# OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse

David R. Cok
GrammaTech, Inc.
Ithaca, NY, USA
cok@frontiernet.net

OpenJML is a tool for checking code and specifications of Java programs. We describe our experience building the tool on the foundation of JML, OpenJDK and Eclipse, as well as on many advances in specification-based software verification. The implementation demonstrates the value of integrating specification tools directly in the software development IDE and in automating as many tasks as possible. The tool, though still in progress, has now been used for several college-level courses on software specification and verification and for small-scale studies on existing Java programs.

## 1 Introduction to OpenJML

OpenJML is a tool for verification (checking consistency of code and specifications) of Java programs. It is an implementation of the Java Modeling Language (JML) [29, 32] and is built using the OpenJDK [41] compiler. OpenJML can be used both as a command-line tool and through a GUI built on Eclipse.

The tool has been under development in bursts since 2009. Initially the project was simply an experiment to see whether OpenJDK would make a good replacement for the custom parser that underlies ESC/Java(2) [37, 16] and the MultiJava compiler that underlies the JML2 tools, neither of which implemented Java generics. The development was reinvigorated in 2011 and is now supporting academic and trial users and staying current with developments in Java.

OpenJML's long-term goal is to provide an IDE for managing program specifications that naturally fits into the practice of daily software development and so becomes a part of expected software engineering practice. To that end, the project has several nearer-term objectives, discussed in more detail in the body of this paper:

- To create an implementation of JML, replacing ESC/Java2;
- To provide a tool that can be used in courses and tutorials on software specification and verification;
- To provide an integration of many techniques proposed in the software verification research or implemented as standalone proofs of concept over the past few decades;
- To be a platform for experimentation:
  - with specification language features
  - with specification encodings
  - with SMT solvers on software verification problems
- To enable case studies of verification of industrial software;
- Eventually, to be a tool to be used in daily, industrial software development.

OpenJML includes operations for a number of related tasks:

- parsing and typechecking of JML in conjunction with the corresponding Java code
- static checking of code and specifications. OpenJML translates Java + JML specifications into verification conditions that are then checked by SMT solvers.

To appear in EPTCS.

- runtime assertion checking by compiling specifications as assertions into the usual Java .class files. OpenJML uses the OpenJDK compiler, enhancing the processing of source files to add appropriate checks that assertions and other specifications hold during execution of the program.
- integration with both Eclipse and command-line tools
- programmatic access through an API to the internal ASTs, type information, compilation and checking commands, and the results of verification attempts, including counterexamples
- (planned): javadoc tool that includes JML documentation
- (planned): automatic, high-coverage test generation, integrated with unit testing
- (planned): specification discovery (a.k.a. invariant inference, function summarization) for Java

This is a long-term project, and so intermediate milestones and accomplishments are important. One such accomplishment was becoming current with Java 7. The accomplishment highlighted in this paper is that OpenJML has now been successfully used in several academic courses. The important points that are relevant to other tools are the successful use of OpenJDK as a foundation, the integration with Eclipse, and the importance of appropriate UI features in making interaction with formal methods readily accessible to users, in particular, effective display of counterexample information.

## 2 Design of OpenJML

OpenJML follows a design adapted from ESC/Java2 and commonly used for software verification systems [16, 34, 26, 1]).

- The procedure specifications are translated into assumptions and assertions interleaved with the Java code, based on the semantics of the specification language. For example, preconditions become assumptions at the beginning of the procedure and postconditions are assertions at the end.
- The code, assumptions, and assertions are translated into a basic block form that uses single-assignment labeling of variables.
- The basic blocks are translated into compact verification conditions (VCs).
- The verification conditions are expressed in SMTLIBv2 format.
- An SMT solver of choice (we used primarily CVC4 [17], and demonstrated interoperability with Z3 [19]) is applied to the VC.
- If the VC is invalid, a counterexample is obtained from the SMT tool.
- The logical variables of the counterexample are translated back to source code variables and text locations; logical variable values are expressed in programming language terms (cf. section 4).
- The counterexample values and the static "execution" path are displayed in the source code editor by hover information and highlighting.

OpenJML extends this basic model to also check the generated VC for vacuity or for multiple falsified assertions. OpenJML constructs a single verification condition (VC) for a method. The default behavior is to check the validity of the entire VC. It is also possible, as some tools do, to check the validity of each distinct path, or the set of paths to each distinct assertion, or other sub-expressions of the full VC. If the full VC turns out to be valid, it may be the desired case of having consistent code and specifications. However, it may also be that some combination of user-supplied specifications creates infeasible paths to particular assertions or to the method exit. Thus, after a determination that the VC is valid, OpenJML checks that there are feasible paths to the procedure exit and to each assertion.

If the full VC is invalid then there is some specification assertion that is falsified, but it may not be the only one. The user has the option of requesting further checks. OpenJML computes a path condition leading to the falsified assertion, appends the negation of the path condition to the VC, and rechecks the

| OpenJDK phase | OpenJML replacement |
|---|---|
| parsing | augmented to parse JML specifications as well, either in .java files or auxiliary .jml files |
| entering type symbols into the symbol table | augmented to add model types as well |
| entering class members into the symbol table | augmented to add model fields and methods as well |
| annotation processing | annotation processing is not yet used or modified in OpenJML |
| name/type attribution and type checking | augmented to typecheck JML features |
| flow checks | augmented to add flow checks on JML constructs |
| | OpenJML typecheck tool ends here |
| | OpenJML static checking is performed by converting the type-attributed AST to VCs and checking them; static checking does not continue with later phases |
| | OpenJML runtime checking inserts a phase here to modify the OpenJDK AST, inserting code to perform the runtime checks, and then continues with the compilation phases |
| desugaring | [no change] |
| code generation | [no change] |

Table 1: Compiler phases in OpenJDK and OpenJML

VC. The negation of the path condition prevents the same assertion from being falsified again (through the same path, or alternately at all), but allows finding other assertion violations later in the path, if they are not totally blocked by the first failed assertion. By repeating this procedure until the accumulated VC is found valid, all falsifiable assertions in the method under scrutiny are found.

# 3 Building on OpenJDK

OpenJDK [41] is the result of Sun Microsystem's project to release the Java Development Environment as free, open-source code. The software was first available in 2007 and is now managed by Oracle. OpenJDK is the basis for many projects, tools, and experiments with language features.

The OpenJDK compiler architecture has lent itself well to extension, though at present it is not designed for that purpose. The architecture consists of applying a separate operation, embodied in a Java class, for each of a sequence of compiler phases, each of which is replaceable. OpenJML registers replacement tools for many of the compiler phases; the replacement tools extend (with Java inheritance) the original tool, adding additional functionality. The OpenJDK phases and OpenJML replacements are shown in Table 1.

The extension of OpenJDK to OpenJML is not quite a pure extension; some modifications of OpenJDK are required. It has required using non-public APIs, which may change but have been stable so far. Quite a few changes of visibility (e.g., private visibility to protected visibility, to enable inheritance) were required. Some refactorings were needed to encapsulate a portion of the functionality within a large method, in order to override just that portion. However, out of 942 files in the source code portion of OpenJDK langtools, 6 needed bug fixes, 5 needed only visibility changes, 21 needed minor refactoring

and visibility changes, 6 needed very moderate changes in structure or functionality, and only 1 needed significant refactoring and new functionality.

Keeping OpenJDK releases up to date through vendor branches encountered few problems. Transition to Java 8 will likely be more difficult and has yet to be attempted.

The main disadvantage of using OpenJDK in conjunction with an IDE is that it is designed as an execute-once compiler. This works efficiently for command-line operations, but is less appropriate for interactive operations through an IDE. An IDE for software verification should track dependencies, so that the tool automatically knows which modules should be re-verified when a given change is made to the software. OpenJML currently does not have that ability, though it is on the list for future work.

# 4 Building on Eclipse

Although OpenJML can be used as a command-line tool, much like a compiler, it is also integrated with an Eclipse plug-in that provides a GUI interface to the type checking, static checking, and runtime assertion checking capabilities of OpenJML. The Eclipse framework is well-supported and mature enough that creating new plug-ins is now a fairly straightforward development task. The author has developed a number of such tools, including interfaces for ESC/Java2 [16], for SMTLIB [14], and for C/C++ using Eclipse/CDT [15]. There are a few aspects of the Eclipse GUI for OpenJML that are particularly relevant to the goal of providing an integrated IDE for verification, discussed in the subsections below.

## 4.1 Status of verification attempts

Recall that JML is a modular verification system: for each Java method, the method's implementation is checked against its own specifications and the specifications of referenced types and methods, but without reference to other methods' implementations. It is convenient to launch verification attempts of all the methods in a system, perhaps in parallel, returning later to address individually the methods and classes that show problems. A typical command-line run will show a mix of failed and successful verifications in a long stream of text.

Thus the OpenJML GUI provides an Eclipse View that maintains a summary of the results of the most recent verification attempts on each method. When dependency tracking is implemented, this View will also show when a result is out of date. A given method may have multiple falsified assertions; the OpenJML GUI records each proof attempt for each method. The user can then select among these to see the details (such as the counterexample values) for any particular proof attempt. A screenshot of the proof status View is shown in Fig. 1. Each method shows the prover that was used and the time taken in the proof attempt; the color of the tree entry shows the status (valid or invalid or out of date).

## 4.2 Showing counterexample paths and values

An important innovation in the Eclipse GUI for OpenJML is the ability to explore counterexample information interactively. If a VC is falsified, OpenJML retrieves the counterexample information from the back-end SMT solver. This information is in terms of the logical variables present in the VC as translated into SMTLIB; those variables and the way the SMT solver expresses its internal values are generally inscrutable even to experts in the tools involved – this has been a complaint since ESC/Java [25] used Simplify [20].

OpenJML retains information about the mapping of source code expressions into the SMTLIB logical variables. Then the counterexample information, which maps logical variables to values, can be reinterpreted as a mapping of source code expressions to values. Recall that the single-assignment transformation gives different logical variables to different uses of source code variables; thus the values assigned to source variables are the values appropriate to the program state at that point in the program flow. The Eclipse GUI can display this information by means of menu actions or hover information when the mouse is placed over a relevant source code expression. Fig. 2 shows a screenshot of the counterexample path for a buggy method.

In addition, the counterexample contains information about the program path from the start of the method to the assertion failure, since it contains the boolean values for each branch condition. Thus the path to the failed condition in the source code can be determined. OpenJML's GUI displays this information both as a textual program trace (in another Eclipse View) and as highlighting overlaid on the source code itself in the editor window.

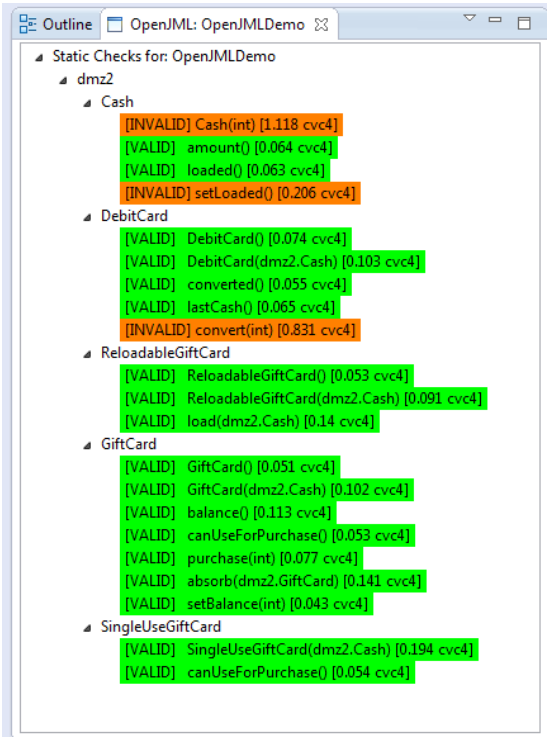


Figure 1: Proof Status View, showing the proof attempt results on a collection of methods

### 4.3 Double compilation

The one disadvantage of this integration with Eclipse is that the Eclipse compiler is executed to show syntax errors and to compile the .class files, and the OpenJML/OpenJDK compiler is executed to perform the OpenJML actions. Thus there is double work - the source code is parsed and typechecked twice. The alternative is to use the Eclipse compiler directly as the front-end for OpenJML. This approach had been tried by both the author and, independently, by Chalin et al.; both efforts concluded that the Eclipse compiler was not (at least at that time) suited for extension in the manner needed by JML. On average the extra work is not noticeable compared to the time to execute the proof attempts using the SMT solvers.

### 4.4 Other Eclipse functionality

The Eclipse GUI contains other features that are adapted to specification and verification tasks:

- Eclipse problem markers to identify syntactic problems or failed specification checks
- integration with the Eclipse build system (to automatically perform type checking, static checking, or runtime assertion compilation)
- menu items to launch various actions, including the ability to map key combinations to actions
- Eclipse preference and help pages
- packaging as a standard Eclipse plugin

Currently JML type checking is performed when a file is saved; background checking during editing is a planned feature. Static checking is performed either when a file is saved or on manual request. Runtime assertions are compiled, when enabled, along with regular compilation.

In the future, OpenJML will acquire some additional GUI features:

- syntax highlighting of keywords in specifications;
- code completion and content assists;
- integrating specifications into Eclipse's Java refactoring and searching tools.

A separate project by the author [15] is implementing invariant inference techniques (for C). OpenJML expects to benefit from this work and to be able to incorporate specification inference as well. That will include features such as these:

- automatically suggested specifications;
- user ability to review, accept, edit, or reject suggested specifications;
- implementing Eclipse Wizards as a way to guide users through the steps of writing specifications.

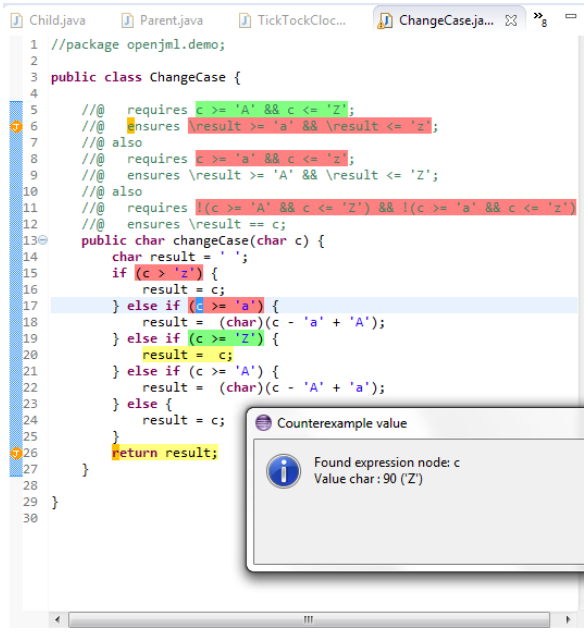


Figure 2: Editor window, with highlighted counterexample path and variable value information

## 5 Building on Software Verification technology

OpenJML intends to be an integration of software verification technology, enabling student learning and further research. These research results come from many different researchers, but are uniquely combined in OpenJML. These various threads include the following:

- the Java Modeling Language (JML community) [46]. Obviously, the Java Modeling Language undergirds OpenJML. Note though that the experience of creating a thorough implementation of JML and assessing its usability has led to many proposals for changes in JML.
- Eclipse plug-in framework (adapted from ESC/Java2). The Eclipse GUI for OpenJML has evolved and improved over time from its origins as the GUI for ESC/Java2.
- basic block/single-assignment translation of programs (Leino and others) and efficient encoding of basic blocks as verification conditions [34, 26, 1] . This basic structure was also used in ESC/Java, though the implementation has been streamlined throughout the OpenJML project. The OpenJML project also expects to integrate with Boogie [28] in the future.
- non-null types (Chalin, Ernst) [9, 8, 42]. JML adopted a default of non-null references.
- handling of undefined expressions (Chalin) [7]. JML originally assigned arbitrary values to otherwise undefined expressions (such as a division by 0). Chalin's survey of users' interpretations of code and specifications led JML to adopt a different interpretation: specifications that cannot be proved to be well-defined are disallowed. So `i/j < 10` (for unconstrained *j*) is undefined, but `j != 0 && i/j < 10` is well-defined.
- arithmetic modes (Chalin) [6]. Chalin's proposal of explicitly allowing different arithmetic modes – pure Java, safe Java (no overflow allowed), and infinite precision arithmetic – was adopted by JML and is being implemented in OpenJML. JML has syntax allowing different modes to be used in different methods and subexpressions (and should allow different scopes).
- source code mechanisms for debugging counterexample information (Cok) [11]. This innovation

is described in section 4.2.
- method calls in specifications (Cok, Müller) [12, 18]. The original ESC/Java inlined called procedures. A scalable system, however, must implement the complex details of modeling called procedures without inlining.
- datagroups and frame conditions (Leino) [33]. JML adopted Leino's work on using datagroups for frame conditions in combination with inheritance and information hiding.
- observational purity (Cok, Leavens) [30]. JML's rules on purity (absence of side-effects) turn out to be somewhat unworkable in practice, particularly combined with inheritance. Cok and Leavens produced initial proposals for additions to JML to accommodate observational purity.
- Java generics (Cok) [13]. Cok's initial design for handling generics in JML with Java 5ff is being extended and adapted as it is implemented in OpenJML.
- vacuity checking (various). A variety of groups used informal techniques to check for specifications causing infeasible paths. OpenJML has implemented the basic technique as a default part of the analysis of a method.
- visibility rules (Leavens, Müller, Leino) [31]. The interaction of information hiding and specifications is intricate. OpenJML has implemented the rules defined in [31]. This has had a distinct effect on how information hiding and specifications are presented to students.
- handling of model fields (Leino, Müller) [36]. Model fields have values defined by *represents* clauses. There are tricky issues regarding when such fields have well-defined values, which can cause unsoundness if implemented poorly.

In addition, there are a number of other technologies planned or under consideration for implementation:
- Universe types [21]
- JSR 308 [22], which allows annotations as part of type identifiers, enabling subtypes to be defined within Java and enabling user-defined type-state-like properties
- integration with annotation processors
- multi-threading
- BML (Bytecode Modeling language) [5], a means to embed specifications in Java byte code
- Use and semantics of model programs [43]
- Java 8
- Integration with unit testing and Daikon [23]

# 6 Status of the implementation of OpenJML

As mentioned already, OpenJML is still a work in progress; nearly all of the development has been an unfunded volunteer effort. Nevertheless, the project has proceeded to the point to be useful in educational settings and for small scale case studies on existing code. Not all features of Java and JML are yet implemented. This section gives a summary of the status as of the submission of this paper. Updates can be found on the OpenJML web site (`www.openjml.org`). In general, all Java and JML language elements are parsed and type-checked, but those not implemented are ignored for static or runtime checking.

## 6.1 Java Language elements

**Java 4:** All language elements are implemented except
- features of multi-threaded Java
- the strictfp modifier (and floating-point semantics in general)

- the volatile, transient, and native modifiers (and details of memory models)

**Java 5:** These are the new features in Java 5:

- Generic types - partially implemented
- enhanced for loop - implemented
- autoboxing and unboxing - implemented
- typesafe enums - simple enum classes are implemented
- static import - implemented (but JML model imports are treated just like Java imports)
- varargs - partially modeled
- Java annotations - No checks on annotations are modeled by ESC or compiled by RAC

**Java 6:** No language changes

**Java 7:** New Java 7 language elements:

- binary literals - implemented
- literals with underscores - implemented
- Strings in switch statements - Compiled in RAC, but not modeled or checked by ESC
- try with resources - Compiled in RAC, but not modeled or checked by ESC
- Catching multiple exception types - Compiled in RAC, but not modeled or checked by ESC
- Type inference - not implemented in JML
- Runtime errors associated with varargs parameter lists - not implemented in ESC

**Java 8:** (OpenJML is only implemented at present for Java 7). These new language features will be required to migrate to Java 8:

- lambda expressions (closures)
- JSR308 and other enhancements to Java annotations

## 6.2 JML language elements

The Java Modeling Language has a large number of features (over 100 groups of features). To list the status of all of them here would be space-consuming, quite quickly out of date, and not particularly interesting in the details. The reader is referred instead to the OpenJML web page (`www.openjml.org`) and the specific page listing implementation status for current status at future times.

The most important missing features are these:

- Basic (level 0) JML is nearly all implemented, with a few restrictions. The main gaps are the implementation of the Universe type system and encoding and SMT solver support for `\sum`, `\product`, and `\num_of` quantifiers. The maps clause (level 1) is needed for more substantial examples. Checks on static and instance initialization are still in progress.
- There is significant desire to include the proposed `\past` operator in standard JML and OpenJML, as an improvement on the traditional `\old`.
- Key advanced features that are waiting for implementation include model programs, some of the less commonly used method specification clause types, and object set predicates such as `\reach`.
- Support for concurrency still requires research, experimentation and trial use before it is ready to be implemented.
- Support for specifications in class files (as in BML [10]) has not yet been investigated.

Despite the missing features, the current status is sufficient to be applied to interesting educational examples and small-scale use cases of industrial code.

## 6.3 Soundness

OpenJML intends to be a sound tool. That is, it intends to correctly encode the semantics of Java and JML. However, the soundness and completeness of all software verification tools is limited in various ways. OpenJML has addressed some of the sources of unsoundness and incompleteness that were present in ESC/Java (cf. Appendix C in [37], from which some of the following discussion is derived). A user of OpenJML experimenting with Java and JML will be concerned about two questions: (a) does every inconsistency between the software and the specifications trigger a warning by OpenJML; (b) does every warning by OpenJML correspond to a fault in the code+specifications.[1]

**Does every inconsistency between the software and the specifications trigger a warning by OpenJML, that is, are there missed errors (false negatives)?**

- Note that one cause of missed errors is incorrect specifications. Wrong specifications can introduce spurious errors, but they can also make program paths infeasible, and thereby hide errors. That is why OpenJML checks for feasibility when the first check of a method indicates no inconsistencies between specifications and code.
- Furthermore, OpenJML verifies each method independently, presuming that all the specifications it uses, including those of other methods, are correct. An error in method A's spec may cause errors to be hidden in method B. All specifications and methods must be consistent simultaneously.
- The semantics of JML state that all invariants of all classes and objects must hold at the beginning of any method. In practice, OpenJML assumes all the invariants of a heuristically-determined relevant set of classes. This has the potential of missing a relevant invariant that might constrain the logical state enough to uncover an error. This source of false negatives is less prevalent if invariants are 'well-behaved', that is, they refer only to fields of their own objects. Implementation of the ownership type system will also mitigate this problem.
- OpenJML relies on SMT solvers to find counterexamples in a collection of logical assertions that represent the program and specifications. The solver may exhaust available time or memory resources; in that case, it can at most state that the solver does not know whether there is a specification inconsistency or not. This is particularly the case when there are quantified assertions, in which case the logical validity problem may be undecidable. Also, of course, the solver may have bugs.
- Finally, of course, OpenJML is not complete, and the portions that are complete may have bugs.

**Does every warning by OpenJML correspond to a fault in the code+specifications, that is, are there spurious warnings (false positives)?**

- There is an intrinsic source of false positives, namely the incompleteness of the underlying logical theories. This allows the underlying solvers to find 'counterexamples' that are not valid in reality. This is particularly the case when quantified expressions are used.
- The above is exacerbated by the fact that some of Java's behaviors have yet to be modeled by JML or the underlying SMT provers (floating point semantics are one example).
- Finally, the modular nature of JML's modeling may preclude some deductions that might be possible with a different style of specification and specification checking.

[1] These two questions correspond to the questions of soundness and of completeness, but which label is applied to which question varies depending on perspective.

# 7 Uses and Experience

To date, there have been these principal uses of OpenJML:

- OpenJML was used as the basis for student assignments in software verification in at least four different courses during the fall of 2013, with others planned in 2014. These courses were undergraduate courses that used formal methods or formal specifications (cf. Section 10).
- OpenJML is used as a basis for research on information flow on the FlowSpecs project (Naumann and Leavens), funded by NSF SaTC grants CNS1228930 and CNS1228695 (`http://www.cs.stevens.edu/~naumann/xxxflowspecs/index.html`).
- OpenJML is integrated into JmlUnitNG [44]
- Use cases of specification and verification of medium-scale Java software are in progress.
- The API has been used (in projects of Kiniry's students) to embed OpenJML in other systems.

Informal experience with OpenJML so far is that, first, the GUI is a major help in editing, checking, and debugging specifications. Second, implementing verification technology for a production language, such as Java, while perhaps more complex for the implementors and users, holds the promise of applying verification technology in actual software development. And third, the integration of the variety of technologies listed in section 5 is a useful (though lengthy) exercise. The exercise of implementing the combination exposes ambiguities in understanding and definition of features. It also exposes complexities in their interactions. One of those is presented in the following subsection.

## 7.1 Frame condition defaults for constructors

Part of the specification of a method is a statement of the frame condition (the *assignable* clause), which states what memory locations (variables, field elements, array elements) the method may modify. For non-constructor methods not marked as pure, the default (when no assignable clause is given) is that the method may modify anything (`assignable \everything;`). For constructors, the default defined by JML is `assignable this.*;`. Originally, `this.*` was defined to mean a list of all the (non-static) fields of the object. This was a useful default, because a typical constructor initializes all the fields of the object. However, this behavior interacts with JML's rules

```
public class A {
  private int _value;

  //@ assignable this.*; // default
  public A(int v) {
    _value = v;
  }
}
```

(a)

```
public class A {
  private int _value; //@ in value;
  //@ model public int value;
  //@ public represents value = _value;

  //@ assignable this.*; // default
  public A(int v) {
    _value = v;
  }
}
```

(b)

```
public class A {
  //@ public model Object \state;//implicit
  private int _value;
  //@ in \state; // implicit

  //@ assignable this.*; //includes \state
  public A(int v) {
    _value = v;
  }
}
```

(c)

Figure 3: (a) Example default assignable clause; (b) with model field; (c) with default model field

about visibility. One visibility rule is that a specification clause may not refer to variables of more restricted visibility than the clause itself; a client that can see the clause must be able to see the variables contained in the clause. Thus, a public assignable clause may refer only to public fields of the object. So, in a public `assignable this.*;` clause, the `this.*` was redefined to expand only to the public fields of the object. As a result, straightforward code such as in Fig. 3(a), will result in a warning: the constructor modifies a field that it is not permitted to. A student might very well write such code as a first attempt to use JML.

There are two remedies for this situation, both of which add complexity for the beginning specifier. The private `_value` field can be declared `spec_public`. Alternately, and equivalently, we can declare a model field serving as a datagroup, and place the private field *in* the datagroup, as in Fig. 3(b). Both of these solutions require some understanding of the interaction of private implementations with public specifications.

A third alternative is to modify JML by adding an implicit model field to every object. The model field, as a datagroup, would contain all the fields of the object (perhaps JML should define `this.*` to be just that model field). This is the equivalent of having the declarations in Fig. 3(c) be present by default in every object's specification. Such a solution would permit the original example to pass without modification; however, it simply hides complexity behind defaults chosen to improve usability.

# 8 Related work

OpenJML builds on ESC/Java [25] and ESC/Java2 [16]. Some static analysis capabilities are now being built into compilers such as Eclipse and Visual Studio. However, the only tools that provide automated checking of functional specifications are Spec# [2] for an extension of C#, Frama-C [27] for C with ACSL [45] specifications, tools for SPARK/Ada and, more recently, Dafny [35, 38]. Eiffel [39] built less expressive specifications directly into the programming language. Leon [4] is a verifier for Scala programs. Key [3] and Why3 [24] are other recent tools that apply to Java programs, but require the user to interact with theorem proving tools separately from the software development IDE. The Mobius project [40] combined a large number of tools (including ESC/Java2) into a suite of verification tools.

# 9 Future work

The work in the immediate future falls into four areas:

- Completing the modeling of the features of basic JML and sequential Java
- Migrating to Java 8 and supporting its significant new features (closures and expanded use of annotations)
- Adding some additional tools, valuable for both education and specification development, such as automatic test generation and documentation
- Enhancing documentation, especially tutorials and instructional material

Eventually, support for concurrency will also be added.

But the more interesting activities are what OpenJML enables. First is work on invariant inference. One goal of an industrial-strength verification system must be to minimize the manual effort and cognitive load on the user by automating as many tasks as possible. Integrating tools to generate candidate specifications from the source code (or, Daikon-like, from test runs), will help to lessen the work currently needed to successfully specify a substantial software application.

A second important area is the thoughtful execution of use cases. The amount of research on verification techniques far exceeds the amount of application of verification. Verification challenges help elucidate the methods needed to handle common programming idioms. In addition, however, tackling real verification problems (in all their messiness) is essential to discovering the techniques, default behaviors, and UI mechanisms that are needed in practice.

A third effort needs to be education in software verification. Adoption of verification techniques as part of daily software development is slow - in part because today's tools are inadequate, but also because of unfamiliarity on the part of software developers. More and better tutorials, education materials, undergraduate and graduate courses, and expectations for actual practice can progress the goal of more application of verification technology and hence better software.

## 10 Acknowledgments

OpenJML is developed by David Cok, with encouragement from the JML community and others concerned with software verification tools.

- Dan Zimmerman (Harvey-Mudd College), Wojciech Mostowski (University of Twente), Marieke Huisman (University of Twente), and others provided valuable usability feedback and bug reports as part of courses they have taught that used OpenJML.
- Student projects by Alysson Milanez and Tiago Massoni contributed test cases.
- An equipment grant from aicas GmbH (`www.aicas.com`) enabled development, particularly on MacOS.
- A small amount of development was performed under NSF SaTC grants, including work by John Singleton on configuration checking: CNS1228930 and CNS1228695, .
- OpenJML is an implementation of JML (`www.jmlspecs.org`), makes use of SMTLIB (`www.smtlib.org`) and is built on both OpenJDK (`openjdk.java.net`) and Eclipse (`www.eclipse.org`); each of those activities have large contributor groups acknowledged on the respective websites. OpenJML also makes use of SMT solvers from various groups, most particularly, CVC4, Z3, and Yices.

## References

[1] Mike Barnett & K. Rustan M. Leino (2005): *Weakest-precondition of unstructured programs*. In Michael D. Ernst & Thomas P. Jensen, editors: *Program Analysis For Software Tools and Engineering (PASTE)*, ACM, doi:10.1145/1108792.1108813.

[2] Mike Barnett, K. Rustan M. Leino & Wolfram Schulte (2005): *The Spec# Programming System: An Overview*. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet & Traian Muntean, editors: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS 2004)*, *Lecture Notes in Computer Science* 3362, Springer-Verlag, doi:10.1007/978-3-540-30569-9_3.

[3] Bernhard Beckert, Reiner Hähnle & Peter H. Schmitt (2007): *Verification of object-oriented software : the KeY approach*. Springer.

[4] Régis Blanc, Viktor Kuncak, Etienne Kneuss & Philippe Suter (2013): *An Overview of the Leon Verification System: Verification by Translation to Recursive Functions*. In: *Proceedings of the 4th Workshop on Scala*, SCALA '13, ACM, New York, NY, USA, doi:10.1145/2489837.2489838.

[5] Lilian Burdy, Marieke Huisman & Mariela Pavlova (2007): *Preliminary Design of BML: A Behavioral Interface Specification Language for Java Bytecode*. In: *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering*, FASE'07, doi:10.1007/978-3-540-71289-3_18.

[6] Patrice Chalin (2004): *JML support for primitive arbitrary precision numeric types: Definition and semantics*. Journal of Object Technology 3(6), pp. 57–79.

[7] Patrice Chalin (2005): *Reassessing JML's logical foundation*. Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP05), Glasgow, Scotland.

[8] Patrice Chalin & Perry R James (2007): *Non-null references by default in Java: Alleviating the nullity annotation burden*. In: ECOOP 2007–Object-Oriented Programming, Springer Berlin Heidelberg.

[9] Patrice Chalin, Perry R. James & Frdric Rioux (2008): *Reducing the use of nullable types through non-null by default and monotonic non-null.* IET Software 2(6), pp. 515–531, doi:10.1049/iet-sen:20080010.

[10] Jacek Chrzaszcz, Marieke Huisman & Aleksy Schubert (2009): *BML and Related Tools*. In FrankS. Boer, MarcelloM. Bonsangue & Eric Madelaine, editors: Formal Methods for Components and Objects, Lecture Notes in Computer Science 5751, Springer, pp. 278–297, doi:10.1007/978-3-642-04167-9_14.

[11] David Cok (2010): *Improved usability and performance of SMT solvers for debugging specifications*. International Journal on Software Tools for Technology Transfer (STTT) 12, doi:10.1007/s10009-010-0138-x.

[12] David R. Cok (2004): *Reasoning with specifications containing method calls in JML and first-order provers*. In Erik Poll, editor: ECOOP Workshop FTfJP'2004 Formal Techniques for Java-like Programs, pp. 41–48.

[13] David R. Cok (2008): *Adapting JML to generic types and Java 1.6*. SAVCBS'08.

[14] David R. Cok (2011): *jSMTLIB: Tutorial, Validation and Adapter Tools for SMT-LIBv2*. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: NASA Formal Methods, Lecture Notes in Computer Science 6617, Springer Berlin Heidelberg, doi:10.1007/978-3-642-20398-5_36.

[15] David R. Cok & Scott C. Johnson (2014): *SPEEDY: An Eclipse-based IDE for invariant inference*. Electronic Proceedings in Theoretical Computer Science (EPTCS).

[16] David R. Cok & Joseph R. Kiniry (2005): *ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system*. LNCS 3362, Springer-Verlag, doi:10.1007/b105030.

[17] CVC4: `http://cvc4.cs.nyu.edu`.

[18] Ádám Darvas & Peter Müller (2006): *Reasoning About Method Calls in Interface Specifications*. Journal of Object Technology 5(5), pp. 59–85. Available at `http://dx.doi.org/10.5381/jot.2006.5.5.a3`.

[19] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: Proceedings of the Theory and Practice of Software, TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg, pp. 337–340. Available at `http://portal.acm.org/citation.cfm?id=1792734.1792766`.

[20] David Detlefs, Greg Nelson & James B. Saxe (2003): *Simplify: A Theorem Prover for Program Checking*. Technical Report HPL-2003-148, HP Labs. Available at `http://www.hpl.hp.com/techreports/2003/HPL-2003-148.pdf`.

[21] Werner Dietl, Sophia Drossopoulou & Peter Müller (2007): *Generic Universe Types*. In Erik Ernst, editor: ECOOP, Lecture Notes in Computer Science 4609, Springer, pp. 28–53, doi:10.1007/978-3-540-73589-2_3.

[22] Michael D. Ernst (2011): *Type Annotations specification (JSR 308)*. `http://types.cs.washington.edu/jsr308/`.

[23] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz & Chen Xiao (2007): *The Daikon system for dynamic detection of likely invariants*. Science of Computer Programming 69(1–3), pp. 35–45, doi:10.1016/j.scico.2007.01.015.

[24] Jean-Christophe Fillitre & Andrei Paskevich (2013): *Why3 — Where Programs Meet Provers*. In Matthias Felleisen & Philippa Gardner, editors: Programming Languages and Systems, Lecture Notes in Computer Science 7792, Springer Berlin Heidelberg, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[25] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe & Raymie Stata (2002): *Extended Static Checking for Java*. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02, ACM, New York, NY, USA, pp. 234–245, doi:10.1145/512529.512558.

[26] Cormac Flanagan & James B. Saxe (2001): *Avoiding exponential explosion: generating compact verification conditions*. *SIGPLAN Not.* 36(3), pp. 193–205, doi:10.1145/373243.360220.

[27] `http://frama-c.com/`.

[28] Claire Le Goues, K. Rustan M. Leino & Michał Moskal (2011): *The Boogie Verification Debugger (Tool Paper)*. In Gilles Barthe, Alberto Pardo & Gerardo Schneider, editors: *Software Engineering and Formal Methods — 9th International Conference, SEFM 2011*, *Lecture Notes in Computer Science* 7041, Springer, pp. 407–414. Available at `http://dl.acm.org/citation.cfm?id=2075679.2075711`.

[29] Gary T. Leavens, Albert L. Baker & Clyde Ruby (1999): *JML: A Notation for Detailed Design*. In Haim Kilov, Bernhard Rumpe & Ian Simmonds, editors: *Behavioral Specifications of Businesses and Systems*, Kluwer Academic Publishers, Boston, pp. 175–188, doi:10.1007/978-1-4615-5229-1_12.

[30] Gary T. Leavens & David R. Cok (2005): *Adapting Observational Purity to JML*. Presented at IFIP WG 2.3.

[31] Gary T. Leavens & Peter Müller (2006): *Information Hiding and Visibility in Interface Specifications*. Technical Report 06-28, Department of Computer Science, Iowa State University, Ames, Iowa. Available at `ftp://ftp.cs.iastate.edu/pub/techreports/TR06-28/TR.pdf`. In ICSE 2007, pages 385-395.

[32] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin & Daniel M. Zimmerman (2008): *JML Reference Manual*. Available from `http://www.jmlspecs.org`.

[33] K. Rustan M. Leino (1998): *Data Groups: Specifying the Modification of Extended State*. In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, ACM, doi:10.1145/286942.286953.

[34] K. Rustan M. Leino (2005): *Efficient weakest preconditions*. *Inf. Process. Lett.* 93(6), pp. 281–288, doi:10.1016/j.ipl.2004.10.015.

[35] K. Rustan M. Leino (2010): *Dafny: An Automatic Program Verifier for Functional Correctness*. In Edmund M. Clarke & Andrei Voronkov, editors: *LPAR-16*, *Lecture Notes in Computer Science* 6355. Available at `http://dl.acm.org/citation.cfm?id=1939141.1939161`.

[36] K. Rustan M. Leino & Peter Müller (2006): *A verification methodology for model fields*. In Peter Sestoft, editor: *European Symposium on Programming (ESOP)*, *Lecture Notes in Computer Science* 3924, Springer-Verlag, pp. 115–130, doi:10.1007/11693024_9.

[37] K. Rustan M. Leino, Greg Nelson & James B. Saxe (2000): *ESC/Java User's Manual*. Technical Note, Compaq Systems Research Center. Available at `http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-TN-2000-002.pdf?jumpid=reg_R1002_USEN`.

[38] K. Rustan M. Leino & Valentin Wüstholz (2014): *The Dafny Integrated Development Environment*. *Electronic Proceedings in Theoretical Computer Science (EPTCS)*.

[39] Bertrand Meyer (1988): *Object-oriented Software Construction*. Prentice Hall, New York, NY.

[40] `http://mobius.ucd.ie`.

[41] `http://www.openjdk.org`.

[42] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins & Michael D. Ernst (2008): *Practical pluggable types for Java*. In: *ISSTA*, pp. 201–212, doi:10.1145/1390630.1390656.

[43] Steve M. Shaner, Gary T. Leavens & David A. Naumann (2007): *Modular verification of higher-order methods with mandatory calls specified by model programs*. In: *OOPSLA*, pp. 351–368, doi:10.1145/1297027.1297053.

[44] Daniel M. Zimmerman & Rinkesh Nagmoti (2011): *JMLUnit: The Next Generation*. In Bernhard Beckert & Claude Marché, editors: *Formal Verification of Object-Oriented Software*, *Lecture Notes in Computer Science* 6528, Springer, pp. 183–197, doi:10.1007/978-3-642-18070-5_13.

[45] ACSL web site. `http://www.frama-c.cea.fr/acsl.html`.

[46] JML web site. `http://www.jmlspecs.org`.