Checking JML Specification Soundness Using ESC/Java2

Final Year Project Final Report

Conor Gallagher

A thesis submitted in part fulfilment of the degree of BSc. (Hons.) in Computer Science with the supervision of Dr. Joseph Kiniry and moderated by Dr. Damien Dalton.

Department of Computer Science

University College Dublin

13 March 2005

Abstract

This report describes the work completed in the Final Year Project "Checking JML Specification Soundness Using ESC/Java2." It introduces some of the basics of Design by Contract and the Java Modelling Language (JML). This paper primarily deals with the Extended Static Checker ESC/Java2, one of the many powerful tools that understand JML. It describes techniques for extending the functionality of ESC/Java2, to allow for the location of specification inconsistencies. This paper gives an overview of ESC/Java2 architecture and the design of a solution is presented along with some possibilities for future improvements.

Table of Contents

1 Introduction

A major component of quality in software is reliability: a system's ability to perform its job according to the specification (*correctness*) and to handle abnormal situations (*robustness*). Put more simply, reliability is the absence of bugs[1]. As society's dependence on software products increases the consequences of failures becomes more and more costly. The Ariane launcher[2] disaster providing a fine example of this. Reliability cannot however be achieved until each component in a software system can be trusted to live up to its specifications.

Design by Contract (DBC) allows the developer to explicitly state the behavioural specifications for each component in a software system through the use of contracts. The principle idea behind DBC is that a class and its clients have a contract with each other. The client must guarantee certain conditions before calling a method from the class (the preconditions) and the class must guarantee certain properties after the call (the postconditions)[3]. JML (Java Modeling Language) is a behavioural specification language for Java similar to the DBC approach seen in Eiffel.

Using assertions contained in comments similar to that of Javadoc, JML is used to specify the behaviour of Java code. JML can, for example, specify the behaviour of Java code through the use of preconditions, postconditions and class invariants. These specifications are written in special annotation comments, starting with an at-sign (@), see Figure 1 below.

```
//@ requires err != null;
//@ ensures correction != null;
public String deleteString(String err)
{ /* Code to delete erroneous word*/
   return correction;}
```

Figure 1. Simple JML example

The above example shows both a precondition and a postcondition. The precondition requires that the string being passed to the method is non-null and the postcondition ensures the returned string is non-null.

ESC/Java2, the world's most advanced extended static checker, is one of the many powerful research tools that understand JML. It checks that a Java program conforms to the formal specifications, supplied by the developer in JML annotations. ESC/Java2 accomplishes this by combining a rich object logic, a weakest precondition calculus for Java and JML, and “Simplify” an automated first-order theorem prover.

For newcomers to behavioural specification languages, one of the most difficult things to come to terms with is pinpointing the cause of specification inconsistencies. Contradicting preconditions result in a flawed logic for the method, see Figure 2 below.

```
//@ requires err != null;
//@ requires err == null;
//@ ensures correction != null;
public String deleteString(String err)
{ /* Code to delete erroneous word*/
   return correction;}
```

Figure 2. Simple specification error

The example above contains 2 contradicting preconditions, requiring that "err" be both null and non-null. This flawed logic results in automatic verification by the theorem prover regardless of other predicates.

One technique for pinpointing the inconsistencies in a specification involves systematically rechecking using ESC/Java2 removing one of the JML annotations for each recheck. While this technique is very labour intensive, it follows regular patterns and can thus become an automated process. The objective of this project is to study the different patterns involved in determining specification soundness errors and ultimately to extend the functionality of ESC/Java2, automating the process for pinpointing these inconsistencies.

The preparatory work needed before undertaking such a task is quite extensive and the following section, Chapter 2, goes through some of the background reading and initial difficulties encountered. There exists an enormous wealth of information on the Internet for getting started with behavioural specification languages (in particular JML), some of which proved invaluable at the beginning of this project.

Chapter 3 describes the step-by-step approach taken in learning ESC/Java2's architecture, through hands on experience with the Universe type system. The design of test cases and lessons learned from use of profiling and debugging techniques are also documented in this chapter.

Chapter 4 deals with the design aspects of this project and the range of alternative solutions considered.

Chapter 5 describes the implementation of the design, its integration within ESC/Java2 architecture and some of the problems encountered.

Chapter 6 covers the conclusions drawn from working with ESC/Java2 and thoughts on future work in the field are discussed.

2 Background Research

The background research that took place is distinguished into three separate categories. The specific tools and operating system needed are discussed in subsection 2.1. JML will be described in detail in subsection 2.2. The final section in this chapter, subsection 2.3 relates to the static checker ESC/Java2, on which the project is based.

2.1 The Operating System and Tools

The operating system chosen to host the design of this project was Linux, more specifically "Red Hat's" Fedora distribution. Although ultimately this operating system proved to be the most suitable, the learning curve for a newcomer is quite steep. The principle source of information used in determining the installation type and requirements for configuring a dual boot environment was "Red Hat Fedora 2"[4] and direct feedback from the project supervisor[5]. Websites such as LinuxQuestions.org[6] and the Red Hat Fedora website[7] provide a vast amount of information and support relevant to configuring the operating system.

GNU Emacs was the principle tool used to edit, develop, compile, and test ESC/Java2. The tutorial accompanying the editor gives the user a basic understanding of simple editing commands, but more complex commands for editing Java code or viewing Texinfo documentation websites such as GNU.org[8] and emacswiki.org[9] were necessary.

The complexity of ESC/Java2 demanded the use of both a debugger and a profiler in order to become familiar with the architecture. The “Jswat”[10] debugger was suitable as it provides standalone, graphical Java debugging. Profiling of ESC/Java2 was carried out using the “Jprofiler”[11] tool. “Jprofiler” allows, among other things, a snapshot of the heap to be taken in order to gain an understanding of what classes are called and what objects are formed under different scenarios.

2.2 The Java Modeling Language

JML, as mentioned earlier, is a specification language for Java that supports the DBC methodology. During the background stage of this project, the following resources proved to be the most valuable with regards to gaining experience with the Java Modeling Language:

- An Overview of JML Tools and Applications[12]
- JML: Notations and tools supporting detailed design in Java[13]
- Design by Contract with JML[14]
- JML Reference Manual[15]
- Modular Invariants for Object Structures[16]

2.2.1 An Overview Of JML Tools and Applications

This was the first paper I used for my background work. It is worth a few notes as it gives some very good definitions and proved an excellent introduction to JML. It describes in detail some of JML’s design goals, and moves from there, to explain the JML notation, followed by a summary of the tools available.

JML was designed to stay as close to the Java syntax as possible in order to be easily understood by Java programmers. It is this detail that makes the transition for programmers to JML relatively painless. Figure 2 below, taken directly from this paper[8] illustrates the main features of JML. Assertions are written inside special comments in the Java code, after //@ or between /*@ … */. The Java complier will ignore these annotations, but they are recognised by special JML tools.

The essential ingredients of DBC: the preconditions (given in “requires” clauses), the postconditions (given in “ensures” clauses) and class invariants, can all be noted in this example. A precondition is an assertion that must be true before calling the method (Figure 2, the “amount” must be at least zero). A postcondition is an assertion that the method guarantees will hold true after the call (Figure 2, the new balance will be the old balance minus the amount debited, or there will be an exception and it will remain unchanged). An invariant is a property that will hold true in all visible states (Figure 2, “balance” will always be at least zero and at most the maximum possible balance). The invariant for the byte array “pin” shows the similarity in notation between JML and Java. This invariant states that the "pin" must be four digits long, with each digit between 0 and 9.

```
public class Purse {

final int MAX_BALANCE;
int balance;
//@ invariant 0 <= balance && balance <= MAX_BALANCE;

byte[] pin;
/*@ invariant pin != null && pin.length == 4 &&
  @                    (\forall int i; 0 <= i && i < 4
  @                                 ; 0 <= pin[i] && pin[i] <= 9);
  @*/

/*@ requires     amount >= 0;
  @ assignable   balance;
  @ ensures      balance == \old(balance) – amount &&
  @              \result == balance;
  @ signals      (PurseException) balance == \old(balance);
  @*/
int debit(int amount) throws PurseException
{ … }
```

Figure 3. Example JML Specification from [8]

It should be pointed out here how difficult it would be for a novice to locate a specification error in the above example compared to Figure 2. The labour intensive technique of removing each assertion one at a time and rechecking is far more suited to be included as a functional option of the ESC/Java2 static checker.

2.2.2 JML: Notations and tools supporting detailed design in Java

This brief paper gives a short description of JML and some of the tools available. It is useful as a stepping-stone towards more in-depth papers and provides an idea of some helpful tools and their uses. A brief description of ESC/Java2's predecessor, "DEC/Compaq SRC ESC/Java" is informative but it should be noted that ESC/Java2 is somewhat functionally different.

2.2.3 Design by Contract with JML

"Design by Contract with JML" gives a tutorial-style introduction to JML and explains how it can be used as a powerful DBC tool for Java. DBC is credited as a way of recording details of method responsibilities, thus helping to avoid rechecking the validity of arguments in a method body. For example, if an array is specified as ordered, it can be assumed true and will not require actively revalidating this fact through code. Recorded details such as this make DBC an excellent way of documenting design decisions.

A clear point of interest in this paper is the explanation of JML for blame assessment. By examining which of the specifications is broken the faulty party can be determined. A broken precondition implies the erroneous code occurs within the caller of a method (the client), and a broken postcondition implies the fault lies within the implementation of the method. It should be noted however that this relates to errors contained within the **code** and that this project is primarily interested in **specification** errors. DBC is also shown to be a way of increasing code efficiency by eliminating the need for defensive checks at runtime (i.e., checking to ensure an array is ordered or something is non-null). Once all contracts are satisfied in debugging, defensive checks are no longer necessary and contracts can be either automatically removed or ignored when running.

Section 2 of “Design by Contract with JML” describes how JML facilitates the use of a wide range of tools for runtime assertion checking, discovery of invariants, static checking, specification browsing, and even formal verification using theorem provers. Needless to say ESC/Java2 is listed as one of these many JML-aware tools. Examples of different techniques for writing DBC using JML are provided along with the following list of tools useful for DBC, through JML:

- The JML compiler “jmlc.”
- The unit testing tool “jmlunit”[17].
- The documentation generator “jmldoc.”
- The extended static checker ESC/Java2.
- The type checker “jml”, a fast substitute for “jmlc”, if compilation is not needed. (ESC/Java2 may also be used as a faster and more capable type checker than “jml”)

A final aspect worthy of noting is that the bytecode “jmlc” generates includes both the Java and assertion checking code.

2.2.4 JML Reference Manual

Probably the most useful of all the JML resources for this project is the 143 page “JML Reference Manual” that provides detailed documentation on almost every aspect of JML. For example, during the task of adding parsing capabilities for new JML keywords to ESC/Java2 (see Section 3), this manual helped provide details of where keywords belonged in the grammar. It has been constantly referenced whilst:

- examining existing ESC/Java2 implementation for other JML keywords.
- trying to build an understanding of JML’s grammar structure and capabilities.

2.2.5 Modular Invariants for Object Structures

This paper provides a detailed explanation of the classical specification approach for invariants over complex object structures and problems with this approach. Although this aspect of JML does not directly relate to the objectives of this project reading, this paper was necessary in order to gain an understanding of the Universe type system. The paper provides details for generalizing classical reasoning techniques in order to provide invariants for object structures without losing modularity.

In adding new parsing capabilities for the Universe type system to ESC/Java2 (see Chapter 3) this paper helped clarify where these new keywords belonged within the JML grammar. The type system necessitates the introduction of a few new keywords used in JML type annotations. In other words, the new keywords would only be permitted to specify an object’s type and could thus only appear in specifications similar to that in Figure 4. Notice that the “rep” keyword indicates that this array object is part of an encapsulated internal representation of the list[16].

```
Public List() {
   array = new /*% rep %*/ int [5];
}
```

Figure 4. Example of Universe type system

2.3 ESC/Java2

In order to get familiar with the use, functionality, and architecture of ESC/Java2 a number of different resources were used. The following subsections give a brief description of some of the resources I found useful and what information they contain.

2.3.1 JML and ESC/Java Homework Exercises

This paper[18] provides a range of exercise built upon one another, designed to teach the reader the basics for writing JML and making the most of JML tools.

2.3.2 ESC/Java2 Implementation Notes

These notes are part of the "ESCTools" package (this package will be discussed in Chapter 3). They document the design and implementation of ESC/Java2. Although quite extensive and overwhelming at first, these notes provide detailed descriptions of all different aspects of this system. The ESC/Java2 implementation notes have become a very important referencing tool whilst trying to maintain the high implementation standards ESC/Java2 boasts.

2.3.3 ESC/Java Design Notes

The design notes for the original ESC/Java (located in the "ESCTools" package) document the development of ESC/Java from the design phase. Although incomplete and often no longer relevant to the ESC/Java2 design, the 32 documents contained within this package provide a valuable resource for this project. Of particular significance is "ESCJ 16c: Java to Guarded Commands translation"[19] as it describes the translating of annotated Java code into a guarded command-like language for the purpose of generating verification conditions.

Developing an understanding of how this translation occurs was a major milestone toward completion of this project. In order to remove a single annotation and rerun ESC/Java2 it is necessary to be able to manipulate these guarded commands. This is a non-trivial task and will be discussed in more detail in Chapters 4.

2.3.4 Compiler and Parser

ESC/Java2's parser reads the JML annotated Java source code and outputs an Abstract Syntax Tree (AST). In order to make the processing of the AST easier, it does not provide an exact mirror of the source code, syntactic elements such as parentheses are discarded. "A Compiler Front End"[20] provides an introduction to AST's helpful for this project.

ESC/Java2 makes use of an AST generator that automatically generates boilerplate code for the different classes needed. "ESCj 24: Astgen Manual"[21] describes how this tool reads in a file containing annotated, partial implementations of AST classes and writes full implementations for those classes.

The generator technique leads to a manageable description of AST classes being located in the AST hierarchy file. The generator technique also allows one to easily change an AST hierarchy and the code found inside of AST classes[21], for example, to add parsing of the Universe type system keywords(see Chapter 3).

3 Approach

3.1 Initial Approach

The following is a task list for starting out with this project:

- gain familiarity with the Linux operating system
- become comfortable working and editing in GNU Emacs
- study, practise using, and gain an understanding of JML
- gain in-depth understanding of ESC/Java2's architecture and functionality

The first few weeks working on the project were spent conducting much of the background reading detailed in Chapter 2. This provided a firm foundation with which to begin the project, and allowed for the installs necessary to run JML and ESC/Java2. Once these tools were operational, the source for the current release of ESC/Java2 was obtained. The package is named "ESCTools" and contains the entire architecture which must be compiled to the full latest version of ESC/Java2. The viewing and editing of all the source code occurred in GNU Emacs and Netbeans. The following subsection briefly describes the source release of ESC/Java2 as contained within the "ESCTools" package.

3.1.1 ESCTools

The "ESCTools" package is divided into 5 sub-packages, which will be referred to as envelopes in this report. These envelopes are:

- "Javafe" - The Java front end that both ESC/Java2 and RCC use. The source code in this envelope handles the parsing and type checking of Java code. Subclasses of the java front end have the ability to extend Java with annotations.
- "Escjava" - Contains the ESC/Java2 extensions of the front end. JML is added as an extension to Java.
- "Simplify" – Contains the sources for the theorem prover Simplify. Simplify is the prover used by ESC/Java2 and Houdini.
- "Rcc" – Sources for a race condition checker for Java. (Not relevant to this project)
- "Houdini" – Contains an annotation assistant for Java. (Not relevant to this project)

In order to gain an understanding of the ESC/Java2 architecture only the "Javafe" and "Escjava" envelopes need be examined. Although the theorem prover Simplify is used to verify the JML annotated Java code, knowledge of its implementation is not necessary. Similarly the "RCC" and "Houdini" envelopes may also be ignored.

3.1.2 Parsing Ability for Universe Types

After the preparatory stage a small project was chosen to improve upon my understanding of the ESC/Java2 architecture. The assigned project was related to the Universe type system, realized through an extension of the JML language with the addition of three new keywords: “\peer”, “\readonly”, and “\rep” (All prefixed with backslashes to indicate they are JML keywords and not Java keywords or identifiers).

The project entailed providing ESC/Java2 with parsing capability for these new keywords. Providing ESC/Java2 with parse and ignore ability for new keywords requires navigating through the architecture and making minor additions to the code. The difficulty of this process is in locating where exactly the additions belong. In most cases classes only require the addition of three or four new lines of code.

As mentioned in Chapter 2, ESC/Java2 makes use of an AST generator. The addition of annotated, partial implementations of AST classes in the hierarchy file results in full implementations for those classes. This was one of the techniques utilized in adding the three new keywords to ESC/Java2.

Although difficult, figuring out where these changes belong is an excellent way of learning the ESC/Java2 architecture. The experience gained with: ESC/Java2's AST generator, the assigning of tags for new keywords, and simple navigating around ESC/Java2's architecture, proved a big milestone toward completion of this project.

3.2 Test Cases

As a reminder, the objective of this project is to study the different patterns involved in determining specification soundness errors and ultimately to extend the functionality of ESC/Java2, automating the process for pinpointing these inconsistencies. With a sufficient understanding of JML, ESC/Java2's architecture and the overall aim of this project, we are now in a position to start formulating a solution.

Similar to the Extreme Programming approach of software design, the first task was to write a large amount of test classes, containing specification errors. The aim of these tests was to cover all of the possible inconsistency types for all of the Java types.

The test classes were produced with the design goal of deluding ESC/Java2 into verification by means of unsound specifications. The tests were divided into sets for the following Java types:

- constructors with no parameters, one parameter and multiple parameters.
- void return methods with no parameters, one parameter and multiple parameters.
- non-void return methods with no parameters, one parameter and multiple parameters.
- object return methods with no parameters, one parameter and multiple parameters.

The above test sets were again divided into four different categories, one each for precondition, postcondition, class invariant inconsistencies and another set of tests containing a combination of all three. Figure 5 shows precondition specification inconsistencies on both a constructor and a method, each taking in multiple parameters. The erroneous specifications in Figure 5 result in the generation of a false Verification Condition (VC), and thus instant verification by the theorem prover.

```
//Constructor
//@ requires o != null;
//@ requires d > 10;
//@ requires d < 10;
RequiresMultiParam(Object o, double d) {
    integer = 0;
    //@ assert false;
}

//Non-Void Method Returning a Non-Object
//@ requires f != 3.14159;
//@ requires i !=0;
//@ requires b > 2;
//@ requires b < 2;
double nonObject(int i, float f, byte b){
    double disk;
    return disk;
    //@ assert false;

}
```

Figure 5. Tests for Preconditions with multiple Parameters

Tests for specification inconsistencies in both preconditions and class invariants may be implemented by simply introducing a contradicting annotation. Postconditions inconsistencies proved a little more difficult to implement, the VC generated is not tested until after method execution, resulting in detection by ESC/Java2.

Specification inconsistencies in postconditions were found to go undetected only when called from within another method. Figure 6 demonstrates this with more clarity.

```
double d;
//Non-Void Method Returning a Non-Object
//@ ensures d != 3.14159;
//@ ensures d == 3.14159;
double nonObjectHelper(byte b){
    return d;
    //@ assert false;
}

//This method calls nonObjectHelper
double nonObject(int i, float f, byte b){
    /*The postconditions for nonObjectHelper are
    * regarded as preconditions for this method.
    */
    double disk = nonObjectHelper(b);
    return disk;
    //@ assert false;
}
```

Figure 6. Postcondition Specification Inconsistency

The postconditions of “nonObjectHelper” are received as preconditions for the call of “nonObject.” The faulty specifications will result in ESC/Java2 verifying the “nonObject” method but will issue a “postcondition possibly not established” warning for the “nonObjectHelper” method.

As mentioned in Chapters 1 and 2, the technique for pinpointing specification inconsistencies involves rechecking with ESC/Java2 iteratively eliminating suspect predicates. It is evident that this labour intensive technique is suitable for automation from the regular patterns it follows.

3.3 Conclusion

In this chapter we covered the initial approach taken with this project. The material covered in this chapter and the previous chapter constitutes the majority of work involved in this project. This is due to the vast knowledge base that is a prerequisite in order to attain a full understanding of ESC/Java2's implementation. Extensive knowledge of Java programming language couple with a firm understanding of:

- parsing and type checking techniques
- formal specifications
- verification
- both predicate and weakest precondition logic

are essential to successful completion of this project. With this knowledge base, coupled with the large test set, the design of a solution could begin.

4 Formulating a Design

The objective of this project is to extend the functionality of ESC/Java2, automating the process for pinpointing specification inconsistencies. The difficulty however, in extending the functionality of an intricate system such as ESC/Java2, is in locating a position within the existing design for the extension. Once in place, the functionality of ESC/Java2 should remain unchanged unless the user by means of a command-line option requests the extension.

As mentioned in the previous chapters, the technique for pinpointing specification inconsistencies involves rechecking with ESC/Java2 iteratively eliminating suspect predicates. The position chosen to implement this extension must thus allow for the looping ESC/Java2 execution. This is to allow for automatic rechecking with the different specifications.

This chapter explores the different stages of ESC/Java2's execution and considers different locations for the proposed extension.

4.1 Stages of Execution

“Main.java,” located within the “Escjava” envelope, controls ESC/Java2’s stages of execution. Finding a location to implement the extension involved thoroughly examining each of these stages. There are 6 stages of execution, these are:

- Stage 1: Loading, parsing and type checking
- Stage 2: Generating the type specific background predicate
- Stage 3: Translation of Java to the Guarded Command (GC) language
- Stage 4: Optional converting of GC to DSA
- Stage 5: Generation of the Verification Condition (VC)
- Stage 6: Calling the theorem prover (Simplify) to satisfy the VC

Fortunately, ESC/Java2 has a large number of command line options for different features of the tool, included is options for examining the results of each stage. The stages relevant to this project are discussed in the following sections.

4.2 Loading Parsing and Type Checking

As mentioned in Chapter 2, ESC/Java2's parser reads the JML annotated Java source code and outputs an Abstract Syntax Tree (AST). This AST holds a representation of the source code but syntactic elements such as parentheses are discarded. The different AST node types represent the parsed language. These node types are distinguishable by unique “tags”, assigned to each type in the language (Java with the JML extension in this case).

The “TagConstants” classes (located within the “ast” packages of both the “Escjava” envelope and the “Javafe” envelope) are responsible for assigning unique integer values to every type in the language. The “tag” of an AST node can be retrieved using the “getTag” method. The “tag” of the binary operator “AND” is for example, the integer 56. A detailed explanation of AST parsing techniques is beyond the scope of this report but can be found in “Programming Language Processors in Java: Compilers and Interpreters”[22].

4.2.1 Desugaring of Specifications

This section describes the refinements that take place at this stage of execution. The parsed JML specifications of the program undergo a process known as “desugaring”. “Desugaring involves the conversion of heavyweight specifications to lightweight ones. The lightweight specifications produced allow for the generation of a VC (in stage 6) the theorem prover can handle. The command line option “-showDesugaredSpecs” will print the desugaring process. An example of "desugared" specifications can be seen in Figure 7.

```
Desugaring specifications for RequiresNoParams.nonObject
   --- parsed specs --- /*@
/*@ requires integer == 0; *//*@ requires integer > 1; */@*/

Desugared specifications for RequiresNoParams.nonObject
   requires (\lblneg Pre:0.29.8 integer == 0) && (\lblneg Pre:0.30.8
integer > 1)
/*@ modifies \old(integer == 0 && integer > 1) ==> (\everything); @*/
```

Figure 7. Desugaring of Specifications

Figure 7 shows the desugaring of two contradicting specifications, taken from one of the test cases. It can be seen that the two separate preconditions ("requires") are concatenated to form of a single expression. This expression is split into a left and a right side, joined with a binary operator, "AND" in this case.

4.2.2 Traversing and AST

In order to help with traversing an AST, two methods, "childCount" and "childAt" exist for counting and extracting the (direct) children of a node[21]. These methods are important if we wish to manipulate the AST's produced, to remove certain annotations for example.

4.3 Translating of Java to Guarded Commands

The translation from Java to VCs is broken into three stages. First, the Java is translated to a sugared form of the GC language that includes high-level features such as iteration and method invocation. Second, the sugared GCs are "desugared" into a simple language based on Dijkstra's guarded commands[23, 24]. ESC/Java2's GC language includes commands of the form assert E, where E is a Boolean expression. Execution of a GC fails if control reaches a subcommand of the form assert E, when E is false[24].

As discussed in previous chapters, inconsistencies in specifications result in a VC that will be automatically satisfied regardless. Testing for specification inconsistencies involves asserting a false predicate. If this predicate is satisfied by the theorem prover, inconsistencies must lie somewhere within the specifications.

The set of test cases all contain "`assert false`" in the body of the routine in order to check the soundness of the specifications. The prover will detect this false predicate only when the specification inconsistency is removed.

It is worth noting at this point that the translated body of the Java routine is not necessary for the testing of specifications. Replacing the body of the Java routine with "`assert false`" before translation occurs is a possibility that would result in increased efficiency in the design.

4.4 Generation of the Verification Condition

Stage 6 involves the generation of a VC for each routine to be tested (each method or constructor in the class). A VC for a guarded command G is a predicate in first order logic that holds for precisely those program states from which no execution of the command G can go wrong[24]. The computation of a VC is similar to the computation of a weakest precondition[23] but includes optimisations to avoid exponential blow-up inherent in a naïve weakest-precondition computation[24, 25].

The generation of the VC is the final stage before the calling of the prover. It is evident that if predicates are to be iteratively removed, it must happen prior to the generation of VCs.

4.5 Design and Conclusion

As mentioned previously "Main.java," controls ESC/Java2's stages of execution. After careful consideration, the conclusion was reached to implement the extension immediately before execution of Stage 3.

4.5.1 Representation of a Routine

Stages 3 – 6 all occur within the "processRoutineDecl" method located in "Main.java" The return of this method is a short status message. This status message indicates whether the supplied routine "passed," "failed," "timed out" or that error occurred: "unexpectedly missing Simplify output." The routine is supplied in the form of an AST node named "RoutineDecl." "RoutineDecl" represents both method and constructor declarations.

The AST node "RoutineDecl," holds the specifications of a routine in two separate AST vector structures:

- ModifierPragmaVec – a vector of ModiferPragmas (pmodifiers)
- TypeModifierPragmaVec – a vector of TypeModifierPragmas (tmodifiers)

The vector of ModifierPragmas (pmodifiers) may represent all of the Modifier Pragmas of the JML extended Java language. Among these pragmas are "ensures" (the postconditions) and "requires" (the preconditions).

4.5.2 Manipulation of a Routines Specifications

The initial design was to call "processRoutineDecl" iteratively, removing elements of the pmodifers vector on each iteration. The idea was that this would result in the specification inconsistency being removed on one of the iterations. The removed pmodifier element could then be printed alongside the return of "processRoutineDecl" (the status message), allowing the user to easily interpret the result.

This design was flawed. It was presumed that each pmodifier element held individual Modifier Pragmas representing user supplied annotations, i.e. one pmodifier element for "`requires integer == 0`" and another for "`requires integer != 0.`" However, recall from Section 4.2.1, the "desugaring" process. Desugaring in Figure 7 resulted in the concatenation of the two separate preconditions ("requires") to form a single expression. This expression is split into a left and a right side, joined with the binary operator "AND" in this case.

In order to remove a single annotation from a "RoutineDecl," each pmodifier element must be searched for binary operators, such as that in Figure 7. If found, "processRoutineDecl" must be run separately with both the left and right side of this operator removed.

This design will result in "processRoutineDecl" (Stages 3 to 6) executing iteratively for each annotation removed. The details of the removed annotation, along with the result will be printed together. This technique is almost identical to the manual process users currently employ for locating specification inconsistencies. The following chapter describes the implementation of this design.

5 Detailed Design and Implementation

5.1.1 Enabling the Extension

ESC/Java2 provides a class in the “Escjava” envelope named “Options.java.” This class contains all of the command-line options possible in ESC/Java2. In order provide ease of use with this extension, the following option was added to this class:

```
-ConsistencyCheck, -inChk
```

These additions allow the user to make use of the “Inconsistency Checker” by simply including the option (either the long or abbreviated version) at the command-line, see Figure 8.

```
escj –ConsistencyCheck RequiresMultiParams.java

escj –inchk RequiresMultiParams.java
```

Figure 8. Using the “Inconsistency Checker”

An “if” statement, inserted before the call of “processRoutineDecl,” receiving a non-null value for this option enables the extension.

5.1.2 Locating Binary Operators

As a reminder, in order to remove a single annotation from a “RoutineDecl,” each “pmodifier” element must be searched for binary operators. As mentioned in Section 4.2, the “tag” of an AST node can be retrieved using the “getTag” method. This method makes the task of locating Binary Operators somewhat easier.

The “processRoutineDecl” method must be run separately with both the left and right side of this operator removed. In order to accomplish this a private method “visit” was created to perform a “depth first search” on a Modifier Pragma and return references to all the occurrences of binary operators “AND” and “OR”.

Using the results of this method the following steps will produce the desired output:

1. Rearranged the original Modifier Pragma AST “oMP” to produce a replica Modifier Pragma “tMP” missing one annotation.
2. Replace “oMP” with “tMP” in the “pmodifiers” vector of the “RoutineDecl”.
3. Execute “processRoutineDecl” on “RoutineDecl.”
4. Print results.
5. Return to step 1 unless all combinations have been tested.

The technique described above must be implemented for each element in a “RoutineDecl's” “pmodifiers” vector.

5.1.3 Problems Encountered

Unfortunately problems were encountered during implementation of the “visit” method. The “childAt” method (described in Section 4.2.2) was used in “visit” to iterate down through the tree. The idea being that every child of an AST node was itself and AST node; however, problems resulted when calling the “visit” method on children of the Modifier Pragma. Attempts to cast these children as AST nodes resulted in a cast exception being thrown.

The problem is not helped by the “Java1.4.2” type system (necessary with ESC/Java2) and could be statically avoided with a richer type system, similar to that in Java 1.5. We are however optimistic that this problem will soon be overcome and look forward to the presentation of a working model on Tuesday next, March 15th.

6 Conclusions and Future Work

A sound design for the extension of ESC/Java2’s functionality, to help with the detection of specification inconsistencies, was presented in this project. However, due to time frames and difficulties encountered a number of trade-offs were made with this design. This Chapter outlines some of the possibilities for building upon the framework of this design.

6.1 Extending checking ability

Specification inconsistencies come in many forms and there is a large amount of future work possible with this project. A major shortcoming to the designed solution described in this report is that it primarily helps the user detect precondition and postcondition inconsistencies. Class invariants and user added axioms are also a huge source of specification soundness errors and extension of this tool to cover detection of these errors would be a major improvement.

6.2 Increasing Efficiency

As mentioned in Section 4.3, there exists the possibility to avoid translating the Java body of a method to the GC language. The current design results in performing this translation for every annotation removed and is extremely inefficient. Discarding the body of the routine as described in Section 4.3 would however have implications for the postcondition tests in the current design. Revaluation of the design to incorporate this increased efficiency, without loss of functionality, would be a huge improvement.

7 References

[1] http://archive.eiffel.com/doc/manuals/technology/contract/

[2] http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html

[3] http://www.gauss.muc.de/tools/dbc/dbc-intro.html

[4] Bill Ball and Hoyt Duff. *Red Hat Fedora 2, Unleashed.* Sams Publishing 2005

[5] Dr. Joseph Kiniry

[6] http://www.linuxquestions.org/

[7] http://fedora.redhat.com

[8] http://www.gun.org/software/emacs/emacs.html

[9] http://www.emacswiki.org/cgi-bin/wiki

[10] http://www.bluemarsh.com/java/jswat/

[11] http://www.ej-technologies.com/products/jprofiler/overview.html

[12] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino and Erik Poll. *An overview of JML tools and applications.* www.jmlspecs.org

[13] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. *JML: Notations and tools supporting detailed design in Java.* August 2000.

[14] Gary T. Leavens and Yoonsik Cheon. *Design by Contract with JML.* November 22, 2004.

[15] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok and Joseph Kiniry. *JML Reference Manual.* Department of Computer Science, Iowa State University 2004.

[16] Peter Muller, Arnd Poetzsch-Heffter, and Gary T. Leavens. *Modular Invariants for Object Structures.* Technical Report 424.

[17] Yoonsik Cheon and Gary T. Leavens. *A simple and practical approach to unit testing: The JML and Junit way.* ECOOP 2002.

[18] Joseph R. Kiniry. *JML and ESC/Java Homework Exercises.* May 2004.

[19] Rustan M. Leino, Raymie Stata. *ESCJ 16c: Java to Guarded Commands translation.* Compaq Confidential (11 April 1997).

[20] Ian Kaplan. *A Java Compiler Front End. (March 5 2000 revised:* April 11 2000*)* http://www.bearcave.com/software/java/java_fe.html

[21] *ESCJ 24: Astgen Manual.* Compaq Confidential (September 22, 1998).

[22] David Watt and Deryck F. Brown. *Programming Language Processors in Java: Compilers and Interpreters*. Prentice Hall (April 15, 2000).

[23] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[24] Cormac Flanagan, Greg Nelson, Rustan M. Leino, James B. Saxe, Mark Lillibridge and Raymie Stata. *Extended Static Checking for Java*. Compaq Systems Research Centre, 130 Lytton Ave. Palo Alto, CA 94301, USA.

[25] Cormac Flanagan and James B. Saxe. *Avoiding exponential explosion: Generating compact verification conditions*. ACM 2001