

Soundness and Completeness Warnings in ESC/Java2

Joseph R. Kiniry, Alan E. Morkan and Barry Denby
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland

ABSTRACT

Usability is a key concern in the development of verification tools. In this paper, we present an usability extension for the verification tool ESC/Java2. This enhancement is not achieved through extensions to the underlying logic or calculi of ESC/Java2, but instead we focus on its *human interface* facets. User awareness of the *soundness* and *completeness* of the tool is vitally important in the verification process, and lack of information about such is one of the most requested features from ESC/Java2 users, and a primary complaint from ESC/Java2 critics. Areas of unsoundness and incompleteness of ESC/Java2 exist at three levels: the level of the underlying logic; the level of translation of program constructs into verification conditions; and at the level of the theorem prover. The user must be made aware of these issues for each particular part of the source code analysed in order to have confidence in the verification process. Our extension to ESC/Java2 provides clear warnings to the user when unsound or incomplete reasoning may be taking place.

1. INTRODUCTION

ESC/Java2 [7] is a programming tool that attempts to partially verify JML [3] annotated Java programs by static analysis of the program code and its formal annotations. Users can control the amount and kinds of checking that ESC/Java2 performs by annotating their programs with specially formatted comments called pragmas.

In order for the user to have confidence in the verification process, s/he must have confidence in the feedback from the tool. However, ESC/Java2 is neither sound nor complete. ESC/Java2 being unsound means that it emits false positives. That is, it misses errors that are actually present in the program it is analysing. As ESC/Java2 is an extended static checker rather than a program verifier, some areas of unsoundness are incorporated into the checker by design, based

on intentional trade-offs of unsoundness with other properties of the checker, such as efficiency and the frequency of false alarms. ESC/Java2 being incomplete means that it emits false negatives. That is, it warns of potential errors when it is impossible for these error to occur in any execution of the program. Since ESC/Java2 attempts to check program properties that are, in general, undecidable, some degree of incompleteness is inevitable. In addition, the developers of ESC/Java2 were willing to accept some avoidable areas of incompleteness in order to improve performance and to keep the tool simple.

This paper presents an extension to ESC/Java2 that attempts to improve the usability of the tool by providing warnings in cases where the reasoning of the tool is either unsound or incomplete. These warnings should give the user greater confidence in using ESC/Java2.

Unfortunately, such user interaction and feedback is very rarely incorporated in static analysis tools, and in formal methods tools in general. Indeed, there is very little published related work in this field. Many tools are only used by a small community and are not designed for broad adoption, especially across computing disciplines (including students and programmers in industry). In addition, user feedback needs to be “honest”. Although, many tools aim for soundness and a high level of completeness, it is uncommon for them to openly declare to the user the limitations of the tool.

ESC/Java2, on the other hand, is aimed at a broad number of users. It reasons about an established industrial-strength language, detecting common programming errors, while allowing users to determine the amount of checking performed by providing pragmas in a straightforward behavioural specification language (JML)¹. In addition, the extensions described in this paper are inspired by “honesty”. It is essential that the user be aware of the limitations of ESC/Java2, much the same as any verification tool that they use. Finally, efforts to make ESC/Java2 more user friendly are continuous. More details of this can be found in Section 4.

The rest of this paper is organised as follows: Section 2 describes the soundness and completeness of ESC/Java2. A detection and warning system for areas where the reasoning

¹JML is also considered the *de facto* standard specification language for Java

of ESC/Java2 is potentially unsound or incomplete is presented in Section 3. Future work is considered in Section 4 and Section 5 concludes.

2. LIMITATIONS OF ESC/JAVA2

Although ESC/Java2 contains a full Java program verifier, the goal of ESC/Java2 is not to provide formally rigorous program verification. Rather, its aim is to help programmers find some kinds of errors more quickly than they might be found by other methods, such as testing or code reviews. Consequently, ESC/Java2 embodies engineering trade-offs among a number of factors including the frequency of missed errors, the frequency of false alarms, the amount of time used by the tool and the effort required to implement the tool. These trade-offs mean that ESC/Java2 is neither sound nor complete.

It is important to note that, when discussing program verification the words “soundness” and “completeness” are often used imprecisely. Referring to a single concept “soundness” or a single concept “completeness” hides the various layers of each concept that exist in a verification environment. Firstly, there is the soundness and completeness of the underlying logic in which the verification conditions will be generated. Secondly, there is the soundness and completeness of the translation of program constructs into verification conditions. Finally, there is the soundness and completeness of the theorem prover that disposes the verification conditions.

Consequently, we avoid using terminology that can be confusing such as “false positives” and “false negatives”. In this section, we discuss the various instances of unsoundness and incompleteness in ESC/Java2, paying special attention to the category to which it belongs.

2.1 Forms of Unsoundness

This section presents the areas of unsoundness in ESC/Java2 classified according to the underlying cause.

2.1.1 Semantics

There are a number of constructs in Java and JML whose semantics are not treated in a sound manner by ESC/Java2. These are:

Unsound Pragmas

The use of unsound pragmas such as `assume` and `axiom` allow the user to introduce assumptions into the verification process. ESC/Java2 trusts them, assuming them to be true. When these assumptions are invalid, the verification is unsound.

Arithmetic Overflow

ESC/Java2 reasons about integer arithmetic as though machine integers were of unlimited magnitude. This is unsound. However, it simplifies the checker and reduces the annotation burden for the user, while still allowing ESC/Java2 to catch many common errors.

Inherited pragmas

The `also_modifies` and `also_requires` pragma are unsound because they allow an overriding method to have a weaker specification than the method it overrides.

Constructor Leaking

There are a number of ways (often involving exceptional behaviour) in which a constructor can make the new object under construction available in contexts where its instance invariants are assumed to hold, but without actually having established those instance invariants.

Shared Variables

ESC/Java2 assumes that the value of a shared variable stays unchanged if a routine releases and then re-acquires the lock that protects it, ignoring the possibility that some other thread might have acquired the lock and modified the variable in the interim.

String Literals

Java’s treatment of string concatenation is not accurately modeled by ESC/Java2. This is a source both of unsoundness and incompleteness.

2.1.2 Verification Methodology

Additionally, there are a number of ways in which ESC/Java2 does not translate the semantics of the constructs in a Java program into appropriately sound verification conditions.

Loops

ESC/Java2 does not consider all possible execution paths through a loop. It considers only those that execute at most one complete iteration, together with testing the guard before the second iteration. Although this is a straightforward approach and avoids the need for loop invariants, it is unsound.

Object Invariants

When checking the implementation of a method, ESC/Java2 assumes initially that all allocated objects satisfy their invariants. However, on checking a call to a method, ESC/Java2 imposes a weaker condition on the caller. This is that all actual parameters of the call and all static fields that are in scope are shown to satisfy their invariants, but not every object in existence. Since more is assumed than is proven, this is unsound.

In addition, when ESC/Java2 checks the body of a routine r , it does not consider all invariants but only a heuristically chosen “relevant” subset. If an invariant is deemed irrelevant during the checking of a routine that calls r , yet deemed relevant during the checking of r , then the invariant will not be checked (even for parameters) at the call site. However, it will nonetheless be assumed to hold initially during the verification of r . Conversely, ESC/Java2 might consider some invariant to be irrelevant to r , yet relevant to a caller. In this case, ESC/Java2 will not check that the body of r preserves the invariant. Nonetheless, it will assume, while checking the caller, that the invariant is preserved by the call.

Modification Targets

When reasoning about a call to a routine, ESC/Java2 assumes that the routine modifies only its specified modification targets (as given in `modifies` and/or `also_modifies` pragmas). However, when checking the implementation of a method, ESC/Java2 does not check that the implementation modifies only the specified targets.

Multiple Inheritance

When checking a method `m` of a class `C`, which inherits from `A` and `B`, ESC/Java2 assumes that the preconditions of `m` in `A` and `B` hold. However, if a routine `r` contains a call to `m` from an object of dynamic type `C` and static type `A`, then ESC/Java2 will only check the preconditions of `m` in `A`. This is unsound.

Ignored Exceptional Conditions

ESC/Java2 ignores cases where instances of unchecked exception classes (e.g., `OutOfMemoryError`, `StackOverflowError`, `ThreadDeath`, `SecurityException`) might be thrown either synchronously or asynchronously, except by explicit `throw` statements in a routine body being checked or in accordance with the `throws` clauses of routines called by a routine being checked.

Static Initialisation

ESC/Java2 does not perform extended static checking of static initialisers and initialisers for static fields. Neither does it check for the possibility that they do not give rise to errors such as null dereferences, nor does it check that they establish or maintain static or object invariants.

Class paths and .spec files

When a `.spec` file exists on the class path, ESC/Java2 chooses the specifications to check in an unsound manner. If ESC/Java2 is run on `A.java` where `A.spec` also exists, only the specifications in `A.java` are used. If ESC/Java is run on `B.java`, which contains calls to methods in `A.java`, then only the specifications in `A.spec` are used.

Initialisation of Fields declared `non_null`

In the case where a field is declared `non_null`, it may arise that ESC/Java2 uses the existence of a `modifies` pragma in the constructor (or in the specifications of a method called from the constructor) to *assume* that this field is indeed set to a non-null value. However, the `modifies` pragma simply declares what *can* be modified. It does not ensure that the field *is* modified. Therefore, this assumption is unsound.

Quantifiers and Allocation

When `T` is a reference type, specification expressions of the forms `(\forall T t; ...)` and `(\exists T t; ...)` quantify over allocated instances of `T`. If a method allocates new objects but is not annotated with a postcondition containing an occurrence of `\fresh` or `\old`, ESC/Java2 may infer unsoundly that some property holds for all allocated objects after completion of a call, when the property may in fact not hold for objects allocated during the call.

2.1.3 Theorem Prover

Finally, there are areas of unsoundness in Simplify, the main theorem prover currently used by ESC/Java2 [4]. Our work identifying issues with Simplify and warning the user about such will need to be repeated with each new theorem prover that is being added to ESC/Java2. Currently, partial support exists for PVS [10], the SMT-LIB [11] provers Sammy [6] and Harvey [1], and the new CVC3 (a merge of CVC Lite [2] and Sammy), and Coq [5].

Search Limits in Simplify

Simplify sometimes fails to prove the validity of an input formula or provide a counterexample. Such failures happen in a number of different ways. These scenarios are typical of many automated first-order provers.

- **Time Limits.** The first way Simplify can fail is it can simply not find a proof or a (potential) counterexample for the verification condition for a given routine within a set time limit. In this case, ESC/Java2 issues no warnings for the method, even though it might have issued a warning if given a longer time limit. If Simplify reaches its time limit after reporting one or more (potential) counterexamples, then ESC/Java2 will issue one or more warnings, but perhaps not as many warnings as it would have issued if the time limit had been longer.
- **Limit the Number of Warnings.** There is also a bound on the number of counterexamples that Simplify will report for any conjecture, and thus on the number of warnings that ESC/Java2 will issue for any routine. Thus many warnings “early” in a method can result in missing (possibly more serious) problems “later” in the method.
- **Universal Quantifiers.** Additionally, Simplify has problems dealing with (universal) quantifiers. When reasoning about universal quantifiers, Simplify frequently needs “triggers” to guide skolemization. A set of heuristics are used to help guide proof search, but they are not guaranteed to be sound. In particular, Simplify can miss seemingly “obvious” proofs because it moves down a branch of the proof tree and is unable to back-track properly.

These kinds of failures are witnessed in practise because first-order assertions are usually directly translated into first-order terms in verification conditions. Thus, while the quantifiers used in ESC/Java2’s object logics are “well-triggered,” user quantifiers are not. This type of failure must be communicated to the user in a natural manner, so rather than showing a mysterious failure from the prover, ESC/Java2 indicates that the user’s specifications are overly-rich for the current prover and suggests trying other provers.

Prover Failures

Simplify, like many complex programs, also occasionally crashes. When Simplify fails, it is not sufficient to just hide

the crash from the user and report back an incomplete verification, but instead it must try to characterise the failure so that the user can take remedial action by either rewriting specifications or using a different prover.

Arithmetic

The Simplify theorem prover, like many Nelson-Oppen inspired provers [9], includes a decision procedure for linear rational arithmetic based on the simplex algorithm. If integer operations in Simplify's simplex module result in overflows, they will silently be converted to incorrect results. Likewise, if non-linear arithmetic is used in assertions, then Simplify's arithmetic subsystem is not sound. Thus, when potential overflow or non-linear arithmetic expressions are detected by the system, an appropriate warning must be issued.

Other provers that use decision procedures, particularly new SMT-LIB provers, have exactly the same kind of behaviour and require the same kind of warnings. Unfortunately, characterising such prover limitations, especially in the presence of multiple interacting decision procedures, requires intimate knowledge of the prover's design and construction and is sometimes more art than science.

2.2 Forms of Incompleteness

This section presents the areas of incompleteness in ESC/Java2, each classified according to the underlying cause.

2.2.1 Semantics

Many sources of incompleteness in ESC/Java2 stem from the fact that we do not fully capture the semantics of Java and JML in the tool.

Floating-Point Numbers

The semantics for floating-point operations in ESC/Java2 are currently extremely weak. They are not strong enough to prove $1.0 + 1.0 == 2.0$ or even $1.0 != 2.0$.

Strings

The semantics for strings are also quite weak. They are strong enough to prove `"Hello world" != null`, but not strong enough to prove the assertion `s == 'l'` after the assignment `c = "Hello world".charAt(3)`. Also, Java's treatment of string concatenation is not accurately modeled by ESC/Java2.

New, rich, verification-centric specifications of `java.lang.String` are being written to correct this issue. To accomplish this goal, the new specifications heavily directly leverage the sequence theories supported by modern first-order provers. This work was halted when the new specifications pushed the boundaries of Simplify's capability to reason about sequences too far. Thus, the work is on-hold until CVC3 is integrated.

Unspecified Java APIs

Not all of the classes in the Java libraries have full JML specifications. Therefore, reasoning about calls to methods of these classes is incomplete.

Type Disjointness

According to the rules of the Java type system, if two distinct classes S and T are not subtypes of each other, then S and T have no non-null instances in common. The mechanism that ESC/Java2 uses to model the Java type system is sufficient to enforce this disjointness for explicitly-named types, but not for all types (e.g., the dynamic element types of array variables).

Arithmetic Overflow

In order to reduce the likelihood of arithmetic overflow occurring in the prover, ESC/Java2 treats all integer literals of absolute magnitude greater than 1000000 as symbolic values whose relative ordering is known but whose exact values are unknown. Thus, ESC/Java2 can prove the assertions $2 + 2 == 4$ and $2000000 < 4000000$ but not $2000000 + 2000000 == 4000000$.

Reflection

The semantics for reflection is extremely limited. For example, ESC/Java2 can determine that `Integer.class` is a non-null instance of `java.lang.Class`, but not that it is distinct from `Short.class`, or even that it is equal to `Integer.TYPE`.

2.2.2 Verification Methodology

The verification methodology used in ESC/Java2 is also unsound for a number of reasons.

Modular checking

The use of modular checking causes ESC/Java2 to miss some inferences that might be possible through whole program analysis. When translating a method call `E.m(...)`, ESC/Java2 uses the specification of `m` for the static type of `E`, even if it is provable that the dynamic type of `E` at the call site will always be a subtype that overrides `m` with a stronger specification.

2.2.3 Theorem Prover

The verification conditions that ESC/Java2 gives to the Simplify theorem prover are in a language that includes first-order predicate calculus (FOPC) (with equality and uninterpreted function symbols) along with some (interpreted) function symbols of arithmetic.

Since the true theory of arithmetic is undecidable, Simplify is necessarily incomplete. In fact, the incompleteness of Simplify's treatment of arithmetic goes well beyond that necessitated by Gödel's Incompleteness Theorem. In particular Simplify has no built-in semantics for multiplication, except by constants. Also, mathematical induction is not supported.

In addition, FOPC is only semi-decidable. That is, all valid formulas of FOPC are provable, but any procedure that can prove all valid formulas must loop forever on some invalid ones. Naturally, it is not useful for Simplify to loop forever, since ESC/Java2 issues warnings only when Simplify reports (potential) counterexamples. Therefore, Simplify will sometimes report a (potential) counterexample `C`, even when it is

possible that more work could serve to refute C, or even to prove the entire verification condition.

3. WARNING SYSTEM

Clear user feedback is important in any tool that performs static analysis. Given the potential soundness and completeness pitfalls discussed in Section 2, a warning system for such stumbling blocks would be extremely beneficial, especially to new or inexperienced users.

This section presents such a warning system that has been implemented as an extension to ESC/Java2. We describe how constructs, in Java and JML, that ESC/Java2 treats in an unsound or incomplete manner are detected. In addition, we provide examples of the warnings that are emitted.

3.1 General Detection Methodology

We wish to detect many different kinds of contextual soundness and completeness issues. Also, many of these issues exist across code paths within ESC/Java2. As we now support, or are now working on support for, two calculi (weakest precondition and strongest postcondition), the use of an optional dynamic single assignment translation, three different logics, and five different provers, this means that we have at least seventy different code paths for verification. Thus, our detection methodology needs to be reusable across different parameterisations.

Therefore, we decided to implement each detection algorithm as an independent, type- and assertion-aware visitor that walks the fully resolved, typed, and annotated abstract syntax tree (AST).

For a given execution of ESC/Java2 with warnings enabled, each relevant visitor runs in sequence. The visitors are implemented as pure classes, so they do not affect the state of the AST.

Many of these visitors are simply performing type- and assertion-aware pattern matching on fragments of the AST. For example, to detect the use of large integer literals in arithmetic expressions, all the visitor must detect are AST fragments involving binary expressions, checking for one of a finite set of Java binary operators, recursively searching each operator's subexpressions for large Java integer literals.

Some visitors must be more complex, as they involve AST subtrees that are not obviously directly related in the tree. For example, we must examine all the invariants of an entire type hierarchy (including all inherited interfaces) if we wish to check the structure of relevant invariants for a given context.

3.2 ESC/Java2 Soundness Warnings

In the soundness warning system, there are three categories for constructs about which ESC/Java2 does not reason soundly. These are:

1. Constructs that produce warnings in normal user mode.

2. Constructs that produce warnings only in a special warning mode.
3. Constructs that do not yet produce warnings.

3.2.1 Warning User Mode

Currently, the following constructs emit soundness warnings in a special *Warning User Mode*: *Unsound Pragmas*, *Static Initialisation*, *String Concatenation*, *Specification Inheritance*, *Quantifiers* and *Allocation and Search Limits* in *Simplify*.

This set of constructs has been chosen for *Warning User Mode* as they are relatively easy to detect while not occurring so frequently that the warnings displayed to the user would be overwhelming.

The following is an example of the clear and terse warning emitted in the case where the tool detects the initialisation of a static field on line 15 of a class called `Test.java`:

```
Test.java:15 Warning: ESC/Java2 does not
perform extended static checking of static
initialisers.
```

```
static int a = 1;
^
```

3.2.2 Verbose Warning Mode

Some constructs occur too frequently to emit soundness warnings in a normal warning mode. Consequently, there is also a *Verbose Warning Mode* that emits warnings for all constructs that ESC/Java2 treats in an unsound manner.

These additional constructs are: *Loops*, *Object Invariants* and *Arithmetic Overflow*.

As it is a *verbose* mode, the warning messages emitted also give extra information to the user. This includes an extended explanation of the unsoundness and a pointer towards a source of more information including a direct citation to the relevant documentation.

An example of a warning in this user mode is where the tool detects the a loop on line 36 of a class called `Loop.java` is:

```
Loop.java:36: Warning: ESC/Java2 does not
consider all possible execution paths
through a loop.
```

```
for(int i=0, i<n; i++){
^
```

It considers only those that execute at most one complete iteration, plus testing the guard before the second iteration.

This is unsound.

To make ESC/Java2 consider more iterations, use the `-loop` option.

More information can be found in Section 2.4.3 and Appendix C.0.1 of the ESC/Java2 User Manual.

This kind of warning behaviour, one that directly cites relevant detailed documentation, is inspired by Eiffel Software's EiffelStudio IDE which cites relevant sections of Meyers's "Eiffel the Language" and "Object-Oriented Software Construction" texts.

3.2.3 Unimplemented Constructs

Finally, there are some constructs that do not yet emit soundness warnings. These are: Ignored Exceptional Conditions, Constructor Leaking, Initialisation of Fields Declared `non-null`, Class paths with `.spec` files and Shared Variables.

3.3 ESC/Java2 Completeness Warnings

In the completeness warning system, the same three categories apply for constructs about which ESC/Java2's reasoning is incomplete.

3.3.1 Warning User Mode

Currently, the following constructs emit completeness warnings in the *Warning User Mode*: *Large Numbers*, *Reflection* and *Bitwise Operators*.

This set of constructs has been chosen for *Warning User Mode* as they are relatively easy to detect while not occurring so frequently that the warnings provided to the user would be overwhelming.

The following is an example of the clear and terse warning emitted where the tool detects the use of the left shift bitwise operator on line 87 of a class called `Bitwise.java`:

```
Bitwise.java:87: Warning: The semantics
of the left shift operator is incomplete.
```

```
int_a << 2;
^
```

3.3.2 Verbose Warning Mode

Some constructs occur too frequently to emit completeness warnings in a normal warning mode. Consequently, these warnings are output in the *Verbose Warning Mode*. These constructs are: *Floating-Point Numbers*, *Strings* and *Arithmetic Overflow*.

The last warning to be given in *Warning User Mode* is to remind the user of the inherent incompleteness of Simplify. This warning states:

The theorem prover used by ESC/Java2, Simplify, is necessarily incomplete.

This is due to the undecidability and semi-decidability of some of the underlying theories used by Simplify.

Note that the warning message is parameterisable across prover names.

As with the soundness warnings, extra information is given to the user in *Verbose Warning Mode*. An example of such a warning is where the tool detects the use of floating-point numbers on line 64 of a class called `Decimals.java` is:

```
Decimals.java:64: Warning: The semantics
of floating-point operations are
incomplete.
```

```
double d = 1.0 + 2.0;
^
```

They are not strong enough to prove $1.0 + 1.0 == 2.0$ or even $1.0 != 2.0$.

For more information, please see Appendix C.1.1 of the ESC/Java2 User Manual.

3.3.3 Unimplemented Constructs

Finally, there are some constructs that do not yet emit completeness warnings. These are *Type Disjointness* and *Modular Checking*.

4. FUTURE WORK

The most obvious piece of further work to be carried out is the extension of the soundness and completeness warning system to cover more scenarios.

The extensions presented in this paper are ones that should be enabled by default in ESC/Java2. At present, it is only an option that can be switched on. Users that are aware of the myriad of options available in ESC/Java2 are those that are experienced in using the tool. These programmers are probably well-aware of the soundness and completeness issues with the tool and are less likely to write inconsistent specifications. So how do we make the tool more user friendly, especially for beginners, without inundating them with excessive feedback?

One solution lies in the evolution of ESC/Java2 from a command line tool into one element of an Integrated Verification Environment (IVE). The authors are part of the EU MOBIUS Project² and are responsible with others for the development of such an IVE. In such a system, the level of feedback to the user will be configurable, allowing the user to fine-tune the information s/he receives. The environment will also highlight or underline pieces of code that are not reasoned about soundly or completely by ESC/Java2.

Currently all of these visitors, their specifications, and associated unit tests are hand-written. Given the complexity of

²The Mobius Project: <http://mobius.inria.fr/>

the tool and aforementioned growing number of critical code paths through the tool, we believe that *generating* the visitors is a wise next step. We plan on defining a formal language in which one can specify the soundness and completeness limitations of various subsystems and generating the appropriate visitors with specifications, much like we already generate the Java and JML AST classes in ESC/Java2.

Likewise, to better support the rich warning messages discussed in Section 3.2.2, we plan on refining the ESC/Java2 architecture into a new version, integrated with the Mobius IVE, using a literate programming-style [8].

Finally, we imagine that some of the more complex situations we wish to check will necessitate the use of a prover to perform logical reasoning.

5. CONCLUSION

We have presented an extensions to the ESC/Java2 tool that provides useful feedback to the user during the verification process. Indeed, user friendliness of static analysis tools is an area that requires more research. It is one of the complaints of first-time users of ESC/Java2 that the feedback offered by the tool is hard to clearly understand and often overwhelming. One step has now been taken in improving this situation, but more are required.

6. ACKNOWLEDGMENTS

This work is being supported by the European Project Mobius within the frame of IST 6th Framework, national grants from the Science Foundation Ireland and Enterprise Ireland and by the Irish Research Council for Science, Engineering and Technology. This paper reflects only the authors' views and the Community is not liable for any use that may be made of the information contained therein.

7. REFERENCES

- [1] Alessandro Armando, Silvio Ranise, and Michael Rusinowitch. A rewriting approach to satisfiability procedures. *Journal of Information and Computation*, 183(2):140–164, June 2003.
- [2] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *CAV*, Lecture Notes in Computer Science. Springer–Verlag, 2004.
- [3] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, Feb 2005.
- [4] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
- [5] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. *The Coq Proof Assistant User's Guide*. INRIA, Rocquencourt, France, rapport techniques 154 edition, 1993.
- [6] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): Fast decision procedures. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'04 (Boston, Massachusetts)*, volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer–Verlag, 2004.
- [7] Joseph R. Kiniry and David R. Cok. 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 *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer–Verlag, Jan 2005.
- [8] Donald E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Center for the Study of Language and Information, 1992.
- [9] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.
- [10] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer–Verlag.
- [11] SMT-LIB: The satisfiability modulo theories library. <http://goedel.cs.uiowa.edu/smtlib/>.