ESC/Java2: Uniting ESC/Java and JML

Progress and issues in building and using ESC/Java2

David R. Cok
457 Hillside Avenue
Rochester, NY 14610
USA

cok@frontiernet.net

Joseph Kiniry
Security of Systems Group
Computing Science Department
University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

kiniry@cs.kun.nl

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The ESC/Java tool was a lauded advance in effective static checking of realistic Java programs, but has become out-of-date with respect to Java and the Java Modeling Language (JML).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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Programming by Contract*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions, invariants, logics of programs, pre- and post-conditions, specification techniques*

Keywords

JML, ESC/Java, ESC/Java2, static analysis, verification, annotation languages, program specification

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 [10]. The tool and its built-in prover operated automatically with

reasonable performance and needed only program annotations against which to check a program's source code. The annotations needed were easily read, written and understood by those familiar with Java and were partially consistent with the syntax and semantics of the separate Java Modeling Language (JML) project [1, 15]. Consequently, the original ESC/Java (hereafter called DEC/SRC ESC/Java) was a research success and was also successfully used by other groups for a variety of case studies (e.g., [12, 13]).

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 DEC/SRC 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 4).

Additionally, JML has grown significantly in popularity. A number of tools have been developed that use JML, representing the work of several groups [3, 1, 20, 19, 15]. Thus, many new research tools worked well with "modern" JML, but DEC/SRC ESC/Java did not.

Finally, some of the deficiencies of the annotation language used by DEC/SRC 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 DEC/SRC 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 DEC/SRC 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 DEC/SRC 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 [9]) 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 currently have an alpha version of ESC/Java2 available on the web[1] and encourage experimentation and feedback. The source code is available (and additional contributors are welcome) 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 principal changes made to DEC/SRC ESC/Java in creating ESC/Java, the extensions to static checking, the backwards incompatibilities introduced, unresolved semantic issues in JML, and the direction of the ongoing work in this project. Appendices list the details of the enhancements to DEC/SRC 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 efforts: the definition of the Java Modeling Language and the production of DEC/SRC ESC/Java and the Simplify prover in the first place.

2. CHANGES TO DEC/SRC ESC/JAVA

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

2.1 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.

2.2 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, virtually all of the differences between JML and DEC/SRC ESC/Java noted in the JML Reference Manual have been resolved.

2.3 New verification checks

Though all of JML is parsed, not all of it is currently checked. Static checking of the following features has been added to that performed by DEC/SRC ESC/Java. The space available in this paper permits only a summary of the embedding of the above into the underlying ESC/Java logic.[3]

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,

```
\not_modified(x+y) ≡ ( (x+y) == \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 [16]. With datagroups, the items in an `assignable` clause may represent sets of program locations, and those sets may be extended by subtypes. Each (of possibly multiple) specification cases 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 spec-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

---

[1]http://www.niii.kun.nl/ita/sos/projects/escframe.html

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

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.

The most substantial complication is that datagroups may be recursively defined and thus may have unbounded size. For 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 routine calls and dynamic allocations. JML, but not DEC/SRC ESC/Java, allows pure method calls 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 provers, the result of a function 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 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 can be recast as method calls and treated as described above. Dynamic allocations of arrays can be translated into first-order logic as functions without difficulty.

model fields and represents clauses. The combination of model fields and `represents` clauses 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. This allows a useful degree of reasoning when combined with the class invariants that describe the behavior of the model fields.

2.4 Backwards incompatibilities

The DEC/SRC ESC/Java specification language and JML arose separately; there was some initial but incomplete work to unify the two. 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 DEC/SRC 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. This also changed the usage and semantics of the `also` keyword.
- 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.

3. 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 discussions[4] 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 annotations. The semantics of such use is clearer and simpler if such routines are *pure*, that is, they do not have side-effects. 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 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 [14].)
- *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

---

[4]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

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.

4. USAGE EXPERIENCE TO DATE

The SoS group at the University of Nijmegen, along with other members of the European VerifiCard Project[5], has used DEC/SRC ESC/Java for several projects. For example, Hubbers, Oostdijk, and Poll have performed verifications of Smart Card applets using several tools, including DEC/SRC ESC/Java [13]. Hubbers has also taken initial steps integrating several JML-based tools [12].

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 [18]. 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 DEC/SRC ESC/Java and other verification tools like Jack, Krakatoa, and the Loop tool [2, 4, 17].

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

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 DEC/SRC 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.

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 DEC/SRC ESC/Java [6] and partially with the Loop tool [5], is being re-verified completely with ESC/Java2 using "heavyweight" specifications.

---

[5]http://www.verificard.org/

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. 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 accompanying formal reference documentation 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 DEC/SRC ESC/Java and ESC/Java2 is, by design and admission of the original authors, neither complete or fully sound. 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., [11]) 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 DEC/SRC ESC/Java project implemented some work towards appropriately pruning and interpreting counterexample and trace information, but there remains room for improvement.

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 the 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.

6. ACKNOWLEDGMENTS

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] 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. Fokkink, 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. Breunesse, 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] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. *Lecture Notes in Computer Science*, 2021, 2001.

[10] 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.

[11] 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.

[12] 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.

[13] 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/.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

APPENDIX

A.   PRINCIPAL CHANGES IN 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; and
- consistency in the format of annotations in order to match the language handled by other JML tools.

In addition, virtually all of the differences between JML and DEC/SRC 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 DEC/SRC ESC/Java;
- serialization;
- annotations regarding space and time consumption;
- full support of recursive `maps` declarations;
- model programs;
- equivalence of `\TYPE` and `java.lang.Class`;
- 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.