

ESC/Java2: Uniting ESC/Java and JML

Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system

David R. Cok¹ and Joseph R. Kiniry²

¹ B65 MC01816

Eastman Kodak R & D Laboratories
Rochester, NY 14650-1816, USA
`cok@frontiernet.net`

² Department of Computer Science, University College Dublin,
Belfield, Dublin 4, Ireland***
`kiniry@acm.org`

Abstract. The ESC/Java tool was a lauded advance in effective static checking of realistic Java programs, but has become out-of-date with respect to Java and the Java Modeling Language (JML). The ESC/Java2 project, whose progress is described in this paper, builds on the final release of ESC/Java from DEC/SRC in several ways. It parses all of JML, thus can be used with the growing body of JML-annotated Java code; it has additional static checking capabilities; and it has been designed, constructed, and documented in such a way as to improve the tool's usability to both users and researchers. It is intended that ESC/Java2 be used for further research in, and larger-scale case studies of, annotation and verification, and for studies in programmer productivity that may result from its integration with other tools that work with JML and Java. The initial results of the first major use of ESC/Java2, that of the verification of parts of the tally subsystem of the Dutch Internet voting system are presented as well.

1 Introduction

The ESC/Java tool developed at DEC/SRC was a pioneering tool in the application of static program analysis and verification technology to annotated Java programs [13]. It was a successor to the ESC/Modula-3 tool [22], using many of the same ideas, but targeting a “mainstream” programming language. ESC/Java operates on full Java programs, not on special-purpose languages. It acts modularly on each method (as opposed to whole-program analysis), keeping the complexity low for industrial-sized programs, but requiring annotations on methods that are used by other methods. The program source and its specifications are translated into verification conditions; these are passed to a theorem

*** Formerly with the Security of Systems Group at the University of Nijmegen.

prover, which in turn either verifies that no problems are found or generates a counterexample indicating a potential bug. The tool and its built-in prover operate automatically with reasonable performance and need only program annotations against which to check a program’s source code. The annotations needed are easily read, written and understood by those familiar with Java and are partially consistent with the syntax and semantics of the separate Java Modeling Language (JML) project [1,19]. Consequently, the original ESC/Java (hereafter called ESC/Java) was a research success and was also successfully used by other groups for a variety of case studies (e.g., [16,17]).

Its long-term utility, however, was lessened by a number of factors. First, as companies were bought and sold and research groups disbanded, there was no continuing development or support of ESC/Java, making it less useful as time went by. As a result of these marketplace changes, the tool was untouched for over two years and its source code was not available.

The problem of lack of support was further compounded because its match to JML was not complete, and JML continued to evolve as research on the needs of annotations for program checking advanced. This unavoidable divergence of specification languages made writing, verifying, and maintaining specifications of non-trivial APIs troublesome (as discussed in Section 5).

Additionally, JML has grown significantly in popularity. The activities of several groups [1,3,19,27,28] generated a number of tools that work with JML. Thus, many new research tools worked well with “modern” JML, but ESC/Java did not.

Finally, some of the deficiencies of the annotation language used by ESC/Java reduced the overall usability of the tool. For example, frame conditions were not checked, but errors in frame conditions could cause the prover to reach incorrect conclusions. Also, the annotation language lacked the ability to use methods in annotations, limiting the annotations to statements only about low-level representations.

The initial positive experience of ESC/Java inspired a vision for an industrial-strength tool that would also be useful for ongoing research in annotation and verification. Thus, when the source code for ESC/Java was made available, the authors of this paper began the ESC/Java2 project.

This effort has the following goals: (1) to make the source consistent with the current version of Java; (2) to fully parse the current version of JML and Java; (3) to check as much of the JML annotation language as feasible, consistent with the original engineering goals of ESC/Java (usability at the expense of full completeness and soundness); (4) to package the tool in a way that enables easy application in a variety of environments, consistent with the licensing provisions of the source code release; and (5) as a long-term goal, and if appropriate, to update the related tools that use the same code base (Calvin, RCC, and Houdini [12]) and to integrate with other JML-based tools. This integration will enable testing the tool’s utility in improving programmer productivity on significant bodies of Java source; the tool will also serve as a basis for research in unexplored aspects of annotation and static program analysis.

We have released over seven alpha versions of ESC/Java2. The latest version is available on the [web](#)¹ and we encourage experimentation and feedback. The source code is available (and additional contributors are welcome) and is subject to fairly open licensing provisions. The discussion below of various features of JML and ESC/Java2 is necessarily brief; more detail is available in the implementation notes that are part of an ESC/Java2 release.

The subsequent sections will discuss the most significant changes in creating ESC/Java2, the extensions to static checking, the backwards incompatibilities introduced, unresolved semantic issues in JML, and the direction of the ongoing work in this project. Also discussed is ESC/Java2's first serious use: the verification of portions of the tally subsystem of the Dutch Internet voting system.

Appendices list the details of the enhancements to ESC/Java and those features of JML that are not yet implemented in ESC/Java2. We fully acknowledge that the on-going work described here builds on two substantial prior efforts: the definition of the Java Modeling Language and the production of ESC/Java and the Simplify prover in the first place.

2 JML Example

The Java Modeling language is described in detail in several other publications ([19] and various papers listed at [1]), so here we will give just one example showing some of the syntax. The class in Fig. 1 uses an array to implement a List. A few methods with partial specifications are shown. They demonstrate the following features of JML:

- JML annotations are contained in comments beginning with `//@` or `/*@`.
- `model import` statements declare classes imported for use in annotations.
- `spec_public` indicates that the field named *seq* has public visibility in specifications.
- The `invariant` states that after construction *seq* is never null and is an array with `Objects` as elements.
- The `requires` keyword states a precondition for the *reverse* method, namely its argument is presumed to be non-null.
- The `modifies` keyword states a frame condition for the *reverse* method, namely that the only fields that are assigned during its execution are the elements of the `out` argument.
- The `signals` keyword states a postcondition for the *reverse* method that must hold if an exception is thrown, in this case that it never throws a `NullPointerException`.
- The `ensures` keyword states a postcondition for the *reverse* method that must hold if the method terminates normally.
- The `model` declaration declares a public field used in specifications, typically as an abstract representation of the class. In this case, the class represents a `List`.

¹ <http://www.niii.kun.nl/ita/sos/projects/escframe.html>

```

/*@ model import java.util.List;
/*@ model import java.util.ArrayList;
public class Example {
    /*@ spec_public */ private Object[] seq;
                                /*@ in list;
                                /*@ maps seq[*] \into list;
    /*@ invariant seq != null && \elementype(\typeof(seq)) == \type(Object);

    /*@ requires out != null;
    /*@ modifies out[*];
    /*@ signals (NullPointerException) false;
    /*@ ensures seq.length > 0 ==> out[0] == seq[seq.length-1];
    public void reverse(Object[] out) {
        int i = 0;
        int j = seq.length;
        while (i < seq.length) out[i++] = seq[--j];
    }

    /*@ public model List list;
    /*@ private represents list <- toList(seq);

    /*@ requires input != null;
    @ ensures \result != null;
    @ pure
    @ private model List toList(Object[] input) {
    @   List list = new ArrayList(input.length);
    @   for (int i=0; i<input.length; ++i) list.add(input[i]);
    @   return list;
    @ }
    @ */

    /*@ requires i >= 0 && i < length();
    /*@ modifies list;
    public void insert(int i, Object o) { seq[i] = o; }

    /*@ private normal_behavior
    /*@ ensures \result == seq.length;
    /*@ pure
    public int length() { return seq.length; }
}

```

Fig. 1. A List class with a partial specification.

- The **represents** statement indicates the relationship between the value of the model field and the implementation.
- The next set of declarations constitute a model method declaration and its specifications; a model method is only used in annotations and need not have an implementation.
- The **modifies** clause on the *insert* method indicates that it may modify the value of the model field *list* or any field in its *datagroup*; the **in** and **map** annotations on the declaration of *seq* stipulate that the *seq* field and its array elements are in the *list* datagroup.
- The **pure** modifier on the *length* method indicates that that method has no side effects (it does not assign to any fields).

This class will compile with `javac` and will pass all the checks of the `jml` checker. If it is subjected to the `ESC/Java2` tool described in this paper, three warnings are produced, correctly pointing out three potential problems with this code:

- The default constructor does not set the value of `seq` to a non null array as required by the invariant.
- The assignment to `out[i++]` on line 16 is problematic because the index may be too large for the array; this is fixed by stating that the length of `out` must be equal to the length of `seq`.
- An additional warning on that line indicates that the type of the `out` array may possibly not allow assignments of Object references to its elements.

3 Changes to DEC/SRC ESC/Java

Creating `ESC/Java2` required a number of changes to the `ESC/Java` tool. Here we present the most significant of these.

3.1 Java 1.4

The original work was performed from 1998 to 2000, and Java has evolved since then.² The addition required by Java 1.4 is support for the Java **assert** statement.

JML itself contains a similar assert statement. Hence, the user may make a choice between two behaviors. A Java assert statement may be interpreted simply as another language feature whose behavior is to be modeled. The corresponding behavior is to raise an `AssertionError` exception under appropriate circumstances. Alternatively, a Java assert statement may be interpreted as a JML assert statement. In this case, the static checker will report a warning if the assertion predicate cannot be established. Both alternatives are available through user-specified options.

² In fact, Java 1.5 went beta recently. No work has begun on parsing or statically checking Java 1.5 code. Interested parties are welcome to contact the authors with regard to this topic.

3.2 Current JML

The Java Modeling Language is a research project in itself; hence the JML syntax and semantics are evolving and are somewhat of a moving target (and there is as yet no complete reference manual). However, the core language is reasonably stable. The following are key additions that have been implemented; other changes that relate primarily to parsing and JML updates are listed in the Appendix:

- inheritance of annotations and of `non_null` modifiers that is consistent with the behavioral inheritance of JML;
- support for datagroups and `in` and `maps` clauses, which provides a sound framework for reasoning about the combination of frame conditions and subtyping;
- model import statements and model fields, routines, and types, which allow abstraction and modularity in writing specifications;
- enlarging the use and correcting the handling of scope of ghost fields, so that the syntactic behavior of annotation fields matches that of Java and other JML tools.

In addition, all of the differences between JML and ESC/Java noted in the JML Reference Manual have been resolved.³

3.3 New verification checks

Though all of JML is parsed, not all of it is currently checked. ESC/Java concentrated on checking for possible unexpected exceptions arising from conditions such as null pointers or out-of-bounds array indices, since these did not need annotations to be found; annotations were used, however, to state conditions on method arguments or class fields that would preclude such errors. Thus ESC/Java was capable of checking the pre- and post-conditions of methods as well. However, these could only be expressed at a low-level given the limitations of the ESC/Java input language.

The expanded capabilities of ESC/Java2 allow more thorough checking at a higher level of abstraction. This has required only minor changes in the background axioms used by the theorem prover (mostly regarding primitive types, though additions to handle the semantics of String objects are needed). Most of the changes are implemented by the appropriate translation of JML features into the theorem prover's input logic. The space available in this paper permits only a summary of the embedding of the above into the underlying ESC/Java logic⁴.

Static checking of the following features has been added to that performed by ESC/Java.

³ The tools still differ in (a) the search order for refinement files on the classpath and (b) which methods may be declared as helper methods.

⁴ Subsequent papers are planned that will describe these embeddings in more detail.

The constraint and initially clauses These two clauses are variations on the more common **invariant** clause. They apply to the whole class. A constraint states a condition that must hold between the pre-state and the post-state of every method of a class. For example,

```
constraint maxSize == \old(maxSize);
```

states that **maxSize** is not changed by any method of the class. It is implemented by adding the predicate as a postcondition of every (non-helper) method in the type (and its derived types).

Similarly, **initially** states a condition that must hold of every object after construction. It is implemented by adding its predicate as a postcondition of every (non-helper) constructor of the type (but not of its derived types).

The \not_modified expression The **not_modified** construct is a way of saying, within a postcondition, that a particular expression has the same value in the pre-state and the post-state. That is,

$$\text{\not_modified}(x+y) \equiv ((x+y) == \text{\old}(x+y)) .$$

Uses of the expression in postconditions are expanded inline according to this definition.

Checking of datagroups and frame conditions JML contains syntax to define datagroups [24]. With datagroups, the items in an **assignable** clause may represent sets of program locations, and those sets may be extended by subtypes. Each specification case of a routine may be guarded by a precondition and may specify the set of store locations that may be assigned to.

There are a number of cases to be considered in a full implementation. We will discuss just one here: an assignment statement that has a left-hand side of *expr.field*. For this to be a legal assignment with respect to the specifications, either (a) the *expr* must evaluate to an object that has been allocated since the beginning of the execution of the method, or (b) it must be the case that for every specification case of the method containing the assignment for which the precondition is true (in the pre-state) there is at least one store location in the list of assignable locations that matches *expr.field*. To match, the field names must be the same and the *expr* values must evaluate to the same object. The matching is complicated by the variety of syntax (e.g. *expr.** matches any field of *expr*) and by the fact that a given field designation may have an accompanying datagroup and the match may be to any element of the datagroup. All of this syntax is parsed, and checks are implemented in the logic except where induction is needed to handle recursive definitions.

Recursive definitions of frame conditions (arising from recursive structures such as linked lists and trees) are indeed the most substantial complication in checking datagroups. As an example, consider the datagroup of all of the ‘next’ fields of a linked list. ESC/Java2 currently deals with this by unrolling the recursion to a fixed depth; since in ESC/Java loops are also unrolled to a fixed number of iterations, this solution handles common cases of iterating over recursive structures.

Annotations containing method and constructor calls JML, but not ESC/Java, allows pure method and constructor calls (that is, methods and constructors without side-effects) to be used in annotations. This allows both a degree of abstraction and more readable and writable specifications.

ESC/Java2 supports the JML syntax and also performs some static checking. The underlying prover, Simplify, does support function definitions and reasoning with functions. But, as is the rule in first-order logic, the result of a function in Simplify depends only on its arguments and not on hidden arguments or on global structures referenced by the arguments. Consequently there is a mismatch between the concept of a pure method in Java and the concept of a function in the prover. However, a moderate degree of checking can be performed without resorting to a full state-based translation and logic if we (a) identify some methods as functions, where possible, (b) include the current state of the heap as an additional uninterpreted parameter, and (c) incorporate the specifications of the called method as additional axioms.

Dynamic allocations of objects using constructors are simply static method calls that return new objects and are treated in the same way as other method calls. The logic includes axioms that ensure that a newly allocated object is distinct (reference values are unequal) from any previously allocated object. Dynamic allocations of arrays are translated into first-order logic as functions without difficulty, as they were in ESC/Java.

model fields and represents clauses The combination of **represents** clauses and model fields provides a substantial benefit in abstraction, especially since the representations may be provided by a subtype [8]. Simple representations can be implemented in ESC/Java2 by inlining the representation wherever the model field is used in an annotation. However, that proves not to be workable in larger systems. Instead, we translate instances of model fields as functions of the object that owns them and the global state (because model fields can depend upon fields in other, non-owner, objects). This allows a useful degree of reasoning when combined with the class invariants that describe the behavior of the model fields.

The Simplify theorem prover used by both ESC/Java and ESC/Java2 remains unchanged, except for being compiled for a new platform (Apple's OS X). It is written in Modula-3 and consequently requires compilation for each supported platform. Although the prover has definite limitations, as pointed out below, revising it would be a significant project in its own right.

3.4 Backwards incompatibilities

The ESC/Java specification language and JML arose separately; there was some initial but incomplete work to unify the two [20]. The ESC/Java2 project intends to have the tool reflect JML as precisely as reasonable. In some cases, discussion about differences resulted in changes to JML. In a few cases, some backwards

incompatibilities in ESC/Java were introduced. The principal incompatibilities are these:

- The semantics of inheritance of specification clauses and of `non_null` modifiers was modified to match that defined by JML, since the work on JML resulted in an interpretation consistent with behavioral subtyping. JML has a standalone `also` keyword that indicates there are inherited explicit or implicit specifications; its interpretation of specification inheritance is consistent with behavioral subtyping. By contrast, ESC/Java’s use of inherited specifications had limitations and was a known source of soundness problems [23]. See the section titled “Inheritance and `non_null`” in the ESC/Java2 Implementation Notes for more details [10].
- The specification `modifies \everything` is now the default frame axiom.
- The syntax and semantics of `initially`, `readable_if` and `monitored_by` have changed.
- ESC/Java2 forbids bodies of (non-model) routines to be present in non-Java specification files.

4 Unresolved semantic issues

The work on ESC/Java2 has been useful in exposing and resolving semantic issues in JML. Since ESC/Java2 is built on a different source code base than other JML tools, differences of interpretation in both syntax and semantics arise on occasion. These are generally resolved and documented via mailing list discussions⁵ by interested parties. There are, however, still unresolved issues, most of which are the subject of ongoing research.

- *pure routines*: It is convenient and modular to use model and Java methods within specifications (model methods are methods defined for annotations only and not part of the Java source, such as the *toList* method in Fig. 1). The semantics of such use is clearer and simpler if such routines are *pure*, that is, they do not have side-effects.⁶ This is important when evaluating annotations during execution, since the checking of specifications should not affect the operation of the program being checked. Side-effects also complicate static reasoning. However, some side-effects are always present, such as changes to the stack or heap or external effects such as the passage of time. Some are often overlooked but can be consequential, such as locking a monitor. Others the programmer may see as innocuous, benevolent side-effects, such as maintaining a private cached value or logging debugging information in an output file. An interpretation of the combination of purity and benevolent (or

⁵ See `jmlspecs-interest@lists.sourceforge.net`, `jmlspecs-developers@lists.sourceforge.net`, and `jmlspecs-escjava@lists.sourceforge.net` or the corresponding archives at <http://sourceforge.net/projects/jmlspecs>

⁶ Non-pure methods may be used within annotations in *model programs*, which are not discussed in this paper.

ignorable) side-effects that is suitable for both static and run-time checking and is convenient and intuitive for users is not yet available. (See also the discussion of purity checking in [18].)

- *exceptions in pure expressions*: The expressions used in annotations must not have side-effects, but they may still throw exceptions. In that case the result is ill-defined. A semantics that is suitable for both run-time checking and static verification needs to be established.
- *initialization*: The authors are not aware of any published work on specifying the initialization of classes and objects in the context of JML; initial work formalizing `\not_initialized` was only recently completed for the Loop tool. This task includes providing syntax and semantics for Java initialization blocks, JML’s `initializer` and `static_initializer` keywords, and formalizing the rules about order of initialization of classes and object fields in Java.
- *datagroups*: The `in` and `maps` clauses and the datagroup syntax are designed to allow the specification of frame conditions in a sound way that is extensible by derived classes. We do not yet have experience with the interaction among datagroups, the syntax for designating store locations, and either reasoning about recursive data structures or checking them at run-time.
- *unbounded arithmetic*: Chalin [7] has proposed syntax and semantics to enable specifiers to utilize unbounded arithmetic in a safe way within annotations. Tool support and experience with these concepts is in progress. Axioms and proof procedures will be needed to support this work in static checkers.

There are other outstanding but less significant issues concerning helper annotations, model programs and the `weakly`, `hence_by`, `measured_by`, `accessible` and `callable` clauses.

5 Usage experience to date

The SoS group at the University of Nijmegen, along with other members of the [European VerifiCard Project](http://www.verificard.org/)⁷, has used ESC/Java for several projects. For example, Hubbers, Oostdijk, and Poll have performed verifications of Smart Card applets using several tools, including ESC/Java [17]. Hubbers has also taken initial steps integrating several JML-based tools [16].

These and other VerifiCard projects relied upon the specifications of the Java Card 2.1.1 API written and verified by Poll, Meijer, and others [26]. This specification originally came in two forms: one “heavyweight” specification that used JML models, heavyweight contract specifications, and refinements, and another “lightweight” specification that was meant to be used with ESC/Java and other verification tools like Jack, Krakatoa, and the Loop tool [2,4,25].

Writing, verifying, and maintaining these two specifications was a troublesome experience. Because of limitations of various tools which depended upon the

⁷ <http://www.verificard.org/>

specifications, several alternate forms of specifications were required. Additionally, it was sometimes the case that the alternate forms were neither equivalent nor had obvious logical relationships among them.

This experience was one of the motivators for the SoS group's support of this work on ESC/Java2. Now that multiple tools are available that fully cover the JML language, the incidence of specification reuse is rising and painful maintenance issues are becoming a thing of the past. As a result, early evidence for the success of this transition is beginning to appear.

5.1 Transitional Verifications

First, the specifications of a small case study [5] were updated and re-verified by one of this paper's authors (Kiniry) using ESC/Java2. The original work depended upon "lightweight" JML specifications of core Java Card classes and the verification was performed with ESC/Java and the Loop tool. The re-verification effort used the full "heavyweight" Java and Java Card specifications and was accomplished in a single afternoon by an ESC/Java expert.

Second, several members of the SoS group are contributing to updating the "heavyweight" JML specifications of the Java Card API. As a part of this work, the Gemplus Electronic Purse case study, which has been verified partially with ESC/Java [6] and partially with the Loop tool [5], is being re-verified completely with ESC/Java2 using "heavyweight" specifications.

Finally, recent attempts at verifying highly complex Java code examples written by Jan Bergstra and originally used as stress-tests for the Loop tool have been encouraging. Methods that originally took a significant amount of interactive effort to verify in PVS are now automatically verified in ESC/Java2, much to the surprise of some of the Loop tool authors. This work has caused some re-evaluation of the balance between interactive and automated theorem proving in the SoS group.

5.2 Verification of an Electronic Voting Subsystem

The first major partial verification using ESC/Java2 took place in early 2004.

The Dutch Parliament decided in 2003 to construct an Internet-based remote voting system for use by Dutch expatriates. The SoS group was part of an expert review panel for the system and also performed a black-box network and system security evaluation of this system in late 2003. A recommendation of the panel was that a third party should construct a redundant tally system. Such a second system would ensure a double-check of the election count with an independent system. It was also thought that such an external implementation would provide some third-party review of the original work, as the new implementation would depend entirely upon system design documentation and data artifacts (e.g. candidate and vote files); no source code would be shared, or even seen, by the team implementing the redundant system.

The SoS group bid on the construction of this new system, emphasizing the fact that they would use formal methods (specifically, JML and ESC/Java2) to

specify, test, and verify the tally system. The bid was successful; as a result the SoS group was contracted to write the tally system.

The most challenging aspect of the contract was *not* the use of formal methods. Instead, it was the strict *time requirements* of the contract, as the system was to be used in the upcoming European elections. In particular, the SoS group was asked to construct the vote counting system (henceforth called the KOA system) in approximately *four weeks*, with only three developers.

Development Methodology To approach this problem, the three developers (Dr. Engelbert Hubbers, Dr. Martijn Oostdijk, and the second author) partitioned the system into three subcomponents: file I/O, graphical I/O, and core data structures and algorithms. It was decided that, due to the challenges inherent in full system verification and the restricted time allotted to the project, while all subsystems would be annotated with JML, only the third “core” subsystem (Dr. Kiniry’s responsibility) would be fully elaborated in specification.

Additionally, ESC/Java2 would only be used on the core subsystem. In the allotted time a “best-effort” verification would be attempted, in addition to all other standard software engineering practices. This approach is a standard strategy for lightweight use of formal methods [9].

Table 1. KOA System Summary

	File I/O	Graphical I/O	Core
classes	8	13	6
methods	154	200	83
NCSS	837	1,599	395
Specs	446	172	529
Specs:NCSS	1:2	1:10	5:4

Table 1 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. Assertions were counted by simply counting the number of uses of appropriate core specification keywords (requires, ensures, invariant, non_null, in, set, and modifies).

The size of the code and specifications gives a strong indication of the complexity of the verification effort. Longer methods take significantly more time to specify and verify than short ones. Classes with many methods, on the other hand, do not necessarily take much more time to deal with than shorter classes, as effort is coupled to the complexity of the methods, their specs, and the class invariants (e.g., many short, simple methods are trivial to verify, while one long method might take days).

There is very little inheritance in this system. Visual components all inherit from a top-level **Task** class which implements all state changes in response to

external input, and the I/O classes inherit from an `AbstractObjectReader`, an Apache licensed helper class. Other than that, all classes have no parent (beside `Object`).

Because there is little inheritance and we adopt a closed-system view on the vote tally system (no classloading is permitted), ESC/Java2’s weak strong support for specifying and reasoning about dynamic binding is not an issue.

Specification Coverage and Methodology Unsurprisingly, the GUI portion of KOA is the largest subsystem with the lightest specification coverage, having approximately 1 annotation for every 10 lines of code. The focus of the GUI subsystem specification is a finite state machine that represents the state of the GUI. The state of the KOA application is tightly coupled to this GUI state machine as the vote counting process is highly serialized.

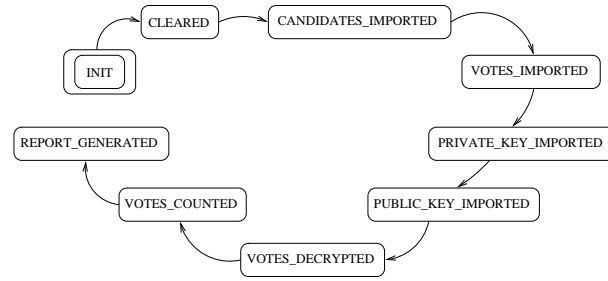


Fig. 2. KOA State Diagram

Figure 2 contains a diagram that summarizes this state machine. The state of the system is represented in a (spec_public) field “`state`” of the main class of the application. The state machine is formally modeled using the standard mechanisms developed in the past by the SoS Group [15].

The file I/O subsystem exhibits better specification coverage, much of which focuses on contracts to ensure that data-structures in the core subsystem are properly constructed according to the contents of input files.

The core subsystem understandably has the highest specification coverage, at over one line of specification for every line of code. This part of the system was designed by contract, and a small-step development process was used throughout (i.e., every time a single line of the specification or the code was changed ESC/Java2 was re-executed). Contractual specifications (e.g., requires/ensures-style and invariants) accounted for the vast majority of the specification; assertions and invariants were only used to assist the verification process.

The verification of the key properties of the system, particularly the property of having a correct tally of votes, are directly tied to the overall state of the system using invariants of the form

```
invariant (state >= <STATE>) ==> (state-field != null);
```

where the states of the system form a total order. Such an invariant says that, if the state of the system is at least `<STATE>`, then the appropriate representation for that state, captured in the `state-field`'s datatype is well-formed. This is a strong claim because if `state-field` is non-null, then not only is it initialized, but all of its invariants hold.

At this time, verification coverage of the core subsystem is good, but not 100%. Approximately 10% of the core methods (8 methods) are unverified due to issues with ESC/Java's Simplify theorem prover (e.g., either the prover does not terminate or terminates abnormally, as discussed below). Another 31% of the core methods (26 methods) have postconditions that cannot be verified, typically due to completeness issues discussed above, and 12% of the methods (10 methods) fail 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 verifies completely.

Since 100% verification coverage was not possible in the timeframe of the project, and to ensure the KOA application is of the highest quality level possible, a large number unit tests were generated with the *jmlunit* tool 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.

Impressions of ESC/Java2 ESC/Java2 made a very positive impression on the KOA developers. Its increased capabilities as compared to ESC/Java, particularly with regards to handling the full JML language, the ability to reason with models and specifications with pure methods, are very impressive. And, while the tool is still classified as an “alpha” release, we found it to be quite robust (perhaps unsurprising given its history, the use of JML and ESC/Java2 in and on its own source code, and the fact that it is passed through seven alpha releases thus far). But there are still a number of issues with ESC/Java2 and JML that were highlighted by the KOA verification effort.

The primary issues that arose include:

- *String semantics in ESC/Java2 are incomplete.* In particular, one cannot reason effectively about String concatenation or equality. While Java Strings are certainly a non-trivial type, they are effectively a pseudo-base type because of their widespread use. Thus, it is vital that this issue be addressed as soon as possible.
- *Issues with reasoning about “representation-less” model variables.* If a class declares a model variable but provides no statement about how that model relates to the implementation of the class (using a `represents` clause or similar), then ESC/Java2 cannot verify assertions that use the model. We believe that a representation-less model variable is equivalent to a ghost variable since it is being used as a specification-only variable in an API spec. Thus, by replacing problematic model variables with ghost variables in API specifications we can successfully perform verifications using the APIs. This

problem indicates either a problem with the design and/or use of model variables in JML, or an implementation issue with ESC/Java2.

- *Inconsistencies and ambiguities in the specification of core APIs, particularly classes in the `java.lang` and `java.util` packages and JML model classes.* This is the first large-scale verification effort using “full” JML specifications of the core JDK. These “full” specifications are much more complete and complex than those that were used with ESC/Java. As these core specifications have never been formally analyzed for consistency or completeness, it is not surprising that the KOA verification effort had problems with their use. It is expected that over time, with more use by a range of JML-compatible tools the core specifications will become more consistent and complete to the benefit of all JML tool users.
- *Completeness issues with first-order predicates.* First-order predicates are expressed with the `forall` and `exists` constructs in JML. Only some of these predicates can be used and/or verified in JML-based tools, including ESC/Java2. Unfortunately, many of the most interesting invariants in non-trivial systems can only be expressed using such constructs. Thus, more focus needs to be put on understanding and reasoning about such assertions.
- *Aliasing issues and specification convenience constructs for `such`.* As usual, reasoning about reference types and avoiding aliasing was one of the key issues in verifying the KOA application. For example, frequently we wished to say that the elements of a set of references were pairwise unequal. The only way to state this in JML today is quite cumbersome, thus the introduction of a new specification construct for `such` seems warranted.
- *The Simplify prover backend and alternate provers.* Simplify is a relatively robust automatic first-order prover, especially given its age and the fact that no one has supported it in many years. Unfortunately, Simplify sometimes fails catastrophically in one of two ways: it crashes due to an internal exception or assertion failure, or it (rarely) consumes as much memory as possible and halts. Neither of these situations is reasonable, of course, but as there is no support for Simplify, the problems indicated by these affects are rarely correctable. It is our intention to initially augment, then eventually replace, Simplify with an alternative modern, supported prover.

6 Ongoing work

The work on ESC/Java2 is continuing on a number of fronts.

Language Issues Two obvious and related ongoing tasks are the completion of additional features of JML, accompanied in some cases by additional research to clarify the semantics and usability of outstanding features of JML. Usage of JML is now broad enough that an accompanying formal reference document would be valuable. As tools such as ESC/Java2 become more widely used, users will also appreciate attention to performance, to the clarity of errors and warnings, and to the overall user experience such tools provide.

Case Studies The current implementation supports the static checking of a stable core of JML. With this initial implementation of frame condition checking, of model fields, represents clauses and use of routine calls in annotations, ESC/Java2 can now be used on complex and abstract specifications of larger bodies of software. Consequently, there is a considerable need for good experimental usage studies that confirm that this core of JML is useful in annotations, and that the operation of ESC/Java2 (and Simplify) on that core is correct and valuable.

Verification Logic The logic into which Java and JML are embedded in both ESC/Java and ESC/Java2, by design and admission of the original authors, neither identifies all potential errors (e.g., because not all aspects of Java are modeled in the logic) nor avoids all false alarms (e.g., because of limitations in the prover). This was the result of an engineering judgment in favor of performance and usability. Research studying expanded and larger use cases may show whether this design decision is generally useful in practical static checking or whether a fuller and more complicated state-based logic is required for significant results to be obtained.

A related issue is the balance between automated and manual proof construction. Use of verification logics will likely be limited to narrow specialties as long as proof construction is a major component of the overall programming task. Thus, automation is essential, though it is expected that full automation is infeasible. The degree of automation achievable will continue to be a research question. However, we believe that broad adoption of automated tools for program checking will require that users only need interact at high levels of proof construction.

User Feedback The purpose of using theorem provers for static analysis, runtime assertion checking, or model checking is to find errors and thereby improve the correctness of the resulting software. Thus, the orientation of a tool must be towards effectively finding *and interpreting* examples of incorrect behavior. A complaint (e.g., [14]) in using such technologies is that it is difficult to determine a root cause from the counterexample information provided by the tool, whether it is a failed proof or an invalid test or execution history. The ESC/Java project implemented some work towards appropriately pruning and interpreting counterexample and trace information [21], but there remains room for improvement. Machine reading of the Simplify output coupled with other tools is also a means to easier interpretation [11].

Tool Integration Finally, though not part of this specific project, an integration of tools that support JML would be beneficial for programmer productivity. A productive programmer's working environment for a large-scale project that uses these tools would need them to be integrated in a way that they seamlessly communicate with one another. A programmer using the tools would naturally move among the various tasks of designing, writing, testing, annotating, verifying and debugging, all the while reading, writing and checking specifications. Design,

specifications and code might all be built up incrementally. Thus, the tools would need to be integrated in a way that allows efficient and iterative behavior.

7 Conclusion

The progress and case studies described above have shown that ESC/Java2 is ready for serious evaluation and use, even in its early “alpha” releases. Our ability to verify large portions of a critical, public system in a very short time frame is a strong statement about the state of the tool and the underlying theory of extended static checking.

Additional evidence comes from several groups around the world that are using ESC/Java2 for instruction and research. We are aware of over a half dozen groups that are using ESC/Java2 for new research in verification, and nearly ten courses are using it for instruction in software engineering, verification, pedagogical instruction of Java, and grading. We continue to see growing interest in ESC/Java2, verification, and extended static checking in general.

One can observe this work in tool creation and evaluation from a number of perspectives. Certainly such work creates working prototypes that test in practice theories of programming and specification language semantics. It also exercises and validates ideas in automated logical reasoning. We prefer to use the viewpoint of programming productivity, particularly given the industrial working environment of the first author. In that context we observe the existence and general use across multiple research groups of the combination of various tools that support using JML with Java programs; this suggests to us that the syntax and semantics of the core of JML are sufficiently useful and natural to provide a basis for future wider use. With respect to logical reasoning, a useful degree of automation is achievable in at least some aspects of static checking tasks; removing the details of proof construction from a programmer’s tasks is essential to larger scale acceptance of such tools.

However, the surrounding issues are as relevant to programmer acceptance and productivity. Tools must have intuitive and unsurprising behavior. They must be efficient in elapsed run-time, but also in the time needed to interpret and act upon the results. They must integrate well with other tools of the same family and with commonly used programmer’s working environments.

There is progress on enough of the above vectors that one might well be optimistic about the eventual success of the enterprise as a whole. After all, the goal need not be fully automated verification of an arbitrary computer program. Reflect that a computer-produced proof of a mathematical conjecture that cannot be understood at least in its broad outlines by mathematicians leaves those mathematicians unsatisfied and unsettled with respect to the proof. Similarly, we expect that “verifications” of programs whose overall design is incomprehensible to readers of the program (not to mention its author) will not engender much confidence in the verification. If programming is writing for others and we expect that the authors could explain their programs to their colleagues, we may well have a chance at being able to explain those programs to a computer.

8 Acknowledgments

The authors would like to acknowledge both the work of the team that developed ESC/Java as well as Gary Leavens and collaborators at Iowa State University who developed JML. In addition, Leavens and students engaged in and helped resolve syntactic and semantic issues in JML raised by the work on ESC/Java2. These teams provided the twin foundations on which the current work is built. Other research groups that use and critique both JML and ESC/Java2 have provided a research environment in which the work described here is useful. Thanks are due also to Leavens for his comments on an early draft of this paper.

Joseph Kiniry is supported by the NWO Pionier research project on Program Security and Correctness and the VerifiCard research project.

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. J. v. d. Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS01, Tools and Algorithms for the Construction and Analysis of Software*, number 2031 in Lecture Notes in Computer Science, pages 299–312. Springer–Verlag, 2001.
3. L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.
4. L. Burdy and A. Requet. JACK: Java applet correctness kit. In *Proceedings, 4th Gemplus Developer Conference*, Singapore, Nov. 2002.
5. J. v. C.-B. Breunese, B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In C. R. H. Kirchner, editor, *Algebraic Methodology and Software Technology*, volume 2422 of *Lecture Notes in Computer Science*, pages 304–318. Springer–Verlag, 2002.
6. N. Cataño and M. Huisman. Formal specification of Gemplus’ electronic purse case study using ESC/Java. In *Proceedings, Formal Methods Europe (FME 2002)*, number 2391 in Lecture Notes in Computer Science, pages 272–289. Springer–Verlag, 2002.
7. P. Chalin. JML support for primitive arbitrary precision numeric types: Definition and semantics. In *Proceedings, ECOOP’03 Workshop on Formal Techniques for Java-like Programs (FTfJP)*, Darmstadt, Germany, July 2003.
8. Y. Cheon, G. T. Leavens, M. Sitaraman, and S. Edwards. Model variables: Cleanly supporting abstraction in design by contract. Technical Report 03-10a, Department of Computer Science, Iowa State University, Sept. 2003. Available from <http://archives.cs.iastate.edu/>.
9. E. Clarke and J. Wing. Strategic directions in computing research: Tools and partial analysis. *ACM Computing Surveys*, 28A(4), Dec. 1996.
10. D. R. Cok. Esc/java2 implementation notes, 2004. Included with all ESC/Java2 releases.
11. C. Csallner and Y. Smaragdakis. Check ’n Crash: Combining static checking and testing. Submitted for publication, 2005.

12. C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. *Lecture Notes in Computer Science*, 2021, 2001.
13. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 234–245, New York, June 2002. ACM Press.
14. A. Groce and W. Visser. What went wrong: Explaining counterexamples. In T. Ball and S. Rajamani, editors, *Proceedings of SPIN 2003, Portland, Oregon*, volume 2648 of *Lecture Notes in Computer Science*, pages 121–135, Berlin, May 2003. Springer-Verlag.
15. E. Hubbers, M. Oostdijk, and E. Poll. From finite state machines to provably correct java card applets. In D. Gritzalis, S. D. C. di Vimercati, P. Samarati, and S. K. Katsikas, editors, *Proceedings of the 18th IFIP Information Security Conference*, pages 465–470. Kluwer Academic Publishers, 2003.
16. E.-M. Hubbers. Integrating Tools for Automatic Program Verification. In M. Broy and A. Zamulin, editors, *Proceedings of the Andrei Ershov Fifth International Conference Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pages 214–221. Springer-Verlag, 2003. <http://www.iis.nsk.su/psi03>.
17. E.-M. Hubbers, M. Oostdijk, and E. Poll. Implementing a Formally Verifiable Security Protocol in Java Card. In D. Hutter, G. Müller, W. Stephan, and M. Ullmann, editors, *Proceedings of the First International Conference on Security in Pervasive Computing*, volume 2802 of *Lecture Notes in Computer Science*, pages 213–226. Springer-Verlag, 2004. March 12–14, 2003, <http://www.dfki.de/SPC2003/>.
18. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: First International Symposium, FMCO 2002, Leiden, The Netherlands, November 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2003.
19. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion, Minneapolis, Minnesota*, pages 105–106. ACM, Oct. 2000.
20. 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>, Apr. 2003.
21. K. R. M. Leino, T. Millstein, and J. B. Saxe. Generating error traces from verification-condition counterexamples. *Science of Computer Programming*, 2004. To appear.
22. K. R. M. Leino and G. Nelson. An extended static checker for Modula-3. In K. Koskimies, editor, *Compiler Construction: 7th International Conference, CC'98*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305. Springer-Verlag, 1998.
23. K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, Oct. 2000.
24. K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 246–257, New York, June 17–19 2002. ACM Press.

25. C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of Java/JavaCard programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1 & 2):89–106, January–March 2004.
26. H. Meijer and E. Poll. Towards a full formal specification of the Java Card. In I. Attali and T. Jensen, editors, *Smart Card Programming and Security*, number 2140 in Lecture Notes in Computer Science, pages 165–178. Springer–Verlag, Sept. 2001.
27. J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings, First Workshop on Runtime Verification (RV’01)*, Paris, France, July 2001.
28. Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. Checking strong specifications using an extensible software model checking framework. Technical Report SAnToS-TR2003-10, Department of Computing and Information Sciences, Kansas State University, Oct. 2003.

A Principal changes to ESC/Java

Language semantics

- inheritance of annotations and of `non_null` modifiers that is consistent with the behavioral inheritance of JML;
- support for datagroups and `in` and `maps` clauses, which provides a sound framework for reasoning about the combination of frame conditions and subtyping;
- model import statements and model fields, routines, and types, which allow abstraction and modularity in writing specifications;
- enlarging the use and correcting the handling of scope of ghost fields, so that the syntactic behavior of annotation fields matches that of Java and other JML tools;

Language parsing

- parsing of all of current JML, even if the constructs are ignored with respect to typechecking or verification;
- support for refinement files, which allow specifications to be supplied in files separate from the source code or in the absence of source code;
- heavyweight annotations, which allow some degree of modularity and nesting;
- auto model import of the `org.jmlspecs.lang` package, similar to Java’s auto import of `java.lang`;
- generalizing the use of `\old`, `set` statements and local ghost variables, to provide more flexibility in writing specifications;
- introduction of the `constraint`, `represents`, `field`, `method`, `constructor`, `\not_modified`, `instance`, `old`, `forall`, `pure` keywords as defined in JML;
- consistency in the format of annotations in order to match the language handled by other JML tools;
- equivalence of `\TYPE` and `java.lang.Class`;
- a beginning of a semantics for String objects, namely the freshness of the result of built-in `+` and equality and inequality of String literals.

In addition, all of the differences between JML and ESC/Java noted in the JML Reference Manual have been resolved.

B Aspects of JML not yet implemented in ESC/Java2

Though the core is well-supported, there are several features of JML which are parsed and ignored, some of them experimental or not yet endowed with a clear semantics, and some in the process of being implemented. For those interested in the details of JML and ESC/Java2, the features that are currently ignored are the following:

- checking of access modifiers on annotations and of the `strictfp`, `volatile`, `transient` and `weakly` modifiers;
- the clauses `diverges`, `hence_by`, `code_contract`, `when`, and `measured_by`;
- the annotations within `implies_that` and `for_example` sections;
- some of the semantics associated with the initialization steps prior to construction;
- multi-threading support beyond that already provided in ESC/Java;
- serialization;
- annotations regarding space and time consumption;
- full support of recursive `maps` declarations;
- model programs;
- some aspects of store-ref expressions;
- verification of anonymous and block-level classes;
- verification of set comprehension and some forms of quantified expressions;
- implementation of `modifies \everything` within the body of routines.