple$Symbol.symb`. Hence we would have two files being generated after the compilation process instead of one: the original `Simple.class` and `Simple$Symbol.symb`.

Writing type signature into a separate Textual file

The third alternative is to write the runtime specification into a separate ascii or readable formatted file (xml files can also be used). This may be similar to header files as in C or C++. This has a distinct advantage over jmlc where the referenced source code is entirely scanned, parsed, and type-checked to extract the type signature information. In this alternative only the runtime information is to be scanned and parsed. They are in human readable format. The xml format may be better in parsing the file to extract runtime information. This method is slow compared to the other two proposed solutions because in this case we require to recompile this ascii file. However this approach is far better than

the existing approach.

Extracting type signature from access methods

Since every model field is translated into access methods having distinct method header names, it may be possible to extract information of type signature from these access methods. For example, a model field, `i` of type `T` defined in `C` has a method header `public T model$i$C`. It may be noted that the method header contains the simple name of the model field `i`. It is possible to access the field `i` as in `C.i` (if `i` is a static field), or by using `this.i`. In either case, we may require to convert qualified names to simple names and then search the corresponding method. A disadvantage of this approach is that it is an ad-hoc procedure.

Curriculum Vitae

Amritam Sarcar was born on April 26, 1985. The second son of Prof. Ashish Kumar Sarcar and Jayashree Sarcar, he graduated from St. Xavier's Collegiate School, India, in the spring of 2003. He entered West Bengal University of Technology in the fall of 2003, and, in the spring of 2007, he graduated with a bachelor's degree in Computer Science and Engineering. In fall 2007 he joined ITC Infotech Pvt. Ltd. as an Associate IT Consultant and continued to work there till spring 2008. In the spring of 2008, after having a short experience in the software industry, Amritam resumed his studies at The University of Texas at El Paso for his master's degree. While pursuing his master's degree in Computer Science he worked as a Research Assistant under Dr. Yoonsik Cheon and as a Teaching Assistant for the Computer Science department. He was an active member of the Software Specification and Verification Research Laboratory.

Permanent address: 42/132 New Ballygunge Road

Kolkata - 700039, West Bengal, India