Full Verification of the KOA Tally System

Final Year Project Final Report

Fintan Fairmichael

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 Mr. John Dunnion.

Department of Computer Science

University College Dublin

12 March 2005

Abstract

The European elections of June 2004 saw The Netherlands's first use of remote voting; that is: voting via the Internet or over the telephone. Most of the system was designed, implemented, and tested by a Dutch consulting firm, LogicaCMG. The Security of Systems (SoS) group[1] also developed a critical portion of the application, the system responsible for counting the votes after they had all been received and recorded. This tally application was developed with formal methods, extensively tested and partially verified to the extent that was possible within the given timeframe. This report discusses an attempt to complete the specification and verification of this vote counting program. The tools used for specification and verification: the Java Modeling Language (JML) and the Extended Static Checker for Java (ESC/Java2) are described and analysed. The issues involved with completing the JML specification and verifying it using the ESC/Java2 tool are also discussed. This final report ends with details and analysis of the work completed during the duration of this project.

[1] The Security of Systems group, led by Professor Bart Jacobs, University of Nijmegen, The Netherlands, of which Dr. Joseph Kiniry was a member from October 2002 to October 2004.

Table of Contents

1 Introduction

In 2003–2004, the Dutch government had an Internet voting system designed and developed by a Dutch consulting firm LogicaCMG. This system was to be initially used in the European elections held in June 2004. This system was named “Kiezen op Afstand" (KOA) – in English: “remote voting”. In late 2003 Prof. Bart Jacobs of the SoS group participated in an external review of the requirements and design of this application. One of the recommendations made by the panel was that the system should not be designed, implemented and tested all by the same company. As a result of this recommendation, the government decided to accept bids for the creation of a separate vote counting subsystem, to be implemented in isolation by a third party. This separate tally application would allow the vote counting to be independently verified. The SoS group put forward a proposal to write this application, and were successful in this bid. The key idea behind their tender was that the vote counting program should be formally verified using JML and the ESC/Java2 tool [6].

1.1 The KOA Vote Counting System

The vote counting system formed only a small but important part of the whole KOA system. This provided the SoS group with a suitable opportunity to test the use of some of the formal techniques and practices that they had been developing. Three members of the SoS group (Dr. Joseph Kiniry, Dr. Engelbert Hubbers and Dr. Martijn Oostdijk) built this application over only a four week period, due to the severe time constraints placed upon them because of the impending election [3]. Java was chosen as the programming language to implement the system on so that JML could be used as the formal specification language.

The three authors broke the application down into three parts: core data structures and algorithms, file input/output (I/O), and graphical I/O. Each author was to develop one of these subcomponents and, due to the time constraints, a “best-effort” verification was only attempted with the core modules using the static checker ESC/Java2.

1.2 The Purpose of this Project

The purpose of this project is to extend and modify the specifications written for the released version of the KOA tally application. As previously stated, these specifications were incomplete and there was the possibility that some were inaccurate or did not match the code.

The ultimate goal of this project was to bring the KOA tally application as close as it is currently possible to being fully specified or, at least, to bring it as close to this goal as was possible within the scope and time-period available.

Even though the work on the original application has already been released to the public domain and used successfully for its desired purpose, the work carried out in this project will further the development and use of the technologies involved, and will serve as a useful case-study of the issues involved in the specification and verification process.

This report details the tools and technologies used in writing and verifying the specifications and discusses some of their key strengths and weaknesses. A description of the achievements and findings made over the duration of this project are also given.

1.3 What is JML?

JML is a notation for formally specifying the behaviour and interfaces of Java classes and methods [11]. This means that it is used to describe both the details of how a Java interface or class interfaces with clients, and how it behaves from a client's point of view.

JML is designed to be used by working software engineers [9]. It uses Java's expression syntax, as well as adding its own special-purpose expressions, and as such is intended to be easier to read and use than specification languages that use mathematical notations [10].

When JML specifications exist for Java modules (classes and interfaces) it is possible to compare the written code against its specifications. There are numerous tools available to do these comparisons, including the JML Runtime Assertion Checker (jmlrac) and the Extended Static Checker (ESC/Java2) [2]. Using these, discrepancies between the specification and the behaviour of the code can be detected. For more on JML, its tools and uses, see [1].

1.3.1 JML Example

```
/*@ behavior                                                   //line 0
  @  requires intArray.length >= 1;                            //line 1
  @  assignable \nothing;                                      //line 2
  @  signals (NullPointerException) false;                     //line 3
  @  diverges false;                                           //line 4
  @  ensures \result                                           //line 5
  @            == (\max int k;                                 //line 6
  @                    0 <= k && k < intArray.length;          //line 7
  @                    intArray[k]);                           //line 8
  @*/                                                          //line 9
public abstract /*@ pure @*/ int arrayMax(                     //line 10
                              /*@ non_null @*/ int[] intArray);//line 11
```

Figure 1. JML Example

The example given in *Figure 1* gives a simple use of JML and some of its syntax. It demonstrates the following features of JML:

- JML annotations are contained in comments beginning with **`//@`** or **`/*@`**.

- The **behavior** keyword (line 0) indicates that the following lines constitute a behavior for the *arrayMax* method.

- The **requires** keyword (line 1) states a precondition for the *arrayMax* method, namely that the argument *intArray* is presumed to be of length >= 1.

- The **assignable** keyword (line 2) states a frame condition for the *arrayMax* method, namely that no fields are assigned to during its execution.

- The **signals** keyword (line 3) states a postcondition for the *arrayMax* method that must hold if an exception is thrown, in this case that it never throws a *NullPointerException*.

- The **diverges** keyword (line 4) gives a condition under which the method may diverge and never return to the caller. The condition *false* indicates that this method will never diverge – it will always terminate and return.

- The **ensures** keyword (lines 5-8) states a postcondition for the *arrayMax* method that must hold if the method terminates normally, namely that the int returned is the maximum for the given array.

- The **pure** keyword (line 10) states a frame condition for the arrayMax method. Purity of a method is discussed in detail later (*Section 2.1.1*).

- The **non_null** keyword (line 11) states a precondition for the *arrayMax* method, namely that the argument *intArray* is presumed to be non-null.

Another important keyword in JML is the invariant. For example:

```
//@ invariant numberOfItems <= maximumNumberOfItemsAllowed;
```

This example states that the variable *numberOfItems* must have a value less than or equal to *maximumNumberOfItemsAllowed* in the postcondition of the constructor as well as in the precondition and postcondition of all constructor methods.

1.3.2 JML Behaviours

The *behaviour* of a method or type describes the state transformations it can perform. The behaviour of a method is specified by describing: the set of states in which calling the method is defined, the set of locations the method is allowed to assign to (and hence change), the relationship between the state when the method was called and the state when it returns (either normally or by throwing an exception), as well as the states for which the method might not return to the caller.

Most computer scientists should be familiar with the notation, known as a Hoare triple:

```
{P} S {Q}
```

denoting that when a program is in a state that satisfies P and the statement S is executed and terminates, the program will the be in a state that satisfies Q. When applied to an object-orientated programming language such as Java, P is the precondition, S the method body and Q the postcondition.

JML behavioural notation allows a similar reasoning in that:

- It provides precondition terms (requires clauses) and postcondition terms (ensures and signals clauses)

- For every behaviour of a method, the method body must guarantee that all program states matching the precondition will produce a resultant state that matches the postcondition, if it terminates.

When forming the precondition, the arguments given in the method call may be used. When forming the postcondition, the returned object or value, as well as the arguments given in the method call may be used. The variables used in the postcondition clause may be taken from the pre-state or the post-state of the method (the pre-state is indicated by use of the *\old* keyword). JML behavioural notation also adds the following constraints:

- A method may not be invoked unless the current state satisfies the precondition of at least one of the method's behaviours.

- Constraints may be applied to what the assignment section (the method body) may do:
    1. Assignable clauses place restraints on what variables may be changed.
    2. Signals clauses give conditions under which an exception can be thrown.
    3. Diverges clauses specify under what circumstances the method may never terminate.

1.4 What is ESC/Java2?

The ESC/Java2 tool is based upon the Extended Static Checker for Java (ESC/Java), developed originally by Digital Equipment Corporation's Systems Research Lab (SRC). It is capable of automatically, formally, statically checking JML specifications against the program code (though only some of the JML specifications were actually used in verification) as well as detecting simple errors such as null dereferences and array bound errors. The authors of the ESC/Java2 tool (David Cok and Joseph Kiniry) took on the task of updating the original program so that it would understand recent versions of Java and parse, as well as check as much as possible, the current version of the JML annotation language [4][3]. It is a program under active development, and is currently in its ninth alpha release.

The checking performed by the ESC/Java2 tool is done modularly. This means that each method *M* is inspected individually and the verification process examines the code of the method along with any specifications that relate to it, including the specifications of all methods used by *M*.

As a static checker, the ESC/Java2 tool reasons about all possible paths through a method. This means that it is capable of deducing if a given condition is true at all times during all possible routes through the code, or whether there is a possible path that would allow the condition to be false. In this respect it differs majorly from run-time checkers (such as the jmlrac) as they are only able to verify conditions for certain routes though a body of code exercised by, for example, a unit test.

1.4.1 ESC/Java2 Examples

Below are some example input classes with the output they produce from ESC/Java2:

```
class Example {
   public void method1(Object[] o) {
         int length = o.length;
         String s = (String)o[0];
         . . .
   }
}
```

Figure 2. The Example class

The class given in Figure 2 may at first glance appear to be harmless, but upon supplying it to ESC/Java2 as input, 3 warnings are produced.

```
------------------------------------------------------------------------
Example.java:3: Warning: Possible null dereference (Null)
        int length = o.length;
                      ^
------------------------------------------------------------------------
Example.java:4: Warning: Array index possibly too large (IndexTooBig)
        String s = (String)o[0];
                            ^
------------------------------------------------------------------------
Example.java:4: Warning: Possible type cast error (Cast)
        String s = (String)o[0];
                   ^
------------------------------------------------------------------------
```

Figure 3. Output from ESC/Java2 on the Example class

The first warning in Figure 3 indicates the possibility of a null dereference (i.e., a *java.lang.NullPointerException*) occurring. This can be resolved in one of two ways: either by adding a specification that ensures that this method is only invoked with a non-null *Object* array *o* or by altering the code to ensure the action is only carried out if *o* is not null. The former can be accomplished by adding the following precondition to the method:

```
//@ requires o != null;
```

The latter can be achieved by replacing the offending line with:

```
if (o != null) {
   int length = o.length;
}
```

The decision as to whether to modify the code or the specifications in order to resolve a problem will be totally dependent on circumstances. In some cases the specifications will be written in advance (DBC) and therefore it is the code that should be changed. Other times the code may be written in advance and it might be desirable to only alter the specifications.

The second warning states that the index given when accessing the array is possibly too big. Since at present we may pass in any *o* that we like, even one of length zero, then there is the possibility that the index used will trigger a *java.lang.ArrayIndexOutOfBoundsException*. This can be prevented by adding the following clause:

```
//@ requires o.length > 0;
```

This warning may also be prevented by modifying the code similar to the technique used for the null dereference warning above.

The final warning has pointed out that the *Object* stored at index 0 in *o* may not be of type *String*. If this is the case then a *java.lang.ClassCastException* will be thrown. This can be avoided by modifying the code in a similar manner to before, or by adding the precondition:

```
//@ requires o[0] instanceof String;
```

JML expressions can use logical operators (with the same syntax as Java), and thus the three clauses just stated can be added to the Example class on a single line as shown in Figure 4:

```
//@ requires o!= null && o.length > 0 && o[0] instanceof String;
class Example {
   public void method1(Object[] o) {
          int length = o.length;
          String s = (String)o[0];
          . . .
   }
}
```

Figure 4.   Modified Example class

Figure 6 shows the output generated from ESC/Java2 when the class Example2 (*Figure 5*) is supplied as input.

```
class Example2 {
    public void method2() {
          Object[] p = new String[10];
          p[0] = new Object();
          . . .
    }
}
```

Figure 5.   Another example class

The error detected by ESC/Java2 would have thrown a *java.lang.ArrayStoreException* at runtime, however the code will compile with no errors.

```
------------------------------------------------------------------------
Example.java:11: Warning: Type of right-hand side possibly not a subtype
of array element type (ArrayStore)
        p[0] = new Object();
             ^
------------------------------------------------------------------------
```

Figure 6.   Output from ESC/Java2 on the Example2 class

The examples just given give a brief insight into the ESC/Java2 tool's ability to quickly and easily detect some common programming errors. The usefulness in using this tool as an aid in producing correct software should already be apparent. These examples, however, only scratch the surface of the complexity of errors in code (and specifications) that the ESC/Java2 tool can detect.

2 Background Research

2.1 JML

JML, its features, strengths and weaknesses are subjects of sufficient depth to merit lengthy research papers on their own, and will therefore only be discussed here in brief and with most relevance to this project (for more information on JML see the JML project website[1]).

When writing specifications using JML there are two standard processes used by application designers and developers. The two extremes might be viewed as “pre-code” vs. “post-code” processes. The former process is widely known as Design by Contract (DBC) and the latter typically is seen when one must add specifications to already written code (sometimes referred to as “Contract by Design”). In simple terms, DBC is the case where the specifications for a class (the “contract”) are written *before* the code. When the code is written it then has to conform to the properties given by its specification [10]. When adding specifications to already written code, the aim is to show (and ideally prove) that the code matches key properties and conditions. This can be done in a cumulative fashion, with small parts of specifications being added at a time, and checked against the code.

2.1.1 The Issue of Purity

There are some problems to do with the definition of what are called “pure” methods in the JML vernacular. A *pure method* is one that does not modify any variables or memory locations that were visible in its pre-state. This means that a *pure constructor* may assign to the non-static fields of the class, as they were not visible before the constructor was called. Static fields of a class and how they relate to purity is a complex issue, since static fields are visible in the pre-state of every method. The relationship between pure methods and static fields is currently being debated amongst the JML researchers.

At present, only pure methods (not pure constructors) may be used in JML specification clauses, as these are the only methods that are guaranteed not to affect the state of any objects. The use of a non-pure method in a method that is otherwise pure results in the previously pure method becoming a non-pure method also.

One specific problem regarding the issue of purity arises when dealing with the action of throwing an exception. Currently a method cannot be declared pure if it may explicitly throw an exception as the constructors for the *java.lang.Exception* class are not specified to be pure. When an *Exception* object is thrown (created) it is allocated memory space and added to the Java runtime heap. The Exception object is located above the “top” of the heap that existed prior to the call to the method that threw the exception, and therefore is unobservable to the outside. These properties mean that a method that throws an exception, and in no other way violates its purity, should be pure. At present, methods that throw an exception cannot be declared pure, and due to the inability to specify conditions hinging on objects created inside the method, can only be specified as possibly modifying everything.

There are numerous instances of problems with purity that are similar to the problem with throwing an Exception just outlined. Two commonly encountered examples are the methods *equals* and *hashCode* which are defined in the *Object* class but are commonly overridden in other objects. These two methods should be declared pure but due to varying implementations of the methods this is not always the case.

It is important that the issues that have been identified with the use of purity are resolved as soon as is possible. Incorrect use of purity affects the ability to accurately write specifications that model the behaviour of a program correctly. Purity of methods is an ongoing area of research and discussion for those involved in the development of JML.

2.2 ESC/Java2

ESC/Java2 can be thought of as an automated verifying compiler as it requires little input from the user other than the annotation of the code provided. In its default usage, it modularly analyses each method of a class, and translates its specification and code into guarded commands. A weakest precondition is derived from these and then a validity check of the specification against the code is made using the Simplify first-order theorem prover [7].

When using the ESC/Java2 tool, the warnings shown are not necessarily all of the warnings that ESC/Java2 could possibly find with the input. It is designed to only give the user enough information to indicate the correctness of the code. Understanding this exact behaviour of the ESC/Java2 tool requires expert knowledge of its workings. Non-expert users should however be aware that due to this, resolving one issue that ESC/Java2 has indicated may not reduce the number of errors reported, and can in fact increase the number of errors given.

As discussed in *Section 3.2.2*, the verification provided by ESC/Java2 is neither complete nor is it sound [12][5], but the tool can still be used to find lots of potential bugs in a program quickly and easily. As the development of ESC/Java2 (coupled with the evolution of JML) progresses, the completeness and soundness of the tool will increase, as well as the confidence one can place in its results.

3 Analysis of the Tools/Applications Used

In this section of the report some of the strengths and difficulties/weaknesses of working with the tools and applications employed in this project are described. The list of strengths, difficulties and weaknesses given is not exhaustive and is discussed with most relevance to the work involved in this project.

3.1 JML

3.1.1 JML Strengths

JML's biggest strength is that it is used with Java. Java is currently one of the most widely used programming languages and has numerous benefits, particularly platform independence. As JML works with Java, and its tools are, in fact, written largely in Java, it enjoys the same strengths.

The fact that JML utilises Java's expression syntax, as well as adding some of its own, means that users familiar with Java are able to learn to use it quickly. This use of expressions already familiar to a Java programmer also makes JML annotations easier to read and understand. Simple, short and clear methods are easily specified using JML. If code is written in this manner it is straightforward to add the appropriate specifications at a later date.

JML provides a good method for recording detailed design decisions or documentation of intended behaviour, for a software module [8]. It is important in the context of the KOA vote counting application to record the intended behaviour of the code: to count all of the votes and output this result correctly.

3.1.2 JML Difficulties/Weaknesses

The largest problem in dealing with JML is also a problem with all formal specification languages: the difficulty in writing specifications. Writing specifications for existing code especially can be immensely difficult. There are many factors that contribute to this. Some of these, those that are most relevant to writing a JML specification for the KOA tally application, are discussed here.

It is very difficult, and sometimes impossible, to accurately write specifications for long and complex methods in an already written Java class. As an extreme analogy, consider the difference in difficulty of writing the Javadocs for an accessor/getter method compared to an obfuscated method several hundred lines of code in length.

If the code is being specified by someone other than the original author(s), this can increase the difficulty of the specification process even further. Understanding another author's code can be very challenging, even more so when code is poorly commented or contains large, unwieldy methods.

JML behavioural specifications specify the behaviour of a method as a whole; they therefore cannot describe anything about variables that are created inside a method's boundaries. This is not a characteristic unique to JML, but is also a standard, and necessary, feature of all specification languages. Certain workarounds can be used, but in some cases the behaviour of a method simply cannot be specified exactly using JML. In these cases, all that can be done is to write the strongest specification possible that is still correct for the given code. Weak specifications for a certain method can lead to problems with other methods that rely on or use its specifications. This can again lead to specifications that are weaker than are desirable.

Writing specifications for loops is another challenging task when specification writing. In fact, only simple loops may be accurately specified using JML. Nested loops and complex loop guards can increase the difficulty considerably.

The issues involved with the purity of methods, as discussed in *Section 2.1.1*, can complicate specification writing. Furthermore, methods incorrectly specified as being pure or those incorrectly specified as being non-pure can lead to specifications that are, respectively, too strong or too weak.

Dealing with inheritance in specification writing brings its own set of problems and complexities. The specification of any method should accurately describe the behaviour of any subclass's implementation of the method, as well as its own, without placing too many (or too few) restrictions. Specifying for inheritance is immensely difficult and, whilst inheritance cannot be avoided completely, every effort should be made to minimise the complexity of the inheritance relationships present in a program.

A final problem is that the JML specifications of the Java runtime library are not complete or verified. Tools are not sophisticated enough to fully verify the specifications contained within the JML suite of these classes. This means that verification of code that relies on these specifications may encounter problems if there are inconsistencies or errors in them. Some of the JML specifications written, while they may or may not be correct, are too complex to be used accurately in the verification process when using the ESC/Java2 tool. As a result of this, some of the ESC/Java2 developers have been working on rewriting the JML specifications of the core of the Java runtime library using less complicated specifications. If this rewriting is carried out successfully, it should then be possible to use these specifications successfully in the verification process when using ESC/Java2.

3.2 ESC/Java2

3.2.1 ESC/Java2 Strengths

The key strength of ESC/Java2 is the ease of its use. For a programmer with pre-acquired knowledge of JML, learning to use ESC/Java2 for simple verification purposes is straightforward. The fact that the tool is automated, only requiring the initial code and specifications as inputs, is also a major benefit to usability.

ESC/Java2 is good at detecting unexpected conditions arising from conditions such as null pointers or out-of-bounds array indices [3]. It inherits these properties from the initial ESC/Java program. This makes it a very useful tool for detecting conditions that could lead to data loss or to a program crashing. The benefits of this with regard to the vote counting system are obvious.

The tool is fast (partly due to its automation) and is largely platform independent: it has been successfully tested on Linux, Mac OS X and on Windows systems. It also runs on a number of other UNIX platforms. This is a benefit as ESC/Java2 therefore does not require any specialist hardware making it a widely available and usable utility.

The recent release of the ESC/Java2 plugin [14] for the Eclipse Integrated Development Environment (IDE) [13] will increase the tool’s appeal to those who prefer to develop programs in a graphical IDE. The plugin effectively and efficiently integrates the tool’s functionality into the Eclipse workbench. There are many beneficial features to the plugin, especially the underlining of code or specifications that are problematic.

3.2.2 ESC/Java2 Difficulties/Weaknesses

The weaknesses of ESC/Java2 largely stem from the fact that it is not complete and not fully sound. Some of the properties that the tool tries to check are undecidable in the worst case, and therefore ESC/Java2 contains a degree of incompleteness and unsoundness by design. It may sometimes generate bogus warnings or fail to detect legitimate errors. The tool also inherits the weaknesses (as well as the strengths) from the use of JML.

The issues involved with pure routines for JML are also relevant for ESC/Java2. It is also an active area of research and discussion for the developers.

Currently ESC/Java2 is unable to deal with bitwise operations. This means that ESC/Java2 is unable to reason about any conditions that rely on the knowledge of a variable’s value after a bitwise operation.

Overall the incompleteness and unsoundness of the tool means it is possible that the tool may generate errors for conditions that might never happen, or similarly pass conditions that might not always be true. It also means that there may be parts of a program and specification that cannot be verified using ESC/Java2. It currently takes expert knowledge of the ESC/Java2 tool to know that its incompleteness or unsoundness is affecting the accuracy of the warnings produced. Dealing with these situations is not straightforward either, for example, a user can specify (using the *nowarn* pragma) for ESC/Java2 not to check conditions on a certain point of the code, but this is hardly an ideal solution.

The greatest difficulty in using the ESC/Java2 during the duration of this project was its ongoing development. At the beginning of this project the ESC/Java2 tool was unable to accurately reason about the built-in *String* class in the *java.lang* package. Strings are immutable objects with a highly complex specification. In January (approximately half-way through this project) some major updates were made to the tool to try to resolve these issues. Unfortunately these updates introduced a major bug into the tool. These caused the tool to fail when checking the majority of the methods in many of the KOA system's classes. As strings are used liberally throughout the KOA tally application, this left the regrettable situation of choosing between using an old version of the ESC/Java2 tool that could not accurately reason with strings, or a newer version that crashed on the majority of the application's code. This bug was recently fixed (approximately two weeks before the completion of this project), but provided a major hindrance to the progress made in the specification and verification process.

Finally, using the ESC/Java2 tool has a long learning curve. While one can become a basic user almost instantly, it takes months to become an expert in understanding some of the output generated (counterexamples in particular), or in resolving problems that are highlighted. For those users who prefer a graphical IDEs the recently released ESC/Java2 Eclipse plugin will alleviate this problem somewhat.

3.3 KOA System Review

When the KOA vote counting system was being designed, precedence was given to verifying the core units and the remainder was lightly annotated with JML notation. The core classes were designed by contract (see [10]) and as result have reasonable specification coverage.

	File I/O	**Graphical I/O**	**Core**
Classes	8	13	6
Methods	154	200	83
NCSS	837	1599	395
Specs	446	172	529
Specs:NCSS	1:2	1:10	5:4

Figure 7. KOA initial release system summary

Figure 2 (from [3]) summarizes the size (in number of classes and methods), complexity (non-comment size of source, or NCSS for short), and specification coverage of the three subsystems, as measured with the JavaNCSS tool version 20.40 during the week of 24 May, 2004 – this is the version of the program that was released and used in the European elections in June 2004.

3.3.1 Core Subsystem

At the time of its initial release, verification coverage of the core subsystem was good, but not 100%. Approximately 10% of the core methods (8 methods) were unverified due to issues with ESC/Java's Simplify theorem prover (e.g., either the prover did not terminate or terminated abnormally). Another 31% of the core methods (26 methods) had postconditions that could not be verified, typically due to completeness issues discussed above, and 12% of the methods (10 methods) failed to verify due to invariant issues, most of which are due to suspected inconsistencies in the specifications of the core Java class libraries or JML model classes. The remaining 47% (39 methods) of the core verified completely.

Since 100% verification coverage was not possible in the timeframe of the original project, to ensure the KOA application was of the highest quality level possible, a large number unit tests were generated with the *jmlunit* tool (part of the JML suite) for all core classes. A total of nearly 8,000 unit tests were generated, focusing on key values of the various datatypes and their dependent base types. These tests cover 100% of the core code and are 100% successful [3].

3.3.2 Graphical and File I/O Subsystems

The non-core portions of the application were poorly covered by JML specifications. The graphical portion of the system had the worst coverage of all, with only 1 line of specification for every 10 lines of code.

The file I/O part of the system's specifications currently focuses on contracts to ensure that data-structures in the core subsystem are properly constructed according to the contents of input files.

The GUI portion was modelled as a state machine which is tightly coupled to the KOA application's state. Specifications were written to ensure that the state of the program could only move forward states and not backwards, unless it is being fully reset and returns to the initial program state. There are 9 states in total; these are listed in order below:

```
Initial → Cleared → Candidates Imported → Votes Imported →
Private Key Imported → Public Key Imported → Voted Decrypted →
Voted Counted → Report Generated
```

The state of the program was stored as an *int* and an invariant was used to guarantee that the state was either equal to the previous state (remained the same), or was equal to the previous state plus one (moved forward one state), or was equal to the initial state (was reset).

4 Analysis of Work to Date

This section looks at what work was carried out over the duration of this project, and compares this to the original aims/goals of the project.

Before any practical work on this project was started it was decided that any changes to the original application would be to the specification only. There were two main reasons for this:

1. It is preferable to be able to show that the originally released program was correct by finishing its specification and verifying it against this, than to alter the code to produce a different (if only slightly) program. If the code is altered any verification then performed does not necessarily say anything about previous versions of the program.

2. It is easier to track the changes to a program if either the code or the specifications are being altered, but not both.

In order to keep track of any changes made a Concurrent Versioning System (CVS) repository was used, and the initial files stored were those from the initial release of the KOA tally application. It was then possible to track every step of any alterations made to the specification, and even to display the differences between a two versions of a file on a line-by-line basis.

Changes made were documented with a justification for the change and took the format:

```
//FF - <comment>
```

Where "FF" is this report's author's initials and "<comment>" would be the actual comment made regarding the change. It is hoped that by following this template for specification modification, that interested parties can easily realise the specification changes that have been made as well as the reasons for these.

4.1 Work Completed to Date

For the first half of this project much of the time spent was on background reading and preparatory work: numerous research papers ([1..12]) were read and studied. There is a great deal of information available on the topics of JML and static verification using the ESC/Java2 tool, and it was attempted to become as familiar with as much of this material as possible.

Significant time was also spent on practicing the techniques of writing specifications and then verifying them using ESC/Java2. These techniques have a long learning curve, and until a good deal of experience is obtained, progress with specification writing can be slow. This all depends on the complexity of the code for which the specifications are to be written.

The class *Hex* was taken to be the first task in specification writing for the KOA tally system. This class was chosen as it was a standalone class with only static methods, no JML coverage, and it was thought that it would be straightforward to specify. This assumption turned out to be wrong as the methods in the class contained many uses of bitwise operations, variables created inside the methods, one long densely-coded method, as well as many uses of strings and arrays. It took a great deal of time to even partially specify the class, but at each step of the way lessons were learned that have and will prove useful when writing and verifying specifications in the future.

The next class to be looked at was *AuditLog.* This is the class that records information about the vote counting as the application proceeds. This information is then used at the end of the vote counting to help fill in the details for two of the reports then generated. This class kept track of the program's progress in a similar manner to that which was used for the overall program state. There were multiple invariants used to ensure the program and auditing proceeded in the correct fashion. Several corrections were required for this class, the bulk of which were modifications to the behaviours of the methods that allowed the *AuditLog*'s state to change. The original specifications allowed the possibility of the variables to be changed to a state where the invariants would not hold. The changes made to this class's specifications disallowed any actions that would violate the object invariants.

Next the class *Task* was examined. This was one of the most challenging classes to specify as it was the only class in the whole KOA system which had subclasses. As discussed earlier, inheritance can severely increase the complexity of the specification process. There were eight subclasses of *Task,* and it was important to ensure that it captured as much of the subclasses' behaviours in its specification without being too restrictive. Since this class and all of its subclasses were already written, this involved making the specifications fit the code as tightly as it and all of its subclasses would allow. There were numerous changes made, the majority of which involved fixing inaccuracies between the specifications and the code of the class and the subclasses.

4.2 Work not Completed

The KOA tally application is not fully specified and verified. While working on this project there were a number of factors that made this goal infeasible. Given below are those that had the biggest detriment:

1. Time constraints: the time required to become familiar with all of the technologies and tools involved with this project meant that there was less time to be spent on the specification and verification effort.

2. The difficulty in writing specifications, especially for pre-written code, as outlined in this report.

3. External factors such as the bugs in the ESC/Java2 tool and the problems and incompleteness of the JML specifications. The success of the project was tightly coupled to these factors.

4. Human error. When planning the work to be carried out, the best decisions were not always made. One important instance of this was not spending sufficient time becoming familiar with the detailed workings of the application's code before attempting to add to or fix the specifications. The time spent learning about the details of the programs workings would have paid off in the long-term due to a better understanding of the program's overall functionalities.

Due to the above issues, but most importantly due to 3, the original project schedule was not adhered to. The ESC/Java2 software bug meant that the core subsystem's classes were not checkable with the latest version of the ESC/Java2 tool, and it was decided instead to first work on some other classes that were almost completely checkable using the latest version.

5 Conclusion

This report has detailed the progress made and the issues encountered during the duration of the project. The ideal goal of the project – to complete the specification and verification of the KOA tally application – was not achieved. There are numerous reasons as to why this was the case, but there should not be too much emphasis placed on the negatives of this project's outcome. There were many benefits gained, just from attempting this task. The ESC/Java2 tool's development was aided by the rigorous testing performed by regularly utilising the latest development version of the tool.

The original development, specification, and verification of the KOA tally system served as a useful case study for researchers working on JML, ESC/Java2, and related tools. Its success showed that these tools can be used effectively when creating correctness-critical software. The continuation of the specification and verification effort, as carried out in this project, should further benefit the Java specification and verification community by pushing forward the boundaries of JML and ESC/Java2 use.

This exercise has been a combination of an experiment and a real-world test for the tools and technologies involved, and is not over yet. It is hoped that the manner in which the KOA tally application's specifications were modified during this project will both allow and encourage the continuation of the effort – whether by this report's author, the original application's authors or any other – to see the completion of the specification and verification of this software to the currently achievable limit.

5.1 Criticisms and Suggestions for the Future

5.1.1 JML

Throughout the duration of this project there were two major shortcomings in the JML field that had a negative impact on progress made:

- JML is sorely lacking a comprehensive guide for beginners. There are numerous tutorials and slides available, but these are either too brief or require a detailed prior knowledge of formal specification writing. Due to the popularity of Java, JML would be an excellent means for programmers to lean about formal methods and their applications. This is only feasible if there is an introduction available aimed at those new to formal methods, that can introduce theory in a coherent, step-by-step fashion with easy to understand examples. This would allow users to learn the basic syntax, terminology and concepts before being bombarded with complex JML models and the full use of the large JML vocabulary.

- The accuracy and consistency of the JML specifications of the Java runtime library is also a problem. Ideally the JML specifications would be guaranteed correct, and this would be checkable. Due to the complexity of specification writing, and especially in writing specifications for priorly written code, this is not the case. It would, however, be more desirable to have a smaller amount of specifications that were correct than to have an abundance of specifications that were possibly inaccurate or inconsistent.

5.1.2 ESC/Java2

Outlined below, are some suggestions and recommendations for the future development of the ESC/Java2 tool:

- Having the suggestion function of this tool return a sensible and coherent idea as to how to fix the problem that has just been detected would be invaluable, especially for beginners. Assuming this feature would not be prohibitively complex to complete, it should have a high priority as a feature to add.

- Improving the counter-example function so that it returns pseudo-code, or similar, that could be much more easily understood by the user, would also be highly useful. If a user could immediately comprehend exactly what the problem with their specification or code was, then their productivity and learning rate would improve.

- The ESC/Java2 plugin development should be continued. Many users prefer to work from a graphical IDE and it is important to appeal to as wide an audience as possible. Graphical output such as the underlining of specific parts of code (or specifications) that cause problems is also especially useful for those who are new to the tool.

6 Acknowledgements

JML would not be as useful or well-defined as it is today without all the hard work given by the developers of the JML technologies and tools, especially to Gary Leavens who took the initiative in starting the development of JML and also oversees JML's current growth and development. For a complete list of the groups and individuals involved in JML see the JML project website [1].

Thanks must also be given to the original authors of the ESC/Java application, the researchers at the Systems Research Centre (K. Rustan M. Leino, Cormac Flanagan, Mark Lillibridge, Greg Nelson, Jim Saxe and Raymie Stata), as well as the developers who have taken the application to its current state as ESC/Java2 (David Cok along with Joseph Kiniry – for a complete list of collaborators see the FAQ on the ESC/Java2 website [15]).

This project would not have come about if it were not for the original development of the KOA tally application. The application's authors (Joseph Kiniry, Engelbert Hubbers and Martijn Oostdijk) demonstrated that it was possible to create efficient and provably correct software even within a very limited timeframe.

Finally, special thanks must be given to this project's supervisor: Joseph Kiniry. Without his mentoring, help, and support, the achievements made in this project would not have been possible. His generous dedication to supporting, developing and promoting JML, as well as his work on the ESC/Java2 tool aptly shows his enthusiasm in the drive for efficient, well-written and most importantly correct, software.

7 References

[1] Many references to papers on JML can be found on the JML project website, http://www.cs.iastate.edu/~leavens/JML/papers.shtml.

[2] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael Ernst, Joseph Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. *An Overview of JML Tools and Applications.* To appear in International Journal on Software Tools for Technology Transfer. Also available from: http://www.jmlspecs.org, Nov. 2004.

[3] David R. Cok and Joseph R. Kiniry. *ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system.* In proceedings for CASSIS 2004. Lecture Notes in Computer Science, volume 3362, Springer-Verlag 2005.

[4] David R. Cok. *Esc/Java2 Implementation Notes*, 2004. Included with all ESC/Java2 releases.

[5] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata.Extended static checking for Java. In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), volume 37, number 5 in SIGPLAN Notices, pages 234–245. ACM, May 2002. Also available from http://citeseer.ist.psu.edu/flanagan02extended.html.

[6] Joseph R. Kiniry. *Electronic and Internet Voting in The Netherlands*. Available from: http://kind.cs.kun.nl/~kiniry/papers/NL_Voting.html.

[7] Joseph R. Kiniry. *The Logics of ESC/Java2*, 2004. Included with all ESC/Java2 source releases.

[8] 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, chapter 12, pages 175-188.

[9] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*. Department of Computer Science, Iowa State University, TR #98-06y, June 1998, revised June 2004. Available from: http://www.jmlspecs.org.

[10] Gary T. Leavens, Yoonsik Cheon. *Design by Contract with JML*. Available from: http://www.jmlspecs.org, Nov. 2004.

[11] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, and J. Kiniry. *JML reference manual.* Department of Computer Science, Iowa State University. Available from: http://www.jmlspecs.org, Nov. 2004.

[12] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. *ESC/Java User's Manual*. Technical Note 2000-002, Compaq Systems Research Center, October 2000. Available from: http://research.compaq.com/SRC/esc/papers.html.

[13] The Eclipse Project Website. http://www.eclipse.org.

[14] The ESC/Java2 Eclipse plugin is available from: http://sort.ucd.ie/projects/escjava-eclipse/.

[15] The ESC/Java2 tool is available from: http://secure.ucd.ie/products/opensource/ESCJava2/.