Int J Softw Tools Technol Transfer (2010) 12:429–446
DOI 10.1007/s10009-010-0164-8

VSTTE 2008

Towards an industrial grade IVE for Java and next generation research platform for JML

Patrice Chalin · Robby · Perry R. James · Jooyong Lee · George Karabotsos

Published online: 15 June 2010

Abstract Tool support for the Java Modeling Language (JML) is a very pressing problem. A main issue with current tools is their architecture; the cost of keeping up with the evolution of Java is prohibitively high, e.g., Java 5 has yet to be fully supported. This paper presents JMLEclipse, an Integrated Verification Environment (IVE) for JML that builds upon Eclipse's support for Java, enhancing it with preliminary versions of Runtime Assertion Checking (RAC), Extended Static Checking (ESC), Full Static Program Verification (FSPV), and symbolic execution. To our knowledge, JMLEclipse is the first IVE to support such a full range of verification techniques for a mainstream language. We present the original tool architecture as well as an improved design based on use of the JML Intermediate Representation (JIR), which helps decouple JMLEclipse from the internals of its base compiler. As a result, we believe that JMLEclipse is easier to maintain and extend. Use of JIR as a tool exchange format is also described.

P. Chalin (✉) · P. R. James · G. Karabotsos
Dependable Software Research Group (DSRG),
Department of Computer Science and Software Engineering,
Concordia University, Montreal, QC, Canada
e-mail: chalin@dsrg.org; chalin@encs.concordia.ca

P. R. James
e-mail: perry@dsrg.org

G. Karabotsos
e-mail: g_karab@dsrg.org

Robby · J. Lee
SAnToS Laboratory, Department of Computing and Information Sciences, Kansas State University, Manhattan, KS, USA

Robby
e-mail: robby@k-state.edu

J. Lee
e-mail: jlee@cis.ksu.edu

Keywords Program verification · Java · Integrated verification environment

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 [40]. Tools exist to support the full range of verification from Runtime Assertion Checking (RAC) to Full Static Program Verification (FSPV) with Extended Static Checking (ESC) in between [10]. In fact, JML is the only BISL supported by all three of these verification technologies.

Unfortunately, JML tools have been aging quickly. Researchers responsible for JML tool development and maintenance have been unable to keep up with the rapid pace of evolution of both Java and JML. A prime example of this is the lack of support for Java 5, despite the fact that it was released in 2004. Keeping up with changes in Java is very labor-intensive; from an academic researcher's point of view it is unrewarding.

In this article, we present JMLEclipse (formerly JML4), an Eclipse-based Integrated development and Verification Environment (IVE) for Java and JML. Being built on top of the Eclipse Java Development Tools (JDT), JMLEclipse gets up-to-date support for Java (front-end, code generation, etc) almost "for free". Our contributions are as follows:

– After a very brief introduction to JML, we summarize the JML tooling state-of-affairs, reflecting upon lessons learned from the development of the first generation of tools, projecting successes into our statement of *goals* for any next generation tooling infrastructure (Sect. 2).

– With the purpose of illustrating progress made in *achieving these goals* we describe the phase I architecture of JMLEclipse (Sect. 3) which was successfully used to implement preliminary versions of a full range of verification techniques both "natively" (RAC, ESC & FSPV) and via "third party" contributions (constraint programming, symbolic execution and automated test generation). While the implementations of the verification techniques are incomplete, we can already witness the synergies possible with the use of complementary verification techniques within the same tool.
– We present an assessment of the successes and challenges of the phase I architecture (Sect. 4). This serves as a motivation for the phase II work which was greatly motivated by the desire to further decouple JMLEclipse from the JDT internals, making use of public Application Programming Interface (API) instead.
– The main element of the phase II redesign is the creation and use of the JML Intermediate Representation (JIR) which not only allows us to reduce the coupling between JMLEclipse and the JDT internals, but also serves as a tool/component exchange format (Sect. 5). A summary of the phase II architecture, the use of JIR and a brief mention of future plans which promise to further decouple JMLEclipse from the JDT are covered in Sect. 6.
– Finally, an overall assessment of the current state of JMLEclipse is provided. In particular, we reassess the goals and identifying risk items (Sect. 7).

The capabilities of verification tools supporting JML as well as other languages are reviewed in the section on related work (Sect. 8). We conclude in Sect. 9. In the next section, we provide a brief introduction to JML for readers who are unfamiliar with the notation.

2 Java Modeling Language

As was mentioned in the introduction, JML is a Behavioral Interface Specification Language (BISL) [57] for Java. It extends Java with support for Eiffel-like Design by Contract (DBC) [41,44]. Hence, the required behavior of methods like `Counter.inc()` in Fig. 1 can be expressed in the form of a method contract which identifies the conditions which callers must respect (preconditions) and conditions which the given method promises to uphold (postconditions), provided the precondition is respected.

In JML, method preconditions and postconditions are expressed using requires and ensures clauses, respectively. The contracts of `Counter.inc()` and `Counter.getCount()` are in fact examples of what are referred to as lightweight contracts. In contrast, the Counter constructor illustrates a heavyweight contract. A heavyweight contract consists of a series of one or more behavior cases preceded

```
/**
 * Counters that count up to MAX and
 * then wrap back to 0.
 */
public class Counter {

public final static int MAX = 100;

    /*@spec_public*/ private int count;
    //@ invariant 0 <= count && count <= MAX;

    /*@ public normal_behavior
      @   requires 0 <= count && count <= MAX;
      @   assignable this.count;
      @   ensures this.count == count;
      @ also public exceptional_behavior
      @   requires !(0 <= count && count <= MAX);
      @   assignable \nothing;
      @   signals (IllegalArgumentException e) true;
      @   signals_only IllegalArgumentException;
      @*/
    public Counter(int count) {
     if (!(0 <= count && count <= MAX))
       throw new IllegalArgumentException();
     this.count = count;
    }

    //@ ensures \result == count;
    /*@pure*/ public int getCount() {
      return count;
    }

    /*@ requires count < MAX;
      @ ensures  count == \old(count) + 1;
      @ also
      @ requires count == MAX;
      @ ensures count == 0;
      @*/
    public void inc() {
      count = count < MAX ? count + 1 : 0;
    }
}
```

Fig. 1 Sample JML specification of a Counter class

by a visibility modifier that establishes the visibility of the behavior case. Having the ability to set the visibility allows JML developers to express public API contracts that differ (generally are more abstract) from protected or public contracts.

The `Counter.getCount()` method illustrates another feature of JML: Java methods can be used in specifications, but only if they are free of side effects. Developers identify such methods by marking them as pure. As with DBC, JML also supports class invariants. Technically, an invariant is a `Boolean` expression that is required to be true in all (client visible) states of a program's execution [43].

Finally, we note that JML goes well beyond the features of DBC [18] to support, for example:

– Method contracts constraining the behavior of methods when exceptions are thrown.
– Frame properties (illustrated by the assignable clause in the `Counter` constructor contract).
– Model fields.

An example of the latter is the `Counter.count` field. From a Java perspective, it is hidden from clients, but from a JML perspective it is made visible as a specification-only

field. Thus clients can use it to reason about the behavior of `Counters`.

3 JMLEclipse: inception and early elaboration phases

3.1 Inception phase

One of the key initial activities of the project's inception phase was a careful analysis of the JML tooling state-of-affairs, the results of which we describe next.

First Generation Tools: duplication of effort & high (collective) maintenance overhead JML can be seen as an extension to Java that adds support for Design by Contract (DBC), though it has more advanced features as well—such as specification-only class attributes, support for frame properties (indicating which parts of the system state a method must leave unchanged), and behavioral subtyping—that are essential to writing complete interface specifications [18]. The chief first generation JML tools essentially consist of the:

- Common JML tool suite[1] also known as JML2, which includes the JML RAC compiler and JmlUnit [10],
- ESC/Java2, an extended static checker [24], and
- LOOP and PVS tool pair which supports full static program verification [55].

Of these, JML2 is the original JML tool set. Although ESC/Java2 and LOOP initially used their own annotation languages, they rapidly switched to JML.

Being independent development efforts, each of the tools mentioned above has its own front-end (e.g. scanner, parser, abstract syntax tree (AST) hierarchy and static analysis code) essentially for *all* of Java and JML. This amounts to substantial duplication of effort and code. Recent evolution in the definition of Java (e.g. Java 5, especially generics) and of JML made it painfully evident that the limited resources of the JML community could not cope with the workload that it engendered.

As a result, for example, none of the current first generation tools yet fully supports Java 5 features. With respect to the evolution of JML, only JML2 fully supports the new non-null by default semantics [17].

Moving Forward with Lessons Learned What lessons can be learned from the development of the first generation of tools, especially JML2 which, since early 2000, has been the reference implementation of JML? JML2 was essentially developed as an extension to the MultiJava (MJ) compiler.

By "extension", we mean that: for the most part, MJ remains independent of JML; many JML features are naturally implemented by subclassing MJ features and overriding methods; in other situations, extension points (calls to methods with empty bodies) were added to MJ classes so that it was possible to override behavior in JML2. We believe that this approach has allowed JML2 to be successfully maintained as the JML reference implementation until recently.Then what went wrong? We believe it was a combination of factors including the advent of a relatively big step in the evolution of Java (including Java 5 generics) and the difficulty in finding developers to upgrade MJ.

Goals for Next Generation Tool Bases Keeping in mind that we are targeting mainstream industrial software developers as our primary user base, our goals for a next generation research vehicle for the JML community can be summarized as follows: the new tooling infrastructure should be

(1) based on, at least a Java compiler, ideally a modern IDE, whose maintenance is assured by developers *outside* the JML community;
(2) built, to the extent practicable, as an "extension" of the base so as to *minimize the integration* effort required when new versions of the base compiler/IDE are released;
(3) capable of supporting *at least* the integrated capabilities of RAC, ESC, and FSPV

As will be discussed in the section on related work, a few recent JML projects have attempted to satisfy these goals. In the sections that follow, we describe how we have attempted to satisfy them in our design of JMLEclipse.

Early Prototype After much discussion, both within our own research group and with other members of the JML community, we decided that basing a next generation JML tooling framework on the Eclipse JDT seemed like the most promising approach. While the JDT is large—approximately 1 MLOC for 5000 files—and the learning curve is steep (partly due to lack of documentation), DSRG researchers nonetheless chose to "take the plunge" and began prototyping JMLEclipse in 2006.

In our first feature set, JMLEclipse enhanced Eclipse 3.3 with: (a) scanning and parsing of nullity modifiers (`nullable` and `non_null`), (b) enforcement of JML's non-null type system (both statically and at runtime) [15], and (c) 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

[1] Formerly the Iowa State University (ISU) JML tool suite.

– 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 Java 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 [15], somewhat independent of other JML features. In particular, the capabilities formed a natural extension to the existing embryonic Eclipse support for nullity analysis.

This early prototype served as a basis for analysis by members of the JML Reloaded "subcommittee" of the JML Consortium. In conclusion, the decision was to move forward with development of JMLEclipse.

3.2 Early elaboration phase

In this section, we describe the mid-2008 JMLEclipse feature set as evidence that the goals stated in Sect. 3.1 are being met, especially with respect to framework capabilities in support of the full range of verification technologies.

3.2.1 Feature set for the full range of verification

Front-end capabilities (and first-generation tools) 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 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.

With respect to the next generation components proper, JMLEclipse's front-end supports what are called JML Level 0, Level 1, and most of Level 2 features [43, §2.9]. Level 0 to 2 cover all essential JML features. The syntactic elements that remain are in Levels 3, C and X which cover quite infrequently used JML features, support for Concurrency and eXperimental features, respectively [43, §2.9]. The JMLEclipse front-end thus provides, to the other components of JMLEclipse, the capabilities of a type checker similar to that JML2's `jmlc` command.

Runtime Assertion Checking (RAC) While basic support for RAC (e.g., inline assertions and simple contracts and invariants) is available, a next generation design inspired from the current JML2 compiler is being lead by its original author [19], Yoonsik Cheon, and his team at the University of Texas at El Paso.

A key element of Cheon's approach to RAC is the use of wrapper methods in which each method implementation is replaced by a wrapper method of the same name. This wrapper method is responsible for calling a host of other RAC methods created for the purpose of checking class invariants, the method precondition and postcondition, etc.

Static Verification: ground-up designs using latest techniques Besides work on the JMLEclipse infrastructure, the DSRG has been focusing its efforts on the development of a new component called the JML Static Verifier. This new component offers early versions of ESC and FSPV. In this section, we provide an overview of the capabilities of the JML Static Verifier, details of its architecture will be given in Sect. 3.3.

The ESC component of JMLEclipse, 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 [6]. Our FSPV tool, called the FSPV Theory Generator, is like the JML LOOP compiler [55] in that it generates theories containing lemmas whose proofs establish the correctness of the compilation unit in question. The FSPV Theory Generator currently produces theories written in the Hoare Logic of Simpl—an Isabelle/HOL based theory designed for the verification of sequential imperative programs [50]. 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 [2]—see Fig. 2.

3.2.2 Static verification features

In addition to supporting ESC and FSPV, the JML Static Verifier component currently supports the following features:

– Multi Automated Theorem Prover (ATP) support including: first-order ATPs (like Simplify and CVC3), and Isabelle/HOL, which, we have found can be used quite effectively as an ATP. While the Why based verifiers Krakatoa and Caduceus also have multi-prover support [31], the JML Static Verifier was recently enhanced with experimental *distributed* multi-prover support [35,37].
– 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 (and sub-VCs) are self-contained, context-independent lemmas (because the lemmas' hypotheses embed their context), and hence they are ideal candidates for proof status caching. That is, the JML Static Verifier 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.

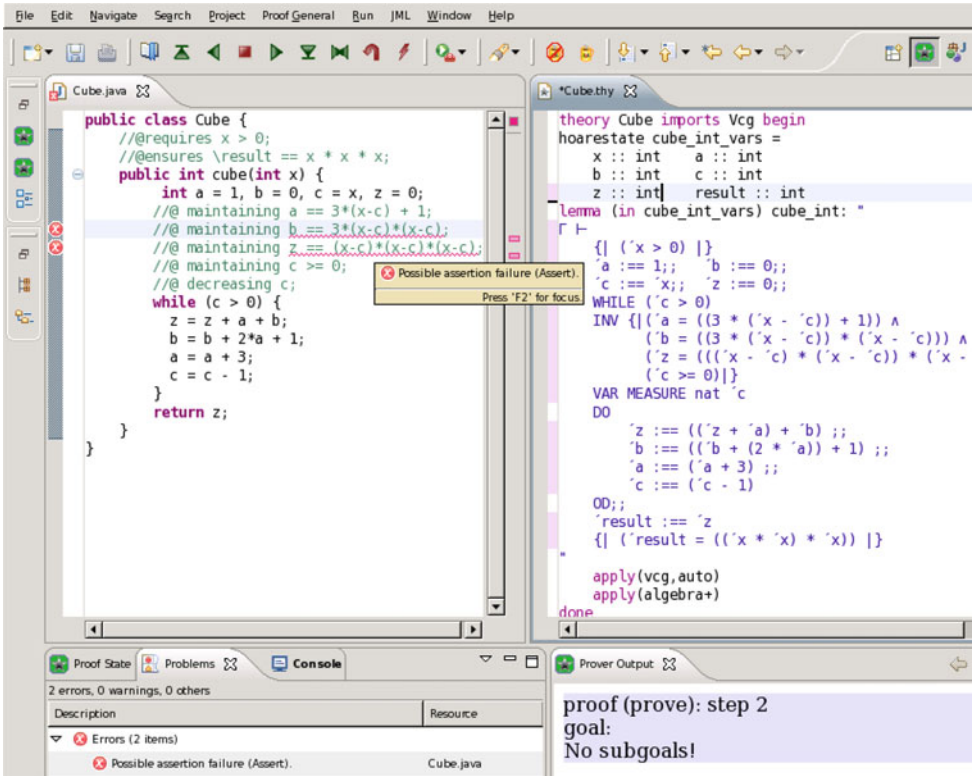

Fig. 2 ESC4 reporting that it cannot prove loop invariants in Cube.java; FSPV Theory Generator Cube.thy theory and its proof confirmed valid by Isabelle

By definition, ESC is a static verification technique that is fully automatic [33], whereas FSPV requires interaction with the developer. OUA ESC offers a compromise: a user is given the 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 [3]. 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 3 sketches the relationship between the effort required to make use of each of the JMLEclipse Static Verifier verification techniques and 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. An example of a method which JML Static Verifier can prove correct using Isabelle/HOL as an ATP is `Cube.java` given in Fig. 2 (the reason ESC4 shows that it is unable to prove the

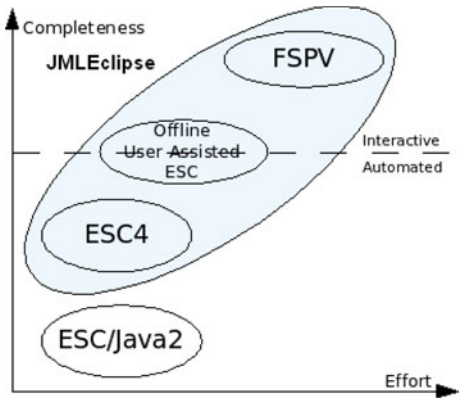

Fig. 3 Static verification in JMLEclipse

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`).

With its current feature set, we believe that JMLEclipse is the first IVE for a mainstream programming language to support the full range of verification technologies (from RAC to FSPV), albeit, preliminary.[2] Its innovative features make it easier to achieve complete verification of JML annotated Java code and this more quickly; initial results show that ESC4 will be at least five times faster than ESC/Java2.

[2] We also note that the proof system used by the Static Verifier component has yet to be proven sound.

Furthermore, features like proof caching, and other forms of VC proof optimization, offer a further 50% decrease in verification time. Better yet, an experimental feature providing distributed multi-prover verification promises improvements that are linear in the number of computing resources used in the verification [35,37]. Of course, until JML Static Verifier supports the full JML language (and certainly research challenges remain before this can be achieved [42]), 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.

3.3 Initial architecture

In this section, we present an architectural overview of JMLEclipse with a particular focus on the compiler (rather than other aspects of the IDE) which is referred to as the JMLEclipse *core*.

3.3.1 Overview

At the heart of JMLEclipse is the JMLEclipse core, whose processing phases are illustrated in Fig. 4. Most phases are conventional. In the Eclipse JDT (and hence in JMLEclipse), there are two types of parsing: in addition to the usual full parse, there is also a diet parse, which only gathers class signature information and ignores method bodies. JMLEclipse-specific phases are shown in bold and include the merge of external specifications and static verification. Code instrumentation for the purpose of Runtime Assertion Checking (RAC) is done during the JDT's code generation phase.

A top-level module view of Eclipse and JMLEclipse is given in Fig. 5. Eclipse is a plug-in based application platform and hence an Eclipse application consists of the Eclipse plug-in loader (Platform Runtime component), certain common plug-ins (such as those in the Eclipse Platform package) along with application specific plug-ins. While Eclipse is written in Java, it does not have built-in support for Java. Like all other Eclipse features, Java support is provided by a collection of plug-ins referred to as the Eclipse Java Development Tools (JDT).

The JML JDT extends the Eclipse JDT to offer basic support for JML. In particular, the JML JDT contains a modified scanner, parser and Abstract Syntax Tree (AST) hierarchy. The JML Static Verifier (SV) component design is described next.

3.3.2 JML Static Verifier

As was explained in the previous section, the JML Static Verifier supports two main kinds of verification: extended static checking—both the normal kind and Offline User-Assisted (OUA) ESC—and full static program verification.

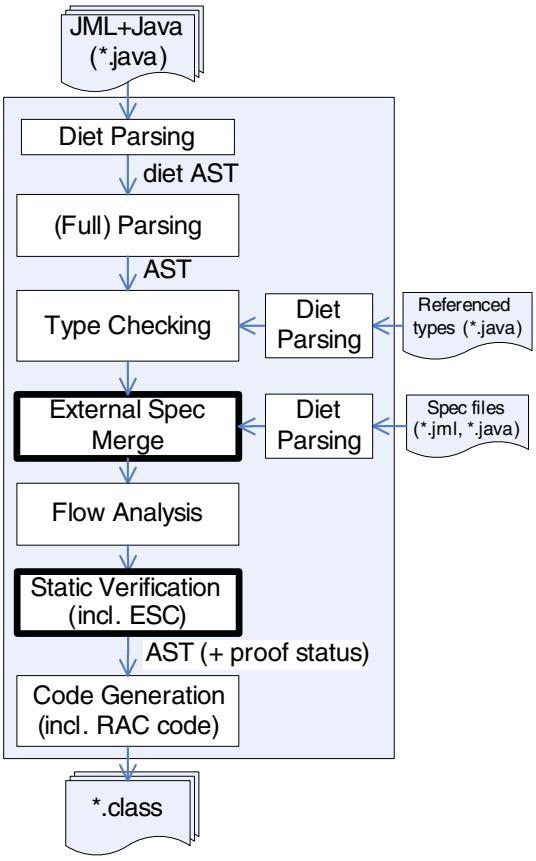

Fig. 4 JDT/JMLEclipse core phases (phases in *bold* are JMLEclipse-specific)

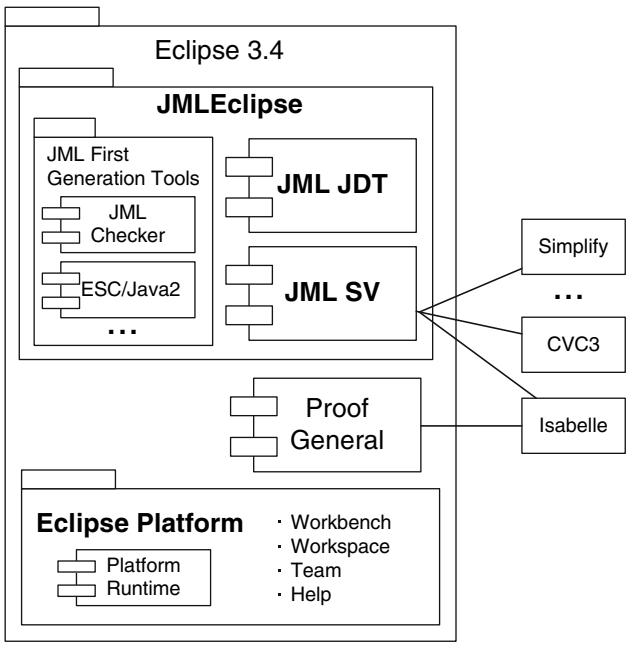

Fig. 5 Eclipse and JMLEclipse component diagram

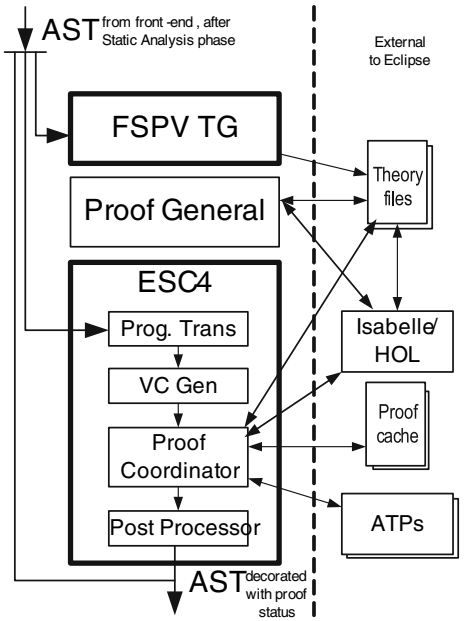

Fig. 6 JMLEclipse Static Verifier dataflow

These are realized by the subcomponents named ESC4 and the FSPV Theory Generator, respectively. A diagram illustrating data flow for the JML Static Verifier is given in Fig. 6. The input to the JML Static Verifier is a fully resolved AST for the compilation unit actively being processed.

Initiated on user request, separate from the normal compilation process, the FSPV Theory Generator produces Isabelle/HOL theory files for the given compilation unit (CU). One theory file is generated per CU. Users can then manipulate the theory files by using Proof General.

When activated (via compiler preferences), ESC4 functionality is activated during the normal compilation process following standard static analysis. The ESC phases are the standard ones [33], though the *approach* used by ESC4 is new in the context of JML tooling: it is a realization of the Barnett and Leino approach [6] 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 such as using a single prover, cascaded VC proofs, OUA ESC and/or a distributed combination of these [35,36]. In the case of OUA ESC, 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 Static Verifier are beyond the scope of this paper.

3.4 "Third Party" features

In addition to providing basic support for JML, JMLEclipse has as a prime objective to be a research platform for other research groups. That is, the success of JMLEclipse will be measured, in part, by how easy it becomes for researchers (other than those working on the JML JDT core) to extend JMLEclipse.

To this end, we have some encouraging signs of success. In addition to Yoonsik Cheon's research group adding RAC support to JMLEclipse, others have built upon JMLEclipse in novel ways including:

- Robby and other SAnToS laboratory members[3] from Kansas State University make use of JMLEclipse as a front-end to the Sireum/Kiasan symbolic execution system and the associated KUnit test generation framework [27].
- Tim Wahls has extended JMLEclipse to enable the execution of specifications through the use of constraint programming [13,38].

4 Initial architectural assessment and phase II redesign

4.1 Successes and challenges

Both the teams of Tim Wahls and Robby essentially worked independently of the JMLEclipse team during the development of their respective JMLEclipse extensions. We believe that this provides an initial indication of the viability of using JMLEclipse as a research platform.

On the other hand, as core JMLEclipse developers, we have come to realize the extent to which the Eclipse JDT is a sophisticated multi-threaded incremental compiler specifically designed to offer effective, e.g., as-you-type well-formedness (e.g., syntax and type) checking to developers. Coming to terms with such an approach, as compared to that of the classical compiler technology that first generation JML tools were built on, has been one of the greatest challenges that we faced while extending the JDT.

JMLEclipse work has been focused on core functionality (e.g. JML language checker, RAC, ESC) as opposed to IDE features proper. We came to realize that offering effective well-integrated IDE support for features, like as-you-type syntax checking and specification refactoring, would require reworking of the JMLEclipse front-end.

4.2 JMLEclipse phase II architectural redesign

This anticipated need to re-architect the JMLEclipse font-end coupled with a closer analysis of the requests of "third party" JMLEclipse projects, has led to a second phase of JMLEclipse research jointly carried out by the DSRG and SAnToS laboratory.

[3] Initially working independently from the original JMLEclipse developers, SAnToS and DSRG have since then become joint contributors to the tool's phase II design as will be explained in the next section.

The main objectives of this second phase redesign of JMLEclipse has been to:

- make the JMLEclipse core minimal, by:
  - factoring out some of the JML specific front-end code, and
  - pulling features such as RAC and ESC out of the core;
- offer non-core components access to fully resolved ASTs built from the JDT's *public* AST class hierarchy—called the DOM—rather than internal AST classes.

As compared to internal JDT packages, the DOM, like all public Eclipse API packages, is considerably less subject to change and is much better documented. The later is a net advantage for JMLEclipse third-party developers.

The principal mechanism by which JMLEclipse is achieving these new design goals is through the use of a carefully designed JML Intermediate Representation (JIR) [48], which not only provides a means of representing JML via the public JDT DOM, but it also offers a standard mechanism for embedding JML specification elements in class files. Our JIR is detailed in the next section before JMLEclipse's revised architecture is presented in Sect. 6.

5 JML Intermediate Representation (JIR)

The JIR not only offers a means of decoupling non-core JMLEclipse components from the internals of the Eclipse JDT, it also serves the more useful purpose of being an exchange format for JML tool components.

Next, we present the JIR design goals, JIR encoding and the JIR infrastructure API used to abstract away, to the extent possible, from the details of the JIR encoding. The work reported in this section builds upon an early JIR design given in [48].

5.1 Design goals and rationale

In a broad sense, our JML intermediate representation is mainly aimed at providing a unified representation for JML that can be easily generated by various JML front-ends and consumed by different JML back-ends. Below are design goals and corresponding rationale that we believe any intermediate representation and its supporting infrastructure should satisfy:

D1 Low barrier of (re-)entry: To help ease adoption, JML tool developers should be able to learn JIR without significant investment, and to use its tool support without a significant learning (and maintenance) overhead.

D2 Comprehensive: It should be possible to represent all JML constructs in JIR.

D3 Extensible: JIR can be easily extended to handle custom JML constructs for experimentation with new language features.

D4 Implementation-independent: JIR should not be tied into a particular JML front-end, nor should it be biased toward a particular analysis technique (e.g., static or runtime assertion checking) or back-end.

D5 Robust tool-support for processing: JIR tool support should be based on stable and robust software infrastructures. In order not to compromise the robustness of the underlying infrastructures, the JIR code base should be small, thus easily maintained. Creating an alternative implementation of JIR should take little to modest effort, hence ensuring JIR's implementation-independence.

D6 Can be tightly integrated to various IDEs: IDEs play an important role in the development of modern software. To help ensure adoption, it should be possible to tightly integrate JIR in popular Java IDEs.

D7 Can easily be constructed by hand: While JIR is targeted for automatic processing, it should also be relatively easy to construct JIR specifications manually. This makes writing test cases easier and allows tool developers working on JML extensions to prototype and experiment with their extension even without a supporting JML front-end.

5.2 JIR definition

While developing JIR, we used the previously stated design goals as a guide; for conflicting goals, we opted for function and ease in engineering/maintenance over form since JIR is not intended as JML input syntax for end-users.

There are three main perspectives from which JIR can be described, namely from the point of view of a:

- JML tool back-end wanting to process JML (represented as JIR);
- JML tool front-end wishing to use the JIR infrastructure to create JIR for back-ends;
- JIR implementations wanting to process JIR class file encodings.

Each of these is covered in the subsections given next. We begin by presenting, at a high level, the JIR APIs offered to JML tooling back- and front-end components. These APIs allow components to be independent of the class file encoding of JIR.

5.3 JIR API for JML tool back-ends

JML tooling back-ends typically expect to receive from a front-end a fully resolved AST with access to AST bindings containing, e.g., type information. In designing the JIR, we faced the conflicting goals of wanting JIR to be front-end independent (D4) and yet be easy to learn and use (D1). From the point of view of a back-end component designer, such as a JIR client, (D4) implies that JML nodes should be presented via an AST hierarchy specific to JIR whereas (D1) implies that the AST should make use of the back-end's native AST classes.

Thankfully, an acceptable compromise is possible; the JIR AST hierarchy offered to back-end components is shown in Fig. 7. The essence of the compromise is to offer JIR specific AST nodes *only* for JML specific nodes, and to rely on the back-end's native AST node types for the rest, essentially grafting native AST node types into JIR ASTs. Hence, in particular, notice that several classes like `JirAssignableClause` have `Expression` as a generic parameter.

A back-end can leverage an existing Java compiler's expression AST type to instantiate `Expression`. To fit a compiler framework's API in JIR, JIR requires an expression parser and an expression pretty printer; these two modules are usually part of a standard compiler framework API. Thus, existing Java compiler frameworks such as JDT and OpenJDK [22] can be used with JIR.

A consequence of the decision to use the back-end's native `Expression` type is that JML-specific expressions must be encoded as pure Java `Expressions`. While the details of this embedding of JML expressions as pure Java expressions is given in Sect. 5.6, suffice it to say here that all JML expression constructs are represented using Java method invocations. Hence, for example, `\result` and `\old()` would be represented by `JIR.result()` and `JIR.old()`, respectively.

The JIR API allows back-ends to load what is called a *JIR Info Map* from any given class file. The JIR Info Map contains all JML specification information for one or more given types. In particular, given a class name, method name, and method signature, JIR API calls can return, from a given JIR Info Map, the corresponding method contract as a JIR AST node build from the types given in Fig. 7. A component known as the *Binding Manager* can be used to access binding (e.g., type) information. The (partial) binding class hierarchy given in Fig. 8 illustrates the kinds of binding information supported in JIR. Similar to the JIR AST hierarchy, some of the binding types are generic over a back-end native binding type `Type`, in this case, representing Java types.

Most type names in the hierarchy are self descriptive; `JirOpBinding` is used to represent all JML constructs that can be used in the context of an expression including operators, keywords and set comprehension expressions.

5.4 JIR API for JML tool front-ends

The primary responsibility of a JIR-enabled front-end is to map JML constructs represented in its native AST node types into JIR. This is typically achieved using visitors that traverse the front-end's native AST creating:

- JIR AST nodes (of Fig. 7) along the way for top-level JML declarations and clauses,
- converting JML expression constructs into their JIR equivalents as (pure) Java expressions, and
- finally accumulating all top-level JML specification elements into a JIR Info Map.

Once this is done, a call to the JIR embedder can be used to embed the JIR Info Map into class files. Details of the encoding are given next.

5.5 JIR class file encoding

The standard Java 5 metadata facility, more commonly referred to as Java 5 annotations, is what we use to embed JIR, and hence JML, into class files. The embedding is achieved as follows:

- JML modifiers like `pure` and `non_null` are embedded as appropriately named Java 5 annotations like `@Pure` and `@NonNull`.
- All other JML constructs are represented via JIR AST nodes. As was mentioned before, these nodes are collected into a data structure called a JIR Info Map. On a per type basis, the relevant part of the map is serialized and attached to the type `.class` file as `@JIR` annotations. The JIR extractor and embedder read/write this annotation.

Tooling front- and back-ends need not be concerned with the details of the JIR encoding since access to JIR is achieved via appropriate APIs as was previously described. This decoupling will allow us to continue experimenting with alternative formulations of the encoding without impacting front- and back-end code.

For sake of completeness, we briefly describe here our current encoding scheme. The JIR Info Map serialization is done using the *XStream*[4] package that can serialize and deserialize Java objects to/from XML. For example, an instance of class C with class members `int i = 3` and

[4] http://xstream.codehaus.org.

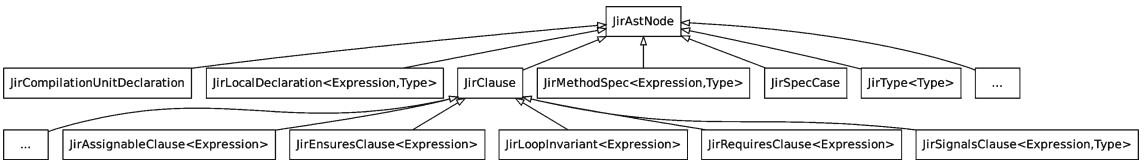

Fig. 7 JIR AST Hierarchy (partial)

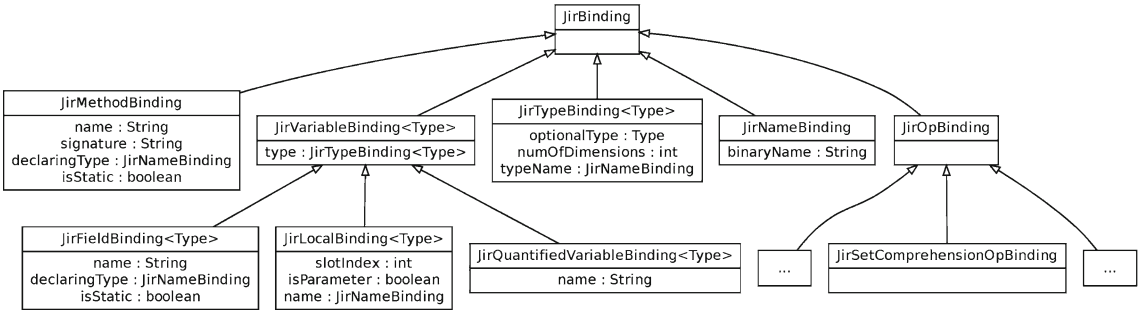

Fig. 8 JIR type bindings hierarchy (partial)

`A a = null` would be represented as

```
<C>
  <i>3</i>
  <A/>
</C>
```

In XStream, it is also easy to provide custom serialization/deserialization mappings. We have made use of this feature to encode non-JIR specification AST nodes, such as Java expressions. In fact, expressions are simply pretty-printed into their Java source syntax; thus, the native AST expression type is not exposed by the encoding. As was mentioned earlier, all JML expressions constructs are represented using pure Java. Details are given next.

5.6 JML expression encoding in JIR

In JIR, JML expressions are represented as fully resolved pure Java expressions that include, for example, typing and "symbol table" information as will be explained shortly. For instance, the ensures clause expression

```
count==\old(count) + 1;
```

of the `inc()` method of Fig. 1 would be represented in JIR as

```
JIR.field(Counter.class, "count", int.class)
==
JIR.old(JIR.field(Counter.class, "count",
                  int.class))+ 1;
```

(position information is omitted in this first example). The `JIR` class contains helper methods such as `old()` and `result()` used to represent JML constructs as method invocations declared as, e.g.:

```
public static native <T> old(T o);
public static native <T> result();
```

In a sense, the JIR encoding of JML expressions has a mixture of source and bytecode level information. More specifically:

- JML operators and keywords such as `\result` and `\old()` are encoded as *method invocations*; and
- regular Java expressions in JML are represented as Java expressions in JIR.
- all JIR expressions are well-formed (hence also well-typed) Java expressions;
- most "symbol table" information is explicitly represented;
- source-level traceability information for symbols is encoded;

Thus, an occurrence of an `int` method parameter `x` on line 10 at column 4, might be represented as[5]

```
annPos(JIR.local("x",int.class,1),10,4))
```

where 1 is the local slot of `x` at the bytecode level. Such bytecode information is needed by back-ends that process class files without inspecting the corresponding Java source. The `annPos()` method call is used to encode, when needed,

[5] Generic types can be preserved using the Java type casting syntax; auto-boxing/unboxing are used to resolve issues with primitive/non-primitive scalars.

the position information for any expression node. As another example, `this` is represented as

```
JIR.thiz(Counter.class)
```

One could also use

```
JIR.local("this", Counter.class, 0)
```

however, the former is a more compact representation without loss of information.

The Java method invocation syntax is used as a *representation* for all JML expression constructs. The Java encoding is not designed to be executable as is; what matters is its *interpretation* by JML tools back-ends that consume JIR expressions.

When a JIR expression is retrieved by a client (e.g., while processing a `requires` clause), the Binding Manager resolves and transforms the expression first. The transformation removes all JIR encodings representing non-JML specific constructs such as source-level traceability information; the returned expression is similar to the Java source-level form (plus JML constructs). Information in JIR encodings such as type and position information are stored as bindings in the Binding Manager accessible through its query API. Moreover, a client can query the Binding Manager to distinguish a JML keyword that is encoded as a method invocation from a real (pure) method invocation in the specification. Thus, when a client traverses the expression AST, it is as if the client were traversing the source JML expressions with all symbol, type, and position information similar to what it would get from a JML front-end. That is, JIR is designed to preserve as much information as needed for JML back-ends.

5.6.1 Quantified expressions

JML quantified expressions are also encoded as method invocations. One key difference as compared to other JML constructs is that quantified expressions declare new variables that are later used in their body. Quantified variables are encoded like local variable references. For example,

```
(\forall int i;... i...)
```

is encoded as

```
JIR.forall(int.class, "i",
   ...JIR.qvar("i",int.class)...)
```

5.6.2 Model specifications

There are three categories of JML model specifications: (1) model types, (2) model fields, and (3) model methods.[6] Each of these is represented in JIR by real Java classes, fields, and methods (marked with `@Model`), respectively. By making these specification elements explicit in JIR, support for model specifications become simpler because symbols associated with model attributes can be resolved as for regular attributes. (A simple checker can be developed to notify users when regular code refers to JIR model elements.)

Also, at first thought, one might believe that this approach to representing model entities by Java entities could affect analysis back-end tools. For example, a program that uses Java reflection to iterate over the methods declared in a class now has more methods to iterate over. Actually, independent of JIR, this is already the case. That is, Java 5 compilers sometimes add extra methods, called *bridge* methods, to help deal with issues that arise due to type erasure in bytecode. Since analysis tools already have to take into account bridge methods, it will be little extra work to filter out `@Model` methods.

5.6.3 Inline specifications

Java 5 annotations are currently limited when it comes to their use inside static blocks or constructor and method bodies; in such cases, annotations can only be applied to (local or catch) variable declarations. This makes it difficult to represent JML inline specifications such as **assert**. To address this, we adopt the approach taken in the Bandera Specification Language (BSL) [25] which leverages Java labels to indicate program points inside methods. For example, we use the method annotation:

```
@Maintaining("L", 14, 20,...)
```

to represent the following loop invariant at label *L* (assuming *L* is at source line 14 and bytecode offset 20):

```
//@ maintaining...;
L: for (int i = 0;...){...}
```

5.7 Alternatives to JIR

The Bytecode Modeling Language (BML), offers a bytecode representation of JML [20]. Similar to BML, JIR uses a mixed source/bytecode level encoding for JML expressions; however, JIR uses standard Java annotations and pure Java expressions that do not require a dedicated processor. The two are complementary, and JIR can easily be translated to BML for tools using BML.

The Microsoft Code Contract project goes beyond our modest goal of providing a means of representing contracts for Java in a tool-independent way. At the heart of the Code Contract approach is a library of static methods and conventions for their use. For example, the postcondition of a user method increasing the value of the field `x` could be

[6] Ghost fields are essentially treated like model fields.

encoded by placing, at the start of the method body, the call to `Contract.Ensures(x>Contract.Old(x))`.

JIR and Code Contract are similar in that they both use method invocations to represent special contract expressions (e.g., `JIR.old()` and `Contract.Old()`). In contrast to JIR, Code Contract specification elements, such as method pre/post-conditions, are written as actual code within the method body. Hence, no special front-end is needed to process them. Since they are written as (plain) code, IDE features are available for use over the contract elements. This can be done because .NET compilers feature generic conditional compilation that can be used to strip out contract code, which alleviates some developers' concern of polluting deployed code with contracts.

Finally, we note that Code Contract tools (a static analyzer and a runtime assertion checker) operate on the bytecode level, while JIR is designed to be used both at the source level and at the bytecode level. We are able to leverage mature Java frameworks such as Eclipse JDT for specification processing for JIR while corresponding frameworks for .NET are still in active development.[7]

Use of method invocations to encode special contract expressions has also been used in other approaches such as in the Bogor software model checking framework [49] for providing language extension mechanisms to model domain-specific abstractions and optimizations (e.g., [4,8, 28]).
Kiasan [26] adopts the same approach to implement custom extensions for interpreting JML-specific expressions [47]. Another example is the Java PathFinder (JPF) [9] that uses native methods as place holders to facilitate extensions.

5.8 Assessment

To a large extent, JIR and its supporting infrastructure satisfy the design goals established in Sect. 5.1. The fact that JML expressions in JIR are encoded as pure Java presents a low barrier of entry (D1), because JML tool developers do not need to learn a new format.[8] Moreover, the representation is front-end/back-end independent (D4). In addition, existing and robust Java compiler (e.g., OpenJDK, Eclipse JDT) and bytecode engineering frameworks (e.g., ASM,[9] BCEL[10]) can be used as is without modification; hence, JIR supporting infrastructure can be implemented within a small core code base (D5). This also means that the JIR infrastructure can be tightly-integrated in popular Java IDEs such as Eclipse or NetBeans (D6). Since JIR uses Java annotations and the method invocation syntax whose schema can be easily modified/enhanced by tool developers, JIR can represent any JML construct and can be easily extended (D2 and D3). In addition, JIR is Java-based at a mixed source/bytecode level that is more amenable for manual construction/modification unlike pure S-expressions (D7).

Perhaps one of the main weaknesses of JIR is that its encoding of JML expressions is a verbose (e.g., symbol table embedding) and non-traditional use of Java syntax. It is non-traditional in that the resulting encoding essentially represents a fully resolved AST, which, if it contains encoding of JML constructs, has no (precise) meaning until it is interpreted by a back-end tool. Our choice of encoding was a difficult choice that we made in order to liberate ourselves from the engineering and maintenance burden of having a dedicated JML processor or being tightly coupled with an unstable (internal) compiler API. However, since JIR is designed mainly for automatic processing, a representation is just that, a representation. Considering JML's tool history, its current state, the scale of the problem with the ever expanding Java language features, and the number of active contributors/benefactors, we believe this is a worthwhile compromise. In the end, what really matters is not the encoding of the JML intermediate representation, but the impact that JML and its tools make on formal method research and software reliability in general.

Regardless, the current verbosity of the JIR expression encoding may be alleviated by string compression techniques that we are currently experimenting with. Moreover, the JIR software infrastructure decouples JIR back-ends and the actual JIR encodings used. That is, the JIR API that back-ends use is not affected by encoding changes.

6 Revised JMLEclipse architecture, current and future plans

6.1 Redesign reduces coupling to JDT

The main effort in the redesign of JMLEclipse has been in the creation of JML Intermediate Representation (JIR) and its supporting infrastructure that has been previously discussed. A secondary effort was the refactoring of JMLEclipse itself to make use of JIR. As a result, both the coupling to the JDT and the invasive code changes to it have been reduced as illustrated in Fig. 9.

While in the original design of JMLEclipse, invasive changes were made to the scanner, parser and the JDT internal AST classes, the only invasive changes that remain in the JDT are to the scanner and parser (and even this can be reduced or completely eliminated as is explained next). Instead of the internal ASTs, JMLEclipse now makes use of the JDT's public DOM types and provides JML specific AST nodes via the JIR AST package.

[7] M. Barnett, personal communication, 2009.

[8] Technically, there is the encoding itself, but it is rather straightforward, e.g., \old() maps to JIR.old().

[9] http://asm.objectweb.org.

[10] http://jakarta.apache.org/bcel.

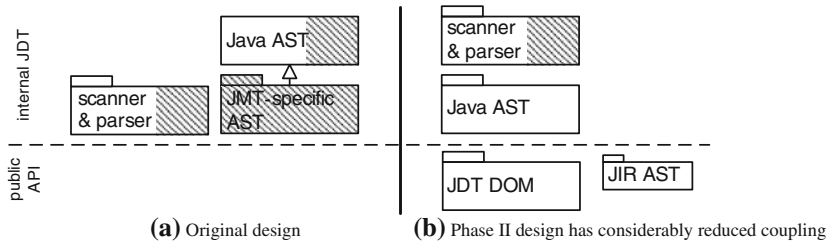

Fig. 9 JMLEclipse design and "foot-print" (amount of invasive code changes) on the JDT. *Grayed* areas show JMLEclipse specific code. **a** Original design. **b** Phase II design has considerably reduced coupling

In the subsection that follows, we present two alternate input syntaxes for JML, which, if adopted, would enable JML to be supported by means of standard Java language features. Some experimental support for these languages has been integrated into JMLEclipse.

6.2 Towards supporting JML as a standard extension to Java

Earlier, the DSRG conducted an empirical study whose goal was to determine if developers, having native programming language support for Design by Contract (DBC), would make use of program assertions (the basic ingredient of contracts) in a proportion higher than in other languages. The study results indicated that programmers using Eiffel (the only active language with built-in support for DBC) tend to write assertions at a rate that is 1.6 to 5 times higher than in languages having basic support for assertions but lacking support for DBC [14].

This study gives weight to one of our secondary JML Eclipse goals: to explore new Java language features to help bring JML closer to being a standard extension to Java. In doing so, our hope is that eventually, Java will be enhanced with native support for DBC: as of the fall of 2009, the top Request For Enhancement (RFE) listed at `bugs.sun.com` is added support for DBC.

6.2.1 A Java 5 annotation-based input syntax

As a first step in this direction, JMLEclipse now also recognizes JML constructs expressed in the form of Java 5 annotations. Figure 10 shows part of the `Counter` class originally shown in Fig. 1 rewritten to make use of the Java 5 annotations designed by Kristina Boysen Taylor in the context of her work on JML5 [53]—JML5 will be discussed in more detail in the related work section. Use of Java 5 annotations enables JML tool features (e.g. ESC) to be written as Java annotation processors, and thus be usable with any Java compiler [52] as opposed to only JMLEclipse. Furthermore, as of Java 6, annotations offer a standard means of embedding JML specifications within class files, something which up until now required a special encoding of a variant of JML called the Bytecode Modeling Language (BML) [20]. Being a standard Java feature, use of annotations also opens the door to the processing of JML by non-JML tools.

```
public class Counter {

    public final static int MAX = 100;

    @SpecPublic private int count;
    //...

    @Ensures("\result == count")
    @Pure public int getCount() {
      return count;
    }

    @Also({
      @SpecCase(
       requires = "count < MAX",
       ensures = "count == \old(count) + 1"),
      @SpecCase(
       requires = "count == MAX",
       ensures = "count == 0")
    })
    public void inc() {
      count = count < MAX ? count + 1 : 0;
    }
}
```

Fig. 10 Example of JML written using Java 5 annotations

This is already being experimented with: Modern Jass [46] offers RAC support for a subset of JML [54] through the exclusive use of "the Pluggable Annotation Processing API to validate contracts, and the Bytecode Instrumentation API to enforce contracts" [45]. Of course, in having different tools processing the same JML annotations, there will be a need to establish a clear semantics for each annotation type [21].

Not all JML constructs can currently be encoded as Java 5 annotations; this will still be the case after the added support of JSR-308 [30] to Java 7 which will allow annotations anywhere types are allowed—including local variable declarations. Hence, JMLEclipse will continue to support the current JML comment-based annotation syntax for a while. In any case, the use of Java 5 annotations to encode JML specifications is still experimental. More extensive use of this feature is needed in order to help confirm the most suitable way to encode JML specifications.

6.2.2 Java contract notation

Inspired by Microsoft's Code Contracts, we have defined a notation that we call Java Contract, or JC for short. A sample JC for our running counter example is given in Fig. 11. One can notice that the notation is similar in overall appearance to original JML2 syntax. The advantages previously mentioned for Code Contracts also apply to JC. Thus, in

```
public class Counter {

    public final static int MAX = 100;

    @CodePrivate public int count;
    //...
    @Pure public int getCount() {
      JC.spec(JC.ensures(
         JC.<Integer>result() == count)
      );
      return count;
    }

    public void inc() {
      JC.spec(
        JC.requires(count < MAX),
        JC.ensures(count == JC.old(count) + 1)
      );
      JC.spec(
        JC.requires(count == MAX),
        JC.ensures(count == 0)
      );

      count = count < MAX ? count + 1 : 0;
    }
}
```

Fig. 11 Example of JML using our Java Contract notation

particular, all Java IDE features are applicable to JC encoded JML such as syntax highlighting. More significantly, e.g., the frequently used Eclipse operations of class, field or method renaming will also apply the renaming to JC. Our redesigned JMLEclipse supports JC to JIR transformations. Since Java does not support conditional compilation, like the .NET compilers we have a separate bytecode "stripper" that removes, e.g., JC method contracts from the start of method bodies since these contracts are non-executable. We are currently exploring an alternative scheme such as providing a simple implementation of conditional (method) compilation in the JMLEclipse core.

We have yet to make extended use of these new alternate input syntaxes to JML, though we feel they hold promise. Extended use will be necessary before conclusions can be drawn on which notation should be proposed as *the* official JML input syntax in the near future.

7 Assessment

Goals In Sect. 3, we listed three goals to be satisfied by any next generation tooling infrastructure for JML. In summary, (1) the infrastructure should be built as an extension to a compiler + IDE whose maintenance is guaranteed by others, (2) minimize integration efforts as the IDE code base evolves, and (3) demonstrate the feasibility of supporting the full range of current verification technologies.

The first goal has been achieved simply by our choice of the Eclipse JDT as a tool base—though doing so introduced some risks that we discuss in the next section. Since the second goal (ease of maintenance) is related to the risk items, we defer assessment of this goal to the next section as well. Achievement of the third goal is demonstrated by the implementation of the most recent JMLEclipse feature set we have achieved, for a non-trivial subset of JML, support for RAC, ESC, FSPV and symbolic execution.

As evidence that JMLEclipse is actually usable in practice, we point out that we have successfully applied it to a case study (totaling over 470K SLOC) in which we made use of the enhanced non-null-type static analysis and RAC capabilities of JMLEclipse [17].

Risk items The Eclipse JDT is a very dynamic code base, with code changes introduced every few weeks. This was perceived as an important risk item for JMLEclipse. As we explain next, early on we found a suitable means of allowing JMLEclipse to extend the JDT so as to keep JMLEclipse code rework to a minimum when JDT changes are released.

JMLEclipse, like JML2, is built as a closely integrated and yet (relatively) loosely coupled extension to an existing compiler. An additional benefit for JMLEclipse is that the timely compiler-base maintenance is assured by the Eclipse Foundation developers. Hence, as compared to JML2, we have traded in committer rights for free maintenance; a choice which we believe will be more advantageous in the long run—in particular due to the rapid pace of the evolution of Java. Unfortunately, loosing committer rights means that we must maintain our own mirror of the JDT code.

A careful adoption of certain coding conventions has allowed us to merge regular JDT updates (since 2006) in less than 15 min on average; this is especially true since our phase II decoupling of JMLEclipse. One of our objectives is also to make it easy for all members of the JML research community to extend JMLEclipse, such as integrating their own tools. While concerns were expressed about the steep learning curve involved with the original JMLEclipse design [48], our initial experience with the JMLEclipse design based on the public DOM API is very encouraging. The public JDT DOM API is well documented and versatile.

JIR and JMLEclipse Phase II Redesign The creation of the JML Intermediate Representation (JIR) has helped us further decouple JMLEclipse from the JDT and it offers the promise of being a convenient exchange format between JML tooling front- and back-ends.

To date, font-end support for JIR has been implemented in both JMLEclipse and OpenJML. JIR enabled back-ends include the JMLEclipse Static Verifier components and Kiasan.

8 Related work

8.1 Verification tool support for Java and/or JML

In this section, we briefly compare JMLEclipse to its sibling next generation projects JML3, JML5, JaJML as well as to Jahob, the Java Applet Correctness Kit (JACK) and

OpenJML. Further details, examples and tools are covered in [16].

JML3 The first next-generation Eclipse-based initiative was JML3, created by David Cok. The main objective of the project was to create a proper Eclipse plug-in, independent of the internals of the JDT [23]. Considerable work has been done to develop the necessary infrastructure, but there are growing concerns about the long-term costs of this approach.

Because the JDT's parser is not extensible from public JDT extensions points, a separate parser for the entire Java language and an AST had to be created for JML3; in addition, Cok notes that "JML3 [will need] to have its own name/type/resolver/checker for both JML constructs [and] all of Java" [23]. Since one of the main goals of the next generation tools is to escape from providing support for the Java language, this is a key disadvantage.

Jahob is a language and system used in the verification of Java programs. In marked contrast to JML, the Jahob language is a BISL for Java based on *higher-order* logic—not without some resemblance in syntax to Isabelle [58]. Jahob tools support VC generation and discharging using a variety of first-order and higher-order provers including CVC3, Z3 and Isabelle. Like JMLEclipse, the Jahob system makes use of multiple provers during any given verification run to discharge VCs [59]. On the other hand, Jahob does not appear to support Runtime Assertion Checking (RAC).

JACK The Java Applet Correctness Kit (JACK) is a tool for JML annotated Java Card programs initially developed at Gemplus (2002) and then taken over by INRIA (2003) [7]. It uses a weakest precondition calculus to generate proof obligations that are discharged automatically or interactively using various theorem provers [12]. While JACK is a candidate next generation tool (offering features unique to JML tools such as verification of annotated byte code [11]), being a proper Eclipse plug-in, it suffers from the same drawbacks as JML3 with respect to the need to maintain a separate compiler front-end. Additionally JACK does not provide support for RAC; we believe RAC is an essential component of a mainstream IVE. An advantage that JACK has over JMLEclipse's current capabilities is that it can present VCs to the user in a Java/JML-like notation. The key drawback of JACK is that, since 2007, it is no longer maintained.

JML5 The JML5 project was the subject of Kristina Boysen Taylor's master's thesis work done under the guidance of Gary Leavens [53]. The main project goal was to explore the feasibility of embedding JML specifications in Java 5 annotations rather than Java comments. As was mentioned earlier, such a change would allow JML tools to be written as standard annotation processors to de-use in conjunction with any Java 5 compliant compiler.

Unfortunately, the implementation of JML5 was not fully completed. This may have been due, in part, to the difficulty faced in adapting the MJ compiler to parse subsets of the full JML grammar. For example, an @Requires annotation value is a string representing a JML predicate expression. In contrast, the Eclipse JDT was designed as an incremental compiler with sophisticated parse error recovery schemes. Thus, adapting JMLEclipse to parse JML expressions or any other JML grammar non-terminal can naturally be achieved given the base capabilities of the JDT core.

JaJML is an early research prototype built atop the JastAdd Java compiler [34]. JastAdd is a compiler framework that uses attribute grammars and supports Aspect Oriented Programming (AOP) [32]. JaJML's main advantage over JMLEclipse is the ease with which it can be extended. Indeed, Haddad and Leavens note that adding RAC support to JaJML for while loop variants and invariants was done in a fraction of the number of lines of code that were needed to add them to JMLEclipse. The main disadvantages of JaJML include its lack of integration with an IDE and no guaranteed third-party maintenance for the underlying JastAdd Java compiler. While use of JastAdd offers increased modularity, there is currently no empirical data on its performance impact. The ability of JaJML to provide support for static verification has yet to be de-risked.

OpenJML is currently the most inclusive next generation JML compiler in terms of its base language support for JML. Built by David Cok atop Sun's OpenJDK, it offers a checker, RAC and ESC.

OpenJML has been a successful experiment in creating a JML compiler as a direct extension to its base. Its main drawback, as compared to JMLEclipse, is its lack of integration with an IDE. OpenJML does not, as of yet, have support for FSPV.

Summary Table 1 presents a summary of the comparison of the tools supporting JML. As compared to the approach taken in JMLEclipse, the main drawback of the other tools, with the exception of OpenJML, is that they are likely to require more effort to maintain over the long haul as Java continues to evolve due to the looser coupling with their base.

8.2 Verification tool support for other languages

KeY While the KeY tool was adapted to accept JML, it also supports other languages [1]. KeY is an integrated development environment that targets verification at a slightly higher level than most other tools. This is because KeY supports the annotation of design artifacts such as class diagrams. KeY does not support RAC or ESC though both automated and interactive FSPV are supported. Like JACK, KeY presents VCs in a JML-like notation.

Omnibus is a functional language with syntax similar to Java's that compiles to Java bytecode [56]. The language was designed with reduced capabilities as compared to Java (e.g., it lacks support for exceptions, interface inheritance, and

Table 1 A comparison of possible next generation JML tools

		JML2	JML3	JML Eclipse	JML5	JaJML	ESC/ Java2	JACK	OpenJML
Base Compiler/IDE	Name	MJ	JDT	JDT	any Java 7+	JastAdd Java	EJ2	JDT	OpenJDK
	Maintained (supports Java≥ 5)	×	✓	✓	✓	✓	×[a]	×	✓
Reuse/extension of base (e.g. parser, AST) vs. copy-and-change		✓	×	✓	×	✓	×	×	✓
Tool Support	RAC	✓	✓	✓	(✓)	✓	N/A	N/A	✓
	ESC	N/A	(✓)	✓	N/A	×	✓	✓[b]	×
	FSPV	N/A	×	✓	N/A	×	N/A	✓	×

MJ MultiJava, *JDT* Eclipse Java Development Toolkit, *N/A* not possible, practical or not a goal, (✓) = planned
[a] ESC/Java2 is being maintained, but its compiler front end has yet to reach Java 5
[b] Strictly speaking, JACK supports an automated form of FSPV, not ESC

concurrency) so as to ease verification. Each of the source files in an Omnibus project has an associated verification policy that gives the level of verification required for it, which can be RAC, ESC, or FSPV. A custom IDE was developed that make these and other development activities easier, including tracking the verification status of (and method used for) each file. Simplify and PVS are the theorem provers used for ESC and FPV, respectively. Hence, Omnibus offers verification support comparable to that of JMLEclipse, though not for a mainstream language.

RESOLVE is the name of a framework, approach/methodology and language used to describe component-based systems. The implementation language is an imperative object-like language whose central swap operator (which replaces the traditional assignment operator) is key to its alias avoidance [29]. The Resolve Verifying Compiler appears to support ESC and FSPV, using either an integrated prover or Isabelle [51]. It is unclear whether the Resolve Verifying Compiler, or other resolve tools, support RAC. In recent work, Kulczycki has illustrated how the Resolve approach could be applied directly when writing a disciplined form of Java [39].

SPARK [5] was developed for the implementation of safety-critical control systems. It is a subset of Ada extended with annotations to provide support for DBC enriched with data flow specifications. The subset was chosen to be amenable to ESC and FSPV and yet be useful for writing industrial applications. Unlike the other languages discussed in this section, there is no support for RAC, since static verification is used to show that errors—including contract violations—cannot happen at runtime.

Static analysis is performed in three stages. The first stage is provided by the Examiner and is similar to the JDT's flow analysis. In the second stage, the Simplifier automatically discharges those VCs that it can and leaves the rest for the Proof Checker, an interactive theorem prover. The Proof Obligation Summary (POGS) tool is used to reduce the various outputs of the static analysis and proof tools to a single report that gives the status of the verification process overall including, in particular, a list of any VCs that remain unproved. In a sense, when using Offline User Assisted ESC, JMLEclipse can be seen to behave like the POGS. All of these tools are stand-alone command-line tools.

9 Conclusion and future work

Reengineering a fairly extensive first generation tool base while creating new tools and elaborating the tooling infrastructure is going to take considerable time and effort. In this paper we have presented our contribution to this effort— JMLEclipse.

The idea of providing JML tool support by means of a closely integrated and loosely coupled extension to an existing compiler was successfully realized in JML2. Although this worked well, unfortunately the chosen base Java compiler is no longer officially maintained. Applying the same approach, we have extended the Eclipse JDT to create the base infrastructure of JMLEclipse. During the inception phase of this project, an early JMLEclipse prototype served as a basis for discussion by some members of the JML consortium, and eventually it came to be adopted as the main avenue to pursue in the JML Reloaded effort. A JML Winter School followed in 2008, during which members of the community were given JMLEclipse developer training [40]. Since then, we have enhanced JMLEclipse's feature set, in particular, with support for next generation ESC, FSPV and symbolic execution components.

Early experience in using JMLEclipse as a tooling platform prompted us to re-architect JMLEclipse creating a minimal JDT-dependant core, with all other components working from public JDT APIs. This has been successful to a large extent because of the creation and use of the JML Intermediate Representation (JIR), which, in addition to helping to further decouple JMLEclipse from the JDT, it also holds the promise of being an effective tool exchange format. As a

preliminary demonstration of this, we have JIR-enabled not only JMLEclipse but also OpenJML. We are hopeful, that this re-architected JMLEclipse will be a strong candidate to act as a next generation research platform and industrial grade verification environment for Java and JML.

Acknowledgments This article is an extension of our earlier work presented as the Second IFIP Working Conference on Verified Software: Theories, Tools, and Experiments [16]. This research was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Québec Fonds de Recherche sur la Nature et les Technologies, and by the US National Science Foundation (NSF) award CNS-0709169 and CAREER award CCF-0644288.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Softw. Syst. Model. **4**, 32–54 (2005)
2. Aspinall, D.: Proof General. http://proofgeneral.inf.ed.ac.uk (2008)
3. Aspinall, D., Winterstein, D., Luth, C., Fayyaz, A.: Proof general in Eclipse: system and architecture overview. In: Proceedings of the OOPSLA Workshop on Eclipse Technology eXchange (ETX) (2006)
4. Baresi, L., Ghezzi, C., Mottola, L.: On accurate automatic verification of publish-subscribe architectures. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 199–208 (2007)
5. Barnes, J.: High integrity software: the Spark approach to safety and security. Addison-Wesley, Reading (2003)
6. Barnett, M., Leino, K.R.M.: Weakest-precondition of unstructured programs. In: Workshop on Program Analysis for Software Tools and Engineering (PASTE), Lisbon, Portugal. ACM Press (2005)
7. Barthe, G., Burdy, L., Charles, J., Grégoire, B., Huisman, M., Lanet, J.-L., Pavlova, M., Requet, A.: JACK: a tool for validation of security and behaviour of Java applications. In: Proceedings of the 5th International Symposium on Formal Methods for Components and Objects (FMCO), (2007)
8. Bianculli, D., Ghezzi, C., Spoletini, P.: A model checking approach to verity BPEL4WS Workflows. In: Proceedings of the IEEE Conference on Serice-Oriented Computing and Applications (SOCA), pp. 13–20 (2007)
9. Brat, G., Havelund, K., Park, S., Visser, W.: Java PathFinder—a second generation of a Java model-checker. In: Proceedings of the Workshop on Advances in Verification (2000)
10. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. Int. J. Softw. Tools Technol. Transfer **7**(3), 212–232 (2005)
11. Burdy, L., Huisman, M., Pavlova, M.: Preliminary design of BML: a behavioral interface specification language for Java bytecode. In: Proceedings of the Fundamental Approaches to Software Engineering (FASE). LNCS, vol. 4422, pp. 215–229 (2007)
12. Burdy, L., Requet, A., Lanet, J.-L.: Java Applet correctness: a developer-oriented approach. In: Proceedings of the International Symposium of Formal Methods Europe. LNCS, vol. 2805, pp. 422–439. Springer, Berlin (2003)
13. Catano, N., Wahls, T.: Executing JML specifications of Java card applications: a case study. In: Proceedings of the ACM Symposium on Applied Computing, Software Engineering Track (SAC-SE), Hawaii, March 2009
14. Chalin, P.: Are practitioners writing contracts? In: Butler, M., Jones, C.B., Romanovsky, A., Troubitsyna, E. (eds.) Rigorous Development of Complex Fault-Tolerant Systems. LNCS, vol. 4157, pp. 100–113. Springer (2006)
15. Chalin, P., James, P.R.: Non-null references by default in Java: alleviating the nullity annotation burden. In: Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP), Berlin, Germany, July–August. LNCS, vol. 4609, pp. 227–247. Springer, New York (2007)
16. Chalin, P., James, P.R., Karabotsos, G.: JML4: towards an industrial grade IVE for Java and next generation research platform for JML. In: Proceedings of the International Conference on Verified Software: Theories, Tools, Experiments (VSTTE), Toronto, Canada. October 6–9, 2008
17. Chalin, P., James, P.R., Rioux, F.: Reducing the use of nullable types through non-null by default and monotonic non-null. IET Softw. J. **2**(6), 515–531 (2008)
18. Chalin, P., Kiniry, J., Leavens, G.T., Poll, E.: Beyond assertions: advanced specification and verification with JML and ESC/Java2. In: Fourth International Symposium on Formal Methods for Components and Objects (FMCO'05). LNCS, vol. 4111, pp. 342–363 (2006)
19. Cheon, Y.: A runtime assertion checker for the Java Modeling Language. Iowa State University, Ph.D. Thesis. TR #03-09, April 2003
20. Chrzaszcz, J., Huisman, M., Schubert, A., Kiniry, J., Pavlova, M., Poll, E.: BML Reference Manual. http://bml.mimuw.edu.pl/ (2008)
21. Cok, D.: Adapting JML to generic types and Java 1.6. In: Proceedings of the International Workshop on Specification and Verification of Component-Based Systems (SAVCBS), Atlanta, Georgia (USA). November 2008
22. Cok, D.: OpenJML. http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJML (2009)
23. Cok, D.R.: Design Notes (Eclipse.txt). http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/trunk/docs/eclipse.txt (2007)
24. Cok, D.R., Kiniry, J.R.: ESC/Java2: uniting ESC/Java and JML. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) Proceedings of the International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04), Marseille, France, March 10–14, 2004. LNCS, vol. 3362, pp. 108–128. Springer (2004)
25. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Robby: Expressing checkable properties of dynamic systems: the Bandera specification language. Int. J. Softw. Tools Technol. Transfer **4**(1), 34–56 (2002)
26. Deng, X., Lee, J., Robby: Bogor/Kiasan: a k-bounded symbolic execution for checking strong heap properties of open systems. In: Proceedings of the IEEE/ACM Conference on Automated Software Engineering (ASE), pp. 157–166 (2006)
27. Deng, X., Robby, , Hatcliff, J.: Kiasan/KUnit: Automatic Test Case Generation and Analysis Feedback for Open Object-Oriented Systems. Kansas State University, Kansas (2007)
28. Dwyer, M.B., Robby, Tkachuk, O., Visser, W.: Analyzing interaction orderings with model checking. In: Proceedings of the IEEE Conference on Automated Software Engineering, Linz, Austria, pp. 154–163 (2004)
29. Stephen, H.E., Wayne, D.H., Timothy, J.L., Murali, S., Bruce, W.W.: Part II: specifying components in RESOLVE. SIGSOFT Softw. Eng. Notes **19**(4), 29–39 (1994)
30. Ernst, M., Coward, D.: Annotations on Java Types. JCP.org, JSR 308. October 17, 2006
31. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Proceedings of the

Conference on Computer Aided Verification (CAV). LNCS, vol. 4590 (2007)
32. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley, Reading (2005)
33. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement. **37**(5), 234–245 (2002)
34. Haddad, G., Leavens, G.T.: Extensible Dynamic Analysis for JML: A Case Study with Loop Annotations. University of Central Florida CS-TR-08-05. April, 2008
35. James, P.R., Chalin, P.: Extended static checking in JML4: benefits of multiple-prover support. In: Proceedings of the ACM Symposium on Applied Computing, Software Verification and Testing Track (SAC-SVT), Hawaii. March 2009
36. James, P.R., Chalin, P.: Faster and more complete extended static checking for the Java Modeling Language. J. Autom. Reason. **44**, 145–174 (2010)
37. James, P.R., Chalin, P., Giannas, L., Karabotsos, G.: Distributed, multi-threaded verification of Java Programs. In: Proceedings of the International Workshop on Specification and Verification of Component-Based Systems (SAVCBS), Atlanta, Georgia (USA). November 2008
38. Krause, B., Wahls, T.: jmle: a tool for executing JML specifications via constraint programming. In: Proceedings of the Formal Methods for Industrial Critical Systems (FMICS). LNCS, vol. 4346. March, 2009
39. Kulczycki, G.: Resolve-style components in Java. In: Proceedings of the RESOLVE Workshop, Virginia, USA (2009)
40. Leavens, G.T.: The Java Modeling Language (JML). http://www.jmlspecs.org (2007)
41. Leavens, G.T., Cheon, Y.: Design by contract with JML. http://www.jmlspecs.org (2006)
42. Leavens, G.T., Leino, K.R.M., Mueller, P.: Specification and verification challenges for sequential object-oriented programs. Form. Asp. Comput. J. **19**(2), 159–189 (2007)
43. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual. http://www.jmlspecs.org (2008)
44. Meyer, B.: Applying design by contract. Computer **25**(10), 40–51 (1992)
45. Rieken, J.: Design by contract for Java—revised. Master's thesis, Universität Oldenburg (2007)
46. Rieken, J.: Modern Jass. http://modernjass.sourceforge.net/ (2007)
47. Robby: Sireum. http://www.sireum.org (2009)
48. Robby, Chalin, P.: Preliminary Design of a unified JML representation and software infrastructure. In: Proceedings of the 11th Workshop on Formal Techniques for Java-like Programs (FTfJP'09), Genova, Italy. July 2009
49. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: An extensible and highly-modular model checking framework. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 267–276 (2003)
50. Schirmer N.: A sequential imperative programming language syntax, semantics, Hoare logics and verification environment. In: Isabelle Archive of Formal Proofs (2008)
51. Smith, H., Harton, H., Frazier, D., Mohan, R., Sitaraman, M.: Generating verified Java components through RESOLVE. In: Proceedings of the International Conference on Software Reuse (ICSR), Virginia, USA. LNCS, vol. 5791, pp. 11–20 (2009)
52. Sun Developer Network.: Annotation Processing Tool. http://java.sun.com/j2se/1.5.0/docs/guide/apt (2004)
53. Taylor, K.B.: A specification language design for the Java Modeling Language (JML) using Java 5 annotations. Masters thesis, Iowa State University (2008)
54. Taylor, K.B., Rieken, J., Leavens, G.T.: Adapting the Java Modeling Language (JML) for Java 5 Annotations. Department of Computer Science, Iowa State University, TR 08-06 (2008)
55. van den Berg, J., Jacobs, B.: The LOOP compiler for Java and JML. In: Margaria, T., Yi W. (eds.) Proceedings of the Tools and Algorithms for the Construction and Analysis of Software (TACAS). LNCS, vol. 2031, pp. 299–312. Springer, Berlin (2001)
56. Wilson, T., Maharaj, S., Clark, R.G.: Omnibus: A clean language and supporting tool for integrating different assertion-based verification techniques. In: Proceedings of the Proceedings of REFT 2005, Newcastle, UK. July 2005
57. Wing, J.M.: Writing Larch interface language specifications. ACM Trans. Program. Lang. Syst. **9**(1), 1–24 (1987)
58. Zee, K., Kuncak, V., Rinard, M.C.: Full functional verification of linked data structures. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI) (2008)
59. Zee, K., Kuncak, V., Rinard, M.C.: An integrated proof language for imperative programs. In: Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI) (2009)