Final Year Project Report

Automatic Soundness and Completeness Warnings in ESC/Java2

Barry Denby

A thesis submitted in part fulfilment of the degree of

BA/BSc (hons) in Computer Science

Supervisor: Dr. Joseph Kiniry

Moderator: Dr. Alexey Lastovetsky

UCD School of Computer Science and Informatics
College of Engineering Mathematical and Physical Sciences
University College Dublin
April 7, 2006

Table of Contents

Abstract

ESC/Java2 is an extended static checker for Java. Checking is performed through formal static analysis of a program. However, in some cases, this analysis is either unsound or incomplete. We will discuss unsoundness and incompleteness in the context of ESC/Java2, what problems unsoundness and incompleteness causes to static analysis of Java programs, where these cases are found in ESC/Java2, and how to detect these cases and warn the user.

Chapter 1: **Acknowledgments**

I would like to thank Dr. Joseph Kiniry for providing me with a lot of very helpful information and feedback regarding my work on this project, without it the advances made would not have been possible. I would also like to thank Alan Morkan one of Dr. Kiniry's postgraduate students for his help and advice on implementing code for generating graphs of trees created by ESC/Java2. Finally I would like to thank the SRG reading group for feedback on a presentation about this work I made to them.

Chapter 2: **Project Specification**

ESC/Java2 is an extended static checker for Java: it automatically proves that a Java program fulfills (parts of) its specification, written in the Java Modeling Language.

Concessions must be made to make ESC/Java2 automatic and easy to use, some of which are formal and some of which are practical. In particular, ESC/Java2 is neither a sound nor a complete verification system. In other words, it generates false positives, false negatives, and it completely misses some errors. All of these problems are due to the fact that its built-in semantics of Java and JML are intentionally incomplete (they do not cover the full language) and that the modular reasoning framework used is intentionally unsound (by design and because of the current theorem prover used).

Therefore, when ESC/Java2 detects a problem, it signals a warning, not and error, because we just cannot be certain that what has been checked is actually true.

This situation is unsatisfactory, but an obvious solution exists to this dilemma . Because we understand ESC/Java2's logics and can characterize them precisely, and because we have a rich context when reasoning about a given method, we know when the user is trying to verify programs in "dangerous waters".

For example, if you use floating point arithmetic, all bets are off. If you try to reason about specification with certain kinds of invariants that have universally quantified subexpressions, Danger Will Robinson!.

The purpose of this project is to characterize these "danger areas" and extend ESC/Java2's functionality to automatically warn the user when they are attempting to perform verifications about programs and/or specifications that are possibly confounded by the soundness and completeness problems of the tool.

Mandatory

1. Learn the ESC/Java2 object logics and their soundness and completeness issues and annotate the booklet "The Logics and Calculi of ESC/Java2" for clarity. 2. Learn the ESC/Java2 calculi and their soundness and completeness issues and annotate the above booklet for clarity. 3. Write a short survey on the soundness and completeness issues with modern object-oriented modular checking techniques. 4. Itemize full list of soundness and completeness issues from the three above stages in a distilled document. 5. Design and implement a contextually-aware soundness and completeness warning system for ESC/Java2.

Discretionary

1. Design and implement a system for outputting graphs of the Abstract Syntax Trees generated by ESC/Java2.

Exceptional

Chapter 3: **Introduction**

ESC/Java2 is an extended static checker for Java source code. It is a *static checker* because the analysis is performed without running the Java code. It is an *extended checker* because ESC/Java2 catches more errors than conventional static checkers such as type checkers.

The user is able to control how this checking is performed. This is done through the use of specially formatted comments called pragmas. Pragmas are written in the Java Modeling Language (JML) which will be discussed later.

As explained in the ESC/Java2 manual [5], ESC/Java2 does not provide formally rigorous program verification. The purpose of the tool is to catch some errors earlier than conventional methods such as testing. Such errors include null deference's and attempted access of array elements that do not exist. As a consequence of automation some trade-offs were made in the design of this tool. Trade-offs include, but are not limited to, how often errors are missed, how much time is required to perform the analysis and how often false alarms occur. The result of these trade-offs is a tool that is both incomplete and unsound in its static analysis.

The purpose of this project is to implement a warning system that generates a warning when a case of unsoundness or incompleteness occurs. It is then left to the user to take action regarding these cases. For example a user may choose to ignore these warnings or introduce testing on the code associated with those warnings. Before we can implement such a system we must first understand what unsoundness and incompleteness mean in ESC/Java2. However, these terms have more than one meaning within ESC/Java2 .

Chapter 4: **Unsoundness and Incompleteness**

In this section we will describe what the terms unsoundness and incompleteness mean in ESC/Java2 and provide and example of each.

4.1 Unsoundness

The ESC/Java2 manual [5] defines unsoundness as:

> An unsoundness is a circumstance that causes ESC/Java to miss an error that is actually present in the program it is analyzing.

As an example, consider the following fragment of Java source:

```
int[] array = new int[10];
for(int i = 0; i < 20; i++)
    array[i] = i;
```

This `for` loop will cause an `ArrayIndexOutOfBoundsException` when i is equal to 10 as we have an array of ten elements indexed 0 through 9. The exception occurs when we try to access array element 10 as elements 10 through 19 do not exist. By the definition above this is a case of unsoundness as ESC/Java2 did not notice that this error would occur. The reason why this happens is because, by default, ESC/Java2 does not consider all possible iterations of the loop. The ESC/Java2 manual [5] explains this behavior:

> In ESC/Java, loop invariants are optional. The checker considers only execution paths in which the loop body is executed at most once (and the test for being finished is executed most twice), rather than the potentially infinite number of paths that are really possible. Because of this simplification the checker doesn't need an invariant to analyze the loop.

This simplification makes ESC/Java2 unsound when analyzing loops. If this simplification was not made then ESC/Java2 could potentially analyze a loop forever. However, the user has the option of telling ESC/Java2 on the command line how many iterations of a loop should be considered. In this example, if the user specified more than ten iterations, then this error is identified correctly.

4.2 Incompleteness

The ESC/Java2 manual [5] defines incompleteness as:

> An incompleteness is a circumstance that causes ESC/Java to warn of a potential error, when it is in fact impossible for that error to occur in any run of the program it is analyzing.

For example, consider this fragment of JML-annotated Java source:

```
int i = 2;
i = i >> 1;
//@ assert i == 1;
```

In this example the bits of `i` are shifted to the right by 1 bit. Another interpretation of this example is that `i` is being divided by 2. The expected result of this operation is that `i` is assigned the value 1. The line "`//@ assert i == 1;`" is a JML pragma. It states that at this point of execution the variable `i` must have the value 1. When ESC/Java2 is run on this code however, it complains of a possible failure of this assertion. This is an incompleteness because, the assertion that `i` is equal to 1 will never fail.

Not all cases of unsoundness and incompleteness occur in Java code without pragmas. Taking the fragment of code above if the pragma "`//@ assert i == 1;`" was not in place that case of incompleteness would not be found, for this reason we will now discuss JML as the pragmas used in ESC/Java2 are the pragmas of JML.

Chapter 5: **JML**

In this section we will describe the Java Modeling Language and how it is used for the specification of Java programs.

5.1 The Java Modeling Language

Leavens and Cheon describe JML as a formal behavioral interface specification language for Java. One use for JML is the Design By Contract (DBC for short) methodology of developing software[4].

In DBC, whenever a class A calls methods of a class B, A is said to be a client of supplier class B. Clients agree to fulfill certain conditions before calling a method of the supplier class, while the supplier class will guarantee that some properties will hold after the method has executed. One could say that the client enters into a contract with the supplier class and the supplier class enters into another separate contract with the client.

JML uses annotation comments to specify contracts. JML annotation comments start with the `@` symbol and are embedded in standard Java comments. Every contract in an annotation comment must end with a semi-colon. For single line annotations we write

```
//@ contract ;
```

where `contract` is a JML contract. For multi-line contracts we write something that looks like this.

```
/*@
  @ contract 1;
  @ contract 2;
  @*/
```

As an example of a contract in JML, suppose we have a method called `foo` that takes in an integer `bar` as its argument.

```
//@ requires bar > 0
public static void foo(int bar)
```

The contract above states that, in order for one to call this method correctly, one must provide an integer argument that has a positive value. According to Leavens and Cheon if a

contract is violated an exception is thrown [4]. It is possible to combine many of these small contracts together in order to create very expressive contracts.

At this point we now understand what unsoundness and incompleteness are in the context of ESC/Java2 . Now we look inside of ESC/Java2 to find out how Java programs are represented internally and where in this representation these cases of unsoundness and incompleteness occur. The internal representation is the Abstract Syntax Tree, however before discussion of the AST it is necessary to visualize them. As ASTs can be drawn as graphs we first describe a means of graphing an AST.

Chapter 6: **DOT File Generation**

This section describes the DOT file format for drawing graphs and how it was applied to ESC/Java2 to generate graphs of the AST.

6.1 DOT File Format

The DOT file format as described by Gansner, Koutsofios, and North [1] is a file format for describing directed or undirected graphs through plain text. These files are processed by programs which take this text representation and convert it to an image of the graph. For the purposes of graphing an AST we concentrate on a small subset of all available operations the DOT file format provides. Figure 6.1 is an example of a DOT file using the commands we need. The `digraph` keyword indicates that this is a directed graph called `simple` and

```
digraph simple {
    root -> branch0
    root -> branch1
    branch0 -> leaf0
    branch0 -> leaf1
    branch1 -> leaf2
    branch1 -> leaf3
}
```

Figure 6.1: A simple DOT file

all nodes and edges relating to this graph are placed between a pair of braces. The other command we need to draw a graph is a command to draw an edge. Take the line
`root -> branch0`. The `->` operator states that we are drawing a directed edge starting from the node called `root` and is directed towards and ends at the node labeled `branch0`. Nodes are implicitly declared via their use in edge definitions. When we convert the graph to an image seen in Figure 6.2.

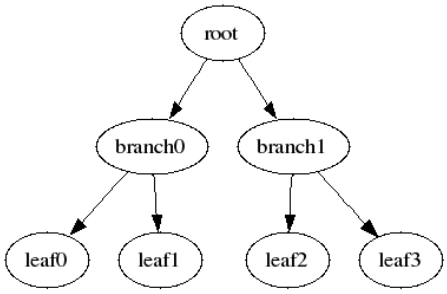

Figure 6.2: A simple graph generated from the DOT format

6.2 Generating DOT files of ASTs in ESC/Java2

To get ESC/Java2 to generate these graphs it was first necessary to add another command line switch to ESC/Java2 specifically for this functionality. ESC/Java2 's `escjava.Options` class was modified to include the switch for this option (–printJavaAstTree) and a boolean variable indicating if this functionality is enabled, as well as including a test to determine if this switch was passed to ESC/Java2 . The `escjava.Main` class of ESC/Java2 was modified to check the boolean indicating if this functionality was enabled. If the functionality was enabled, then code is run which gets the AST for each class that is passed to ESC/Java2 and passes it to a visitor for the AST that produces files in the format described above.

The visitor exploits the visitor design pattern [2]. In this pattern there exists a set of classes that can be visited but each of this classes must be visited in a different way. For example, in our AST a method declaration cannot be treated in the same way as a variable declaration. The pattern solves this problem by adding an `accept` method which takes in a visitor as its argument, to each class to be visited. In the visitor there exists a method for each class which defines how that class is visited. The `accept` method in each class then calls the appropriate method in the visitor to visit that class.

When the visitor is passed a class it first creates a file which will contain a representation of the AST in DOT graph format. First it writes the declaration of a directed graph and an opening brace into the file. Then, when the visitor traverses down an edge it determines the start and end nodes of this edge and creates an edge in the DOT file. The visitor does not write edges to the file when the visitor traverses back up the tree. After the entire tree is traversed we write the closing curly brace to the file and close the file. For example if we

```
public class Test {
        public static void main(String[] args) {
                int i = 2;
                i = i >> 1;
                //@ assert i == 1;
        }
}
```

Figure 6.3: A simple program that right shifts an integer.

generate the DOT file for the AST produced for the Java program in Figure 6.3 and convert it to graphical format we get the graph in Figure 6.4: As we are now able to visualize an

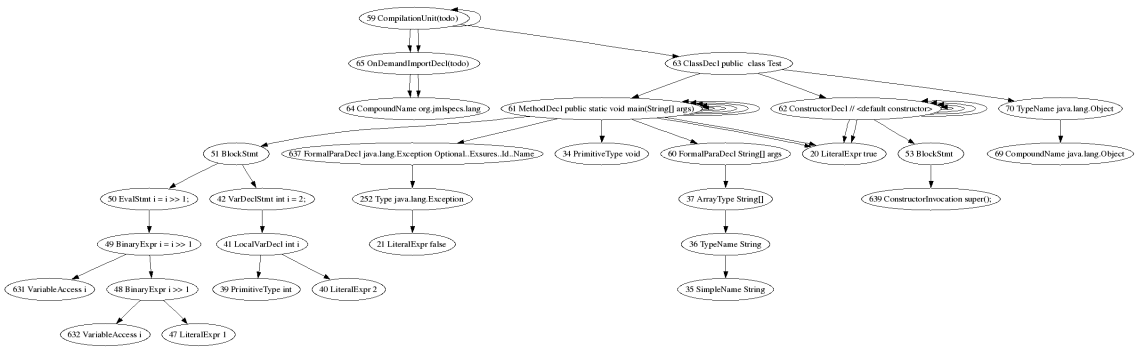

Figure 6.4: AST for the `Test` program

AST, we can now discuss what an AST is.

Chapter 7: **Abstract Syntax Tree**

This section will describe what an Abstract Syntax Tree (AST for short) is and its relation to unsoundness and incompleteness.

7.1 The AST

Wikipedia [3] provides this definition for the AST:

> an abstract syntax tree is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the node operators

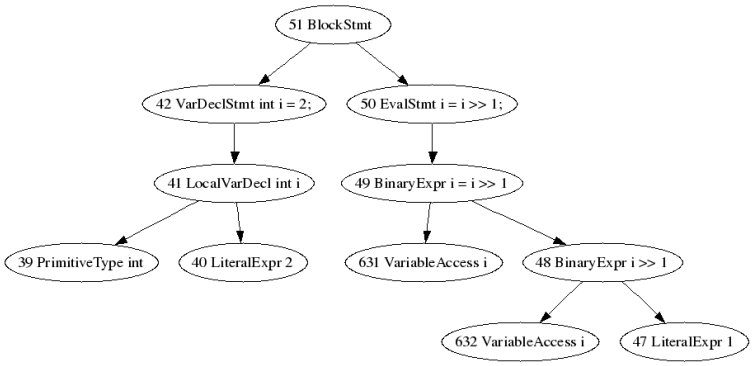

Figure 7.1: Subsection of AST for `Test` program

As an example consider the subsection of the AST for the `Test` program seen in Figure 7.1. Each node is labeled with its node ID, node type, and source code relating to that node. This piece of the AST corresponds to the opening brace after the definition of the `main` method to the closing brace of the `main` method in our `Test` program.

The `BlockStmt` corresponds to the the opening and closing braces associated with the `main` method in the `Test` program. The children of this node are the variable declaration for `i` (`VarDeclStmt int i = 2;`) and a statement which assigns `i` the value of `i` right shifted by 1 bit (`EvalStmt i = i >> 1;`). As the graph shows these statements can be broken down further into simpler operators and operands.

While many operators in the AST take more than one operand, it is possible, as this example shows, to have operators with a single operand.

7.2 Unsoundness, Incompleteness and the AST

An AST represents an entire Java class or program. Every Java source code has an equivalent AST. Using this AST we determine where in the associated code an unsoundness or incompleteness occurs. Thus, if we get an unsound or incomplete case we generate the graph of the AST associated with that unsoundness or incompleteness. Then, using the graph, we determine what parts of the AST defines this case. When those parts are determined we can then proceed to write code to warn about this unsoundness or incompleteness.

For example, in our `Test` program we know an incompleteness exists when the right shift operator is used. As Figure 7.1 shows there are three nodes with source code that includes the right shift operator. If any node in the AST contains the right shift operator we can warn about this incompleteness.

The above example shows how useful visualization of the AST is to understanding an unsoundness or incompleteness. Without the graph generation code understanding an unsoundness or incompleteness requires learning the structure of the AST. The graphs negate the need to learn the AST structure the graphs show the structure of the AST. Thus, understanding an unsoundness or incompleteness is both easier and quicker.

In the next section we will describe how ESC/Java2 was extended to automatically warn about unsound or incomplete cases.

Chapter 8: **Automatic Soundness and Completeness Warnings**

This section describes the design and implementation of the warning system.

8.1 System Design

ESC/Java2 's ability to reason about programs is constantly evolving. One reason for this evolution is because ESC/Java2 supports multiple provers and theories each of which has their own set of unsoundness and incompleteness issues. Thus, the system should support multiple sets of test cases, one for each prover and theory. Without these sets some tests may generate warnings that do not apply to the prover and theory that is used.

With this in mind our system must be easy to extend to accommodate newly discovered unsound or incomplete cases. However, if a test for any such case no longer applies to ESC/Java2 , then the test should discarded.

First we must have a means of determining if an AST contains any unsound or incomplete cases. To test for an unsoundness or incompleteness, we create a visitor that traverses the AST looking for nodes that determine if that unsoundness or incompleteness exists for the Java code represented by the AST. When a case is found in an AST, we emit a warning indicating which case along with the exact location of where the case has occurred in Java code, provided by the user.

8.2 Implementation

The first thing implemented was another command line option to enable this functionality. This was accomplished in the same way as the option for generation of DOT graphs for ASTs. Next, visitors were written to characterize each known case of unsoundness and incompleteness of ESC/Java2 . To do this we need to extend the `javafe.ast.Vistor` class of the AST for each individual unsound and incomplete case. In each of these extended visitors we override the methods for visiting the node types that may contain the case we are looking for. As an example, lets consider the incomplete case of right shifting.

In this example the right shift operator only appears in nodes of type `BinaryExpr`. Thus, if we wish to discover this incomplete case, we define a visitor for the AST by first extending the `javafe.ast.Visitor` class and override the method for visiting a `BinaryExpr` to check if a right shift operator is used. If the right shift operator is found in the AST, the visitor generates and emits a warning to the user.

From the above example a general method for adding a test for an unsound or incomplete

case to the warning system is derived:

1. Generate a graph of the AST associated with the program with the new unsound or incomplete case.
2. Determine which node types and what properties of those nodes determine if the case exists or not.
3. Write a visitor that checks for these nodes and properties. If the unsoundness or incompleteness is found generate a warning.
4. Call the visitor on each Java class processed by ESC/Java2

When ESC/Java2 processes the program in Figure 6.3 with the warning system enabled this is the output produced:

```
[sidewinder@predator testcases]$ escj -warnUnsoundIncomplete Test
ESC/Java version \escjava-CURRENT-CVS
[0.681 s 8720600 bytes]

Test ...
Prover started:0.533 s 12128376 bytes
[4.349 s 12349640 bytes]

Test: main(java.lang.String[]) ...
------------------------------------------------------------------------
./Test.java:5: Warning: Possible assertion failure (Assert)
//@ assert i == 1;
^
------------------------------------------------------------------------
[0.923 s 12316504 bytes]  failed

Test: Test() ...
[0.012 s 12472600 bytes]  passed
[5.285 s 12473480 bytes total]
./Test.java:4: Warning: The semantics of the right shift operator are incomplete.
i = i >> 1;
^
2 warnings
```

If the unsoundness and incompleteness warning system was disabled the only warning that is generated is the warning of a possible assertion failure. In this case the analysis of the assertion is incorrect as this assertion will never fail for the reasons given in Section 4.2. Without the warning system the warning of the assertion failure is a source of confusion. When the warning system is enabled however, the second warning reduces the confusion about the first warning.

As we now have a system for warning about soundness and completeness issues we will now conclude our work.

Chapter 9: **Conclusions & Future Work**

This section will outline the conclusions of this work and future work to improve it.

9.1 Conclusions

An automatic soundness and completeness warning system was implemented. This system alerts users of unsoundness and incompleteness issues that arise when ESC/Java2 is processing their code. The system is easy to extend when adding or removing tests for unsoundness or incompleteness. This system is still experimental thus, the code is not very robust to handling errors.

An AST graph generator was also implemented. The generated graphs improve the understanding of how these trees are structured. This generator simplifies the work necessary to understand an unsoundness or incompleteness. This code is also experimental. Some edges in an AST are not handled correctly because a minority of child nodes are not extended versions of the basic `javafe.ast.ASTNode`. For example in Figure 6.4 there are edges that start and end at the same node. This is a side effect of not handling children that are not extended versions of `javafe.ast.ASTNode`.

9.2 Future Work

In future we want to discover and characterize more cases of unsoundness and incompleteness. By understanding these cases we hope to improve the semantics of ESC/Java2 . Eventually when these semantics improve we wish to enable the warning system by default.

However, the highest priority is to make this system more robust to handling errors. The graph generator could also be improved because some edges are not handled correctly as explained in the conclusions section.

Bibliography

[1] E. Gansner, E. Koutsofios, and S. North. Drawing graphs with dot.

[2] Javaworld http://www.javaworld.com. Java tip 98: Reflect on the visitor design pattern.

[3] Wikipedia http://www.wikipedia.org/. Abstract syntax tree.

[4] G.T. Leavens and Y. Cheon. Design by Contract with JML. 2005.

[5] K. Rustan, M. Leino, G. Nelson, and J.B. Saxe. ESC/Java User's Manual.