Using Isabelle/HOL for Static Program Verification in JML4

Patrice Chalin, Perry R. James, George Karabotsos

Dependable Software Research Group,
Dept. of Computer Science and Software Engineering,
Concordia University, Montréal, Canada
{chalin, perry, g_karab}@dsrg.org

Abstract. JML4 is an Integrated Verification Environment (IVE) for JML-annotated Java that builds upon Eclipse's support for Java. Two forms of static verification are provided: Extended Static Checking (ESC) and Full Static Program Verification (FSPV). In this paper we explain how use is made of Isabelle/HOL in support of both ESC and FSPV.

1 Introduction

The Java Modeling Language (JML) is the most popular Behavioral Interface Specification Language (BISL) for Java. JML is recognized by a dozen tools and used by over two dozen institutions for teaching and/or research, mainly in the context of program verification [8]. Tools exist to support the full range of verification from Runtime Assertion Checking (RAC) to Full Static Program Verification (FSPV[1]) with Extended Static Checking (ESC) in between [3].

In this paper we describe JML4, a next generation Eclipse-based Integrated development and Verification Environment (IVE) for Java and JML [5]. In particular, our focus is on how JML4 makes use of Isabelle/HOL both as an Automated Theorem Prover (ATP) and an Interactive Theorem Prover (ITP) in support of ESC and FSPV, respectively. JML4 can output lemmas (corresponding to unproven verification conditions) in theory files whose proof scripts can be completed offline to allow future invocations of ESC to successfully prove the correctness of methods. Simpl is an Isabelle/HOL theory that provides syntax and semantics of an imperative language. As an alternative to ESC, JML4's FSPV outputs a theory file—one lemma per method—encoded in Simpl.

In the following section, we provide background material on Isabelle, Simpl (a logic built atop Isabelle/HOL), and JML. Section 3 presents the functionality provided by JML4, with an emphasis on its two forms support for Static Verification (SV), both of which make use of Isabelle/HOL. ESC makes use of it as an Automatic Theorem Prover (ATP), while FSPV TG makes use of Simpl. Section 4 gives an overview of the architecture of the SV subsystems. A discussion of the FSPV Theory Generator (TG) is given in Section 5. Conclusion and future work are presented in Section 6.

2 Background

2.1 Isabelle

Isabelle is an interactive theorem prover. Its development is lead by Laurence C. Paulson of the University of Cambridge and Tobias Nipkow of Technische Universität München. In fact Isabelle is more than just a theorem prover: it is a theorem proving framework. It provides enough proving machinery to define new logics. This machinery includes Isabelle's meta-logic (Isabelle/Pure), the classical reasoner, and the simplifier. Additionally, existing logics can be extended, thus defining new ones. Isabelle/HOL is just one of these logics defined on top of Isabelle/Pure. As the name suggests, Isabelle/HOL is a realization of High Order Logic for Isabelle. Isabelle/HOL is the most complete of all object logics written for Isabelle so far. This reason among others is why Isabelle/HOL has served as the basis for a number of additional logics. Some of these include the logic for Computable Functions

[1] Contrary to some other authors, we interpret the term "full verification" to imply *interactive* verification.

(Isabelle/HOLCF), and logics for sequential imperative programs with Hoare semantics defined such as Bali and Simpl [12]. Of interest to us are the Hoare-based logics—in particular we are interested in Simpl.

2.2 Simpl

Simpl is one of the most recent theories that deals with the logic of programming to be published in Isabelle's Archive of Formal Proofs [12]. It was published just this past February and was developed by Norbert W. Schirmer of Saarland University. Simpl is a theory for a sequential imperative programming language. Within its boundaries, syntax and big- and small-step operational semantics are defined. A set of Hoare rules for both partial and total correctness is defined, along with their proofs of soundness and completeness. Simpl provides for a shallow embedding of the target language. In other words Simpl is independent of a particular syntax. This simplifies the logic immensely by concentrating on semantics rather than syntactic manipulations. Nonetheless, it is expressive enough for many language constructs that exist in modern programming languages. Some of these include global and local variables, exceptions, abnormal termination, breaks out of loops, procedures, and expressions with side-effects.

A Simpl theory is structured around a state and a Simpl program. The state takes the form of a `hoarestate` which is used to denote the list of variables used in a Simpl program. Hoarestates are build on top of Isabelle's Statespaces, which can be thought of as records with multiple inheritance that make use of locales. The program can be written within a definition, or as is more often done, within a lemma. A program is similar to a Hoare triple where by a precondition is followed by the program definition, which is finally followed by a postcondition. Tactics are defined that automate verification, these are the vcg-step tactic, which performs verification step by step, and vcg, which performs many steps automatically.

2.3 JML

The Java Modeling Language (JML) is a BISL that was designed for use by Java developers having only modest mathematical training [10]. JML's syntax is similar to that of Java, and its annotations are given in specially formatted comments. Leavens, Baker, and Ruby state that JML is "more expressive … than Eiffel and easier to use than VDM and Larch," as it combines the best features of these other approaches. Compared to the Larch BISLs, JML is simpler and easier to understand since its syntax is similar to Java's [9].

JML can document both the interface provided by Java code and its behavior, so it is well suited to documenting detailed designs. Interface specifications include annotations on declarations with extended type information, such as that used by universal types or the non-null type system. The behavioral specifications often specify pre- and postconditions that state properties that should hold when the Java code is executed as well as which—and under what conditions—exceptions may be thrown by the code. In keeping with its goal to allow for incremental adoption, JML allows specifications to range from being detailed and complete to being as little as a single clause giving a single property [7].

Several tools have been developed that process JML-annotated code. These tools provide support for Runtime Assertion Checking (RAC), Extended Static Checking (ESC), Full Static Program Verification (FSPV), automatic discovery of invariants, automated unit testing, and documentation generation [3].

3 JML4

3.1 Overview

A wide variety of tools have been developed that support JML, but many of these are showing their age. For example, neither the JML Compiler, which instruments Java bytecode with Runtime Assertion Checks, nor the ESC/Java2, an Extended Static Checker, provides support for features introduced in Java 5, such as generics.

JML4 is a next-generation tools framework that is based on the Eclipse Java Development Toolkit (JDT). It's goal is to reduce the effort needed to develop new JML tools by providing access to a JML-decorated AST, without requiring the JML community to maintain the underlying Java compiler. While the JDT is large—approximately 1 MLOC for 5000 files—and the learning curve is steep (partly due to lack of documentation) we nonetheless chose to take the plunge and began prototyping JML4 in 2006

In our first feature set, JML4 enhanced Eclipse 3.3 with scanning and parsing of nullity modifiers, enforcement of JML's non-null type system (both statically and at runtime) and the ability to read and make use of the extensive JML API library specifications. This architecturally significant subset of features was chosen so as to exercise some of the basic capabilities that any JML extension to Eclipse would need to support. These include

- recognizing and processing JML syntax inside specially marked comments, both in `*.java` files as well as `*.jml` files;
- storing JML-specific nodes in an extended AST hierarchy,
- statically enforcing a modified type system, and
- generating runtime assertion checking (RAC) code.

The chosen subset of features was also seen as useful in its own right [4], somewhat independent of other JML features. In particular, the capabilities formed a natural extension to the existing embryonic Eclipse support for nullity analysis.

We mention in passing that in parallel with our work on next generation components we have integrated the two main first-generation JML tools: ESC/Java2 and the JML RAC. Hence, at a minimum, JML users actively developing with first generation tools will be able to continue to do so, but now within the more hospitable environment offered by Eclipse.

3.2 Current capabilities

With respect to the next generation components proper, at the time of writing, JML4 parser support is nearing what is called JML Level 1, which includes the most frequently used core JML constructs [11, §2.9]. Basic support for RAC (e.g., inline assertions and simple contracts) is available, but our research group is leading development in static verification components—details of our work to date are given in the next section. Yet others are exploring the integration of new tools for JML. We are hopeful that next-generation components fully processing JML Level 1 specifications will be ready by the fall of 2008.

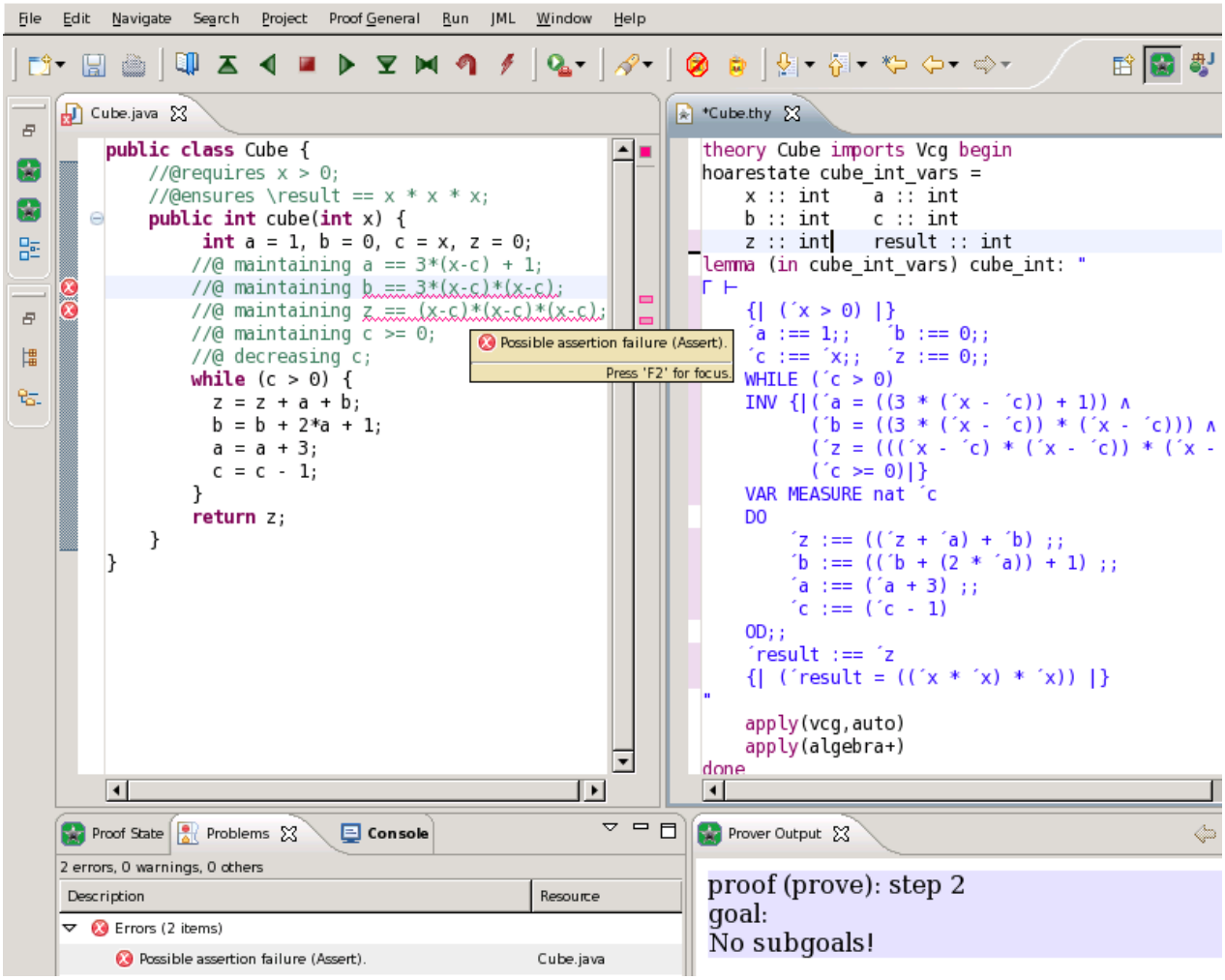


Figure 1. ESC4 reporting that it cannot prove loop invariants in Cube.java (because it is using only first order provers**).** **FSPV TG** theory (Cube.thy) and its proof confirmed valid by Isabelle.

3.3 Static Verification (SV): ground-up designs using latest techniques

Besides work on the JML4 infrastructure, our research group has been focusing its efforts on the development of a new component called the JML Static Verifier (JML SV). This new component offers the basic capabilities of ESC and FSPV.

The ESC component of JML4, referred to as ESC4, is a ground-up rewrite of ESC which is based on Barnett and Leino's innovative and improved approach to a weakest precondition semantics for ESC [2]. Our FSPV tool, called the FSPV Theory Generator (TG), is like the JML LOOP compiler [13] in that it generates theories containing lemmas whose proof establish the correctness of the compilation unit in question. The FSPV TG currently generates theories written in the Hoare Logic of SIMPL—an Isabelle/HOL based theory designed for the verification of sequential imperative programs[12]. Lemmas are expressed as Hoare triples. To prove the correctness of such lemmas, a user can interactively explore their proof using the Eclipse version of Proof General (PG) [1]—see Figure 1.

3.4 Innovative SV features

In addition to supporting ESC and FSPV, the JML SV currently supports the following features, most of which are novel either in the context of verification tools in general or JML tools in particular:

- Multi Automated Theorem Prover (ATP) support including:

  - First-order ATPs: Simplify and CVC3.
  - Isabelle/HOL, which, we have found can be used quite effectively as an ATP.
- A technique we call 2D Verification Condition (VC) cascading where VCs that are unprovable are broken down into sub-VCs (giving us one axis of this 2D technique) with proofs attempted for each sub-VC using each of the supported ATPs (second axis).
- VC proof status caching. VCs are self-contained context independent lemmas (because the lemma hypotheses embed the context) and hence they are ideal candidates for proof status caching. I.e., the JML SV keeps track of proven VCs and reuses the proof status on subsequent passes, matching textually VCs and hence avoiding expensive re-verification.
- Offline User Assisted (OUA) ESC, which we explain next.

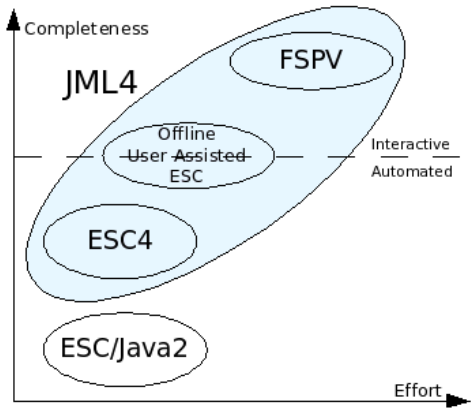


Figure 2. Static verification in JML4

By definition, ESC is fully automatic [6] whereas FSPV requires interaction with the developer. OUA ESC offers a compromise: a user is given an opportunity to provide (offline) proofs of sub-VCs which ESC4 is unable to prove automatically. Currently, ESC4 writes unprovable lemmas to an Isabelle/HOL theory file (one per compilation unit). The user can then interactively prove the lemmas using Proof General. Once this is done, ESC4 will make use of the proof script on subsequent invocations. We have found OUA ESC to be quite useful because ESC4 is generally able to automatically prove most sub-VCs, hence only asking the user to prove the ones beyond ATP abilities greatly reduces the proof burden on users.

Figure 2 sketches the relationship between the effort required to make use of each of the JML SV verification techniques vs. the level of completeness that can be achieved. Notice how ESC4, while requiring no more effort to use than its predecessor ESC/Java2, is able to achieve a higher level of completeness. This is because ESC4 makes use of multiple prover back-ends including the first order provers Simplify and CVC3 as well as Isabelle/HOL. As was mentioned earlier, Isabelle/HOL can be used quite effectively as an automated theorem prover; in fact, Isabelle is able to (automatically) prove the validity of assertions that are beyond the capabilities of the first order provers—e.g., assertions making use of quantifiers. Another example of a method which JML SV can prove correct using Isabelle/HOL as an ATP is Cube.java given in Figure 1 (the reason ESC4 shows that it is unable to prove the loop invariants is because we disabled use of Isabelle/HOL as an ATP for illustrative purposes—to contrast with what can be proven using Cube.thy).

In summary, the latest features added to JML4 make it the first IVE for a mainstream programming language to support the full range of verification technologies (from RAC to FSPV). Its innovative features make it easier to achieve complete verification of JML annotated Java code and this more quickly: preliminary results show that ESC4 will be at least 5 times faster than ESC/Java2. Furthermore, features like proof caching, and other forms of VC proof optimization, offer a further 50% decrease in verification time. Of course, until JML SV supports the full JML language, these results are to be taken as preliminary, but we believe that they are indicative of the kinds of efficiency improvements that can be expected.

In the next section, we explore the architecture of JML4 in general, and the JML SV in particular, allowing us to see how the features described above have been realized.

4 Architecture of JML Static Verifier (SV)

In this section we present an architectural overview of JML4's Static Verifier (rather than the full compiler or other aspects of the IDE).

As was explained above, the JML SV supports two main kinds of verification: extended static checking—both the normal kind and Offline User Assisted (OUA) ESC—and full static program verification. These are realized by the subcomponents named ESC4 and the FSPV Theory Generator (TG), respectively. A diagram illustrating dataflow for the JML SV is given in Figure 3. The input to the JML SV is a fully resolved AST for the compilation unit actively being processed.

Initiated on user request, separate from the normal compilation process, the FSPV TG generates Isabelle/HOL theory files for the given compilation unit (CU). One theory file is generated per CU. The theory files can then be manipulated by the user using Proof General.

When activated (via compiler preferences), ESC4 functionality kicks in during the normal compilation process following standard static analysis. The ESC phases are the standard ones [6], though the *approach* used by ESC4 is new in the context of JML tooling: it is a realization of the Barnett and Leino approach [2] used in Spec# in which the input AST is translated into VCs by using a novel form of guarded-command program as an intermediate representation. The Proof Coordinator decides on the strategy to use: e.g. single prover, cascaded VC proofs or OUA ESC. In the latter case, Isabelle theory files are consulted when sub-VCs are unprovable and a user-supplied proof exists. Unfortunately, the detailed design and full details of the behavior of the JML SV are beyond the scope of this paper.

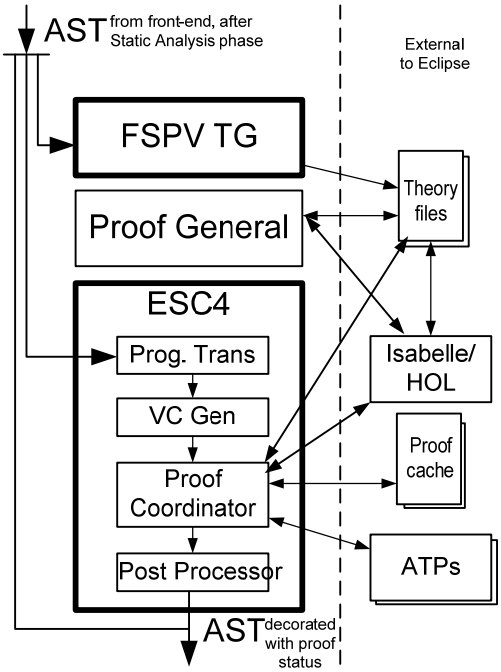


Figure 3. JML SV dataflow

5 Translation to Simpl

The latest addition to the JML4 tooling is the Full Static Program Verification (FSPV) Theory Generator (TG) component. This component is responsible for translating JML annotated Java classes into Simpl theories. Then a user can interactively prove the Simpl theory—i.e. prove the program's correctness. With the advent of Proof Generals's plug-in for eclipse, a user can seamlessly and effortlessly switch between development and verification within a single environment, thus facilitating the use of formal verification techniques on the JML-annotated Java programs.

The FSPV TG has been recently incorporated into the JML tooling—as such it still provides a very limited support for JML and Java. Nevertheless it has provided us with the proof of concept for its incorporation and usage into our tooling.

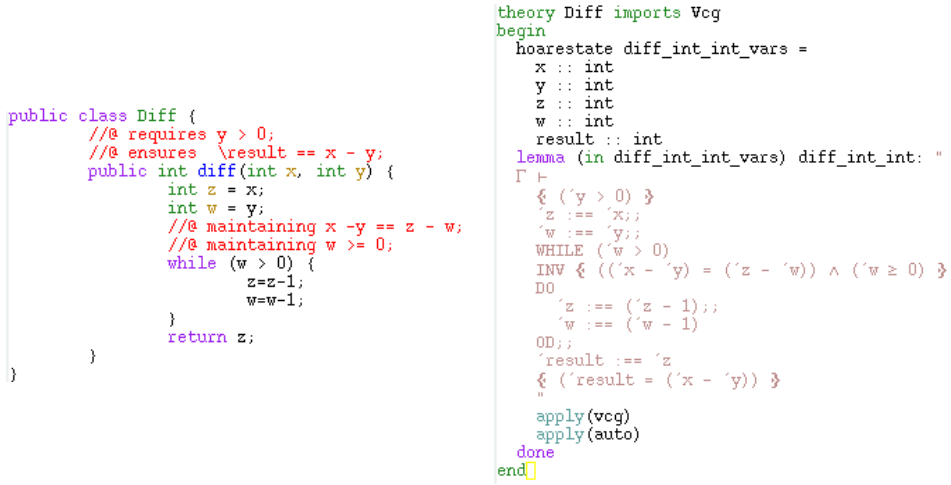
The TG supports a limited but functional subset of JML annotations. This includes basic lightweight contracts with requires and ensures clauses as well as loop annotations with maintaining and decreasing clauses. TG supports a small but functional subset of Java that includes

(1) basic types such as `int`, `boolean`, and `char`,

(2) local variables and method parameters,

(3) expressions with side-effects, such as assignment and binary expressions with most of the common arithmetic and relational operators, and

(4) statements that include if statements, while loops, and local variables with or without initialization.



```
public class Diff {
        //@ requires y > 0;
        //@ ensures  \result == x - y;
        public int diff(int x, int y) {
                int z = x;
                int w = y;
                //@ maintaining x -y == z - w;
                //@ maintaining w >= 0;
                while (w > 0) {
                        z=z-1;
                        w=w-1;
                }
                return z;
        }
}
```

```
theory Diff imports Vcg
begin
  hoarestate diff_int_int_vars =
    x :: int
    y :: int
    z :: int
    w :: int
    result :: int
  lemma (in diff_int_int_vars) diff_int_int: "
  Γ ⊢
    {| (´y > 0) |}
    ´z :== ´x;;
    ´w :== ´y;;
    WHILE (´w > 0)
    INV {| ((´x - ´y) = (´z - ´w)) ∧ (´w ≥ 0) |}
    DO
      ´z :== (´z - 1);;
      ´w :== (´w - 1)
    OD;;
    ´result :== ´z
    {| (´result = (´x - ´y)) |}
    "
    apply(vcg)
    apply(auto)
  done
end
```

Figure 4. JML and Simpl versions of a subtraction routine.

As can be seen in Figure 4, a single theory is created per compilation unit (CU). The theory is named after the CU's file name. For each method in the CU, a single Simpl `hoarestate` and program is generated. The names of both are formed from the method's name and parameter types to disambiguate overloaded methods. The `hoarestate` also contains `_vars` as a suffix. The `hoarestate` collects and defines the parameters and local variables used in the method. The Simpl program is defined within the confines of an Isabelle `lemma` statement. The Simpl pre- and postcondition are created by conjoining and translating the method's requires and ensures clauses, respectively. The Simpl program is a translation of the JML and Java expressions and statements to their corresponding ones in Simpl. Finally applications of the vcg and auto tactics are added to guide the proof of the lemma. It may very well be that these tactics are not appropriate for discharging this lemma—in such case the user should use the appropriate ones in their place. A more complex example of a CU and its corresponding, automatically generated theory can be seen in Figure 1.

Although Simpl supports the definition of local variables we choose not to use them in this initial integration of the TG in JML4. The reason for this was to keep the integration as simple as possible so to facilitate the evaluation of the overall tooling. A better integration is planned in the near future, for more on this we refer you to section 6.

6 Conclusion and Future Work

JML4 is an Integrated Verification Environment (IVE) for JML-annotated Java that is based on Eclipse. It has been built taking into consideration the lessons learned during the development of the first-generation JML tools, as well as taking advantage of advancements of tools for other, related languages. JML4's Static Verification makes available Extended Static Checking that uses multiple automatic theorem provers, including Isabelle/HOL. Its Full Static Program Verification Theory Generator (FSPV TG) relies on Proof General as an interface to Isabelle to prove the correctness of methods as a whole.

There is much work yet to be done. The most pressing is completing the compiler front-end to support all or most of JML. Once this is done, it will be necessary to propagate support for JML constructs into the RAC, ESC and FSPV components.

References

[1] D. Aspinall, "Proof General": http://proofgeneral.inf.ed.ac.uk, 2008.

[2] M. Barnett and K. R. M. Leino, "Weakest-Precondition of Unstructured Programs". *Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, Lisbon, Portugal. ACM Press, 2005.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications", *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212-232, 2005.

[4] P. Chalin and P. R. James, "Non-null References by Default in Java: Alleviating the Nullity Annotation Burden". *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, Berlin, Germany, July-August, pp. 227-247, 2007.

[5] P. Chalin, P. R. James, and G. Karabotsos, “JML4: Towards an Industrial Grade IVE for Java and Next Generation Research Platform for JML”, Concordia University, Dependable Software Research Group Technical Report, 2008.
[6] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended static checking for Java”. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June, vol. 37(5), pp. 234-245. ACM Press, 2002.
[7] T. L. Gary, L. B. Albert, and R. Clyde, “Preliminary Design of JML: A Behavioral Interface Specification Language for Java”, *SIGSOFT Softw. Eng. Notes*, 31(3):1-38, 2006.
[8] G. T. Leavens, “The Java Modeling Language (JML)”: http://www.jmlspecs.org, 2007.
[9] G. T. Leavens, A. L. Baker, and C. Ruby, “JML: A Java Modeling Language”. *Proceedings of the Formal Underpinnings of Java Workshop (at OOPSLA'98)*, October, 1998.
[10] G. T. Leavens, A. L. Baker, and C. Ruby, “JML: A Notation for Detailed Design”, in *Behavioral Specifications of Businesses and Systems*, B. R. Haim Kilov, Ian Simmonds, Ed.: Kluwer, pp. 175-188, 1999.
[11] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin, “JML Reference Manual”, http://www.jmlspecs.org, 2007.
[12] N. W. Schirmer, “Simpl, A Sequential Imperative Programming Language Syntax, Semantics, Hoare Logics and Verification Environment”, in *Isabelle Archive of Formal Proofs*, 2008.
[13] J. van den Berg and B. Jacobs, “The LOOP compiler for Java and JML”. In T. Margaria and W. Yi editors, *Proceedings of the Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, vol. 2031 of *LNCS*, pp. 299-312. Springer, 2001.