A JML compiler based on AspectJ

Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, Sérgio Soares
Pernambuco State University
Computing Systems Department
Rua Benfica, 455, Madalena, Recife - PE, Brazil
{hemr,ricardo,marcio,sergio}@dsc.upe.br

Abstract

The Java Modeling Language (JML) is a behavioral interface specification language (BISL) designed for Java. It was developed with the aim of improving the functional software correctness of Java applications. The JML compiler (jmlc) generates bytecodes that use the Java reflection mechanism and data structures not supported by Java ME applications. In order to overcome this limitation, we propose the use of AspectJ to implement a new JML compiler, which generates an instrumented bytecode compliant with both Java SE and Java ME applications. The paper also includes a comparative study to demonstrate the quality of the final code generated by our compiler. Results indicate that the overhead in code size produced by our compiler is very small, which is essential for Java ME applications.

1 Introduction

The Java Modeling Language (JML) is a behavioral interface specification language for Java. It combines the design by contract approach of Eiffel and the model-based specification approach of the Larch family [4] of interface specification languages, with some elements of the refinement calculus. The generated bytecodes of the JML compiler (jmlc) [2] uses the Java reflection mechanism and data structures, such as *HashSet* not supported by Java ME applications. Observing such a scenario, the main motivation of this work was to create a new JML compiler compliant with both Java SE and Java ME applications.

Our approach consists in implementing a new JML compiler with support for Java ME applications. The strategy adopted in this work employs AspectJ [5] to implement JML contracts (assertions) that, when compiled, produce instrumented bytecodes to check the correctness of the Java programs during runtime.

To the best of our knowledge, this is the first JML compiler with support for Java ME, which adopts aspect-oriented programming with AspectJ to implement JML contracts.

The main contributions of this paper are: (1) a novel JML compiler compliant with Java ME and Java SE applications; (2) the usage of aspect-oriented programming to implement JML contracts; (3) a comparative study between our and original JML compiler proposed by Cheon [2].

This paper is structured as follows. Section 2 presents the Java Modeling Language (JML). Section 3 presents a novel JML compiler based on AspectJ. A comparative study between our and the original JML compiler that are discussed in Section 4. Finally, Section 5 presents related work, and Section 6 contains the conclusions and future work.

2 Java Modeling Language

The Java Modeling Language (JML) [7] is a behavioral interface specification language designed for Java that follows the Design by Contract (DBC) [8] technique. JML specifications are composed of pre-, postconditions, and invariant assertions annotated in Java code in the form of comments.

In the JML environment, Java comments are interpreted as JML annotations when they begin with an `@` sign, that is, comments of the form: `//@` <JML specification> or `/*@` <JML specification> `@*/`. In a JML specification, we use the keywords `requires`, `ensures` and `signals` that are respectively used to specify preconditions, normal postconditions, and exceptional postconditions of a method. Moreover, the keyword `invariant` denotes a predicate that will always hold after constructor execution, and before and after every method call. JML supports both instance and static invariants.

The JML compiler (*jmlc*) [2] converts JML annotations into automatic runtime checks. It reuses the front-end of existing JML tools [1] to verify the syntax and semantics of the JML annotations and produces a typechecked abstract

syntax tree (AST). The compiler introduces two new compilation passes: the "runtime assertion checker (RAC) code generation"; and the "runtime assertion checker (RAC) code printing". The former modifies the AST to add nodes for the generated checking code. The latter writes the new abstract syntax tree to a Temporary Java Source File (TJSF). For each Java method three *assertion methods* are generated into a TJSF: one for precondition checking, and two for postcondition checking (for normal and exceptional termination). Finally, instrumented bytecode is produced by compiling the TJSF through the Java compiler. The instrumented bytecode produced contains *assertion methods* code to check JML contracts during runtime.

3 A novel JML compiler based on AspectJ

In this section we describe a new approach for the implementation of a JML compiler based on AspectJ. Cheon's technique [2] to implement JML compiler has the following limitations to be employed for small devices: (1) it adopts Java reflection specially to support specification inheritance (not supported by Java ME); (2) it employs data structures, such as *HashSet* and *Map*, both from the java.util package (not supported by Java ME).

In order to overcome these limitations, we reuse the front-end of the JML tools [1] and generate *Aspect Assertion Methods* (AAM) for each Java method from the type-checked AST. Then, the AAM are compiled through the AspectJ compiler (ajc), which weaves the AAM with the Java code. The result is an instrumented bytecode, compliant to both Java SE and Java ME applications. This implementation technique (the reuse of the JML tools) is also provided by Cheon [2].

3.1 Mapping contracts to Aspects

In this section, we present a mapping of JML into *aspects* [5]. We concentrate on precondition, normal postcondition and invariant. For the mapping rules, suppose a method with signature `void m()` in a class `C` that inherits from a class `B` with an overridden method `m` that has the precondition α, postcondition γ, invariant θ and possible inherited specifications (from class `B`).

It is important that readers are familiar with the basic concepts of Aspect-Oriented Programming (AOP) with AspectJ [5] and the following terminology: aspect, join point, named and anonymous pointcut, and advice.

3.1.1 Precondition Mapping

In JML, preconditions must be satisfied (held) before any code of a method is executed, otherwise an exception must be thrown to signal the violation of the precondition. Thus, for the precondition mapping, we have to know the current precondition and take into account all the inherited preconditions (if any). In JML, subclasses inherit specifications as well (*specification inheritance model*). Thus, specifications can be added to an overridden method. According to Leavens [6, Definition 1], the result of precondition inheritance is a disjunction of the form $(\texttt{pre}' \mid\mid \texttt{pre})$, where $\texttt{pre}'$ is the method precondition defined by the subclass and `pre` is the method precondition defined by the superclass.

In order to achieve the effect of precondition inheritance, a *static crosscutting* mechanism of AspectJ also known as *inter type declaration* is used to insert a new method into class C[1] as follows:

```
1: public boolean C.checkPre$m(){
2:  return α || super.checkPre$m();
3: }
```

This code inserts a method into class `C` (the `C.` prefix tells AspectJ where to insert the method definition) that checks its precondition in conjunction with the proper inherited precondition (`super.checkPre$m()`). It is also used by the following AspectJ's advice to give a full-fledged JML preconditions semantics.

```
1: before(C current) :
2:  execution(void m())) && within(C) &&
3:  this(current){
4:    if(!current.checkPre$m()){
5:      throw new JMLPreconditionError();
6:    }
7: }
```

This `before` advice (line `1`) is responsible for inserting an extra behavior (lines `4`, `5` and `6`) before some specified points in the Java program. Moreover, the advice parameter `current` indicates that we want to expose some information about the execution context. The designator `this` (line `3`) exposes the currently executing object, which is the object executing the method `m`. The affected points are defined through the AspectJ designators `execution` and `within` (line `2`). The former specifies which methods' executions will be affected, in this case, executions of a `void` method `m` without parameters in a class `C`. The latter constrains the execution points to methods of a specified class (`C`), which avoids executions in subclasses of it. The behavior added by this advice is to check the precondition (line `4`) and throws `JMLPreconditionError` (line `5`) if it is violated. Assertion violations in the JML semantics are instances of java.lang.Error [2].

3.1.2 Normal Postcondition Mapping

Postconditions are properties in JML that must hold after method calls. There are two kinds of postconditions

[1] If the method has formal parameters, their declaration is useful to distinguish from overloaded methods.

in JML: normal and exceptional postcondition. Here, we cover only normal postcondition. Exceptional postcondition will be treated in a future work. About postcondition inheritance, Leavens [6, Definition 1] states postconditions as a conjunction of implications in the form (\old(pre′)==>post′) && (\old(pre)==>post). Expressing that when one of the preconditions holds, then the corresponding postcondition must hold. The expression (\old(pre′)) corresponds in JML, the evaluation of the precondition before method call. Its implementation will be treated in a future work as well. The method inserted into C to verify the normal postcondition is the following:

```
1: public boolean C.checkPost$m$C(){
2:  return !α || γ;
3: }
```

The returned expression by postcondition checking method (line 2) is the Java code equivalent to the JML notation (\old(α) ==> γ). The AspectJ's advice that implements normal postcondition is the following:

```
1: void around(C current) :
2:  execution(void C.m()) && this(current){
3:    .../// saving all old values
4:    proceed();
5:    if(!current.checkPost$m$C()){
6:      throw new JMLPostconditionError();
7:    }
8: }
```

Note that this advice contains the postcondition testing method with the class name (line 5), because normal postconditions methods in subclasses preserve the postcondition of corresponding methods of their superclasses.This is expressed by the absence of within(C) that makes the advice affect executions of m in subtypes of C. It uses conjunction of normal postconditions by checking all the inherited postconditions. We use the around advice in order to manipulate data before and after executing the proceed method (necessary to treat JML old expressions not discussed here). The proceed method (line 4) represents the call to the original method from within the advice.

3.1.3 Invariant Mapping

Invariants are properties that have to hold in all visible states of objects [7]. Moreover, like postconditions, invariants are conjoined in the specification inheritance model. Leavens [6, Definition 2] states that the resulting invariant of C is the conjunction of the local invariant of C and the local invariants of all proper superclasses of C. Moreover, this leads to an instrumentation similar to preconditions. Thus, for a class C whose invariant assertion is θ, the following method is inserted into C:

```
1: public boolean C.checkInv$instance(){
2:  return θ && super.checkInv$instance();
3: }
```

The returned expression (line 2) is the conjunction of invariants (from the subtype and their supertypes).

In the JML semantics, the invariant test is performed at the beginning and at the end of every method (with normal or abnormal termination). Suppose that a method violates an invariant assertion during method call and terminates by throwing a *JMLPostconditionError*. In this case, the assertion violation error thrown must be kept. Only if the method terminates normally (when it holds the postcondition), the *JMLInvariantError* should be thrown. Note that if the exception thrown is not an assertion violation error, the invariant test must be performed. In order to instrument this semantics, we used before and the two kinds of after advice: after returning and after throwing advice. The after returning advice is applied when the method returns normally, with no exception thrown. On the other hand, when the executing method terminates by throwing an exception, the after throwing advice is applied to add behavior after it. These advices use the method super.checkInv$instance defined above and with a implementation similar to preconditions, provide the invariant test properly. However, the only difference is the use of the AspectJ designator execution with the clause execution(!static * *(..)). This clause provides the invariant test of every non-static method.

The invariants instrumentation that we described here, only checks instance invariants. In JML, static invariants are also allowed and the mechanism used to instrument it will be discussed in a future work.

4 Comparative Study

To evaluate our compiler, we have used the same application, a Java ME floating point calculator[2](MiDlet application [9]), in three different ways: (1) using our compiler based on AspectJ language (*CalcAspSol*); (2) using the original compiler [2] (*CalcJmlSol*); (3) a pure one, with no bytecode instrumentation (*CalcPureSol*). We compared the use of the three compilers by analyzing: MiDlet class size; bytecode size; library API size.

The Java ME floating point calculator application presents a calculator screen where the operands and operations are requested and the result shown. We annotated it with JML constructs presented here—which are fully supported by our compiler—to ensure two properties: it yields only positive results and it prevents division by zero.

When we compile the *CalcJmlSol* version by using the JML compiler (jmlc) setting the class path to the Java ME

[2] An open source Java ME application available at https://meapplicationdevelopers.dev.java.net/demo_box.html

Table 1. Java ME calculator application metrics sizes results

	MidLet class size (KB)	JAR size (KB)	Lib JAR size (KB)
CalcAspSol	21.1	11.8	4.6
CalcJmlSol	39.5	278.0	261.0
CalcPureSol	4.9	2.7	—

API [9], the bytecodes we obtain do not pass the analysis of the Java ME *preverifier* tool that checks bytecode compatibility to run in the Java ME environment. The reason for this failure is that Java ME does not support all features present in Java SE. Despite the incompatibility, we use the code in the comparative study.

Considering the three versions of the calculator application, we analyze the mentioned metrics with the same annotated input source file. Table 1 presents the result of the analysis. Concerning bytecode size, we observe that, *CalcAspSol* is 77.2% bigger than *CalcPureSol*, but 95.8% smaller than *CalcJmlSol*. In relation library API size, *CalcAspSol* showed to be 98.3% smaller than *CalcJmlSol*. This happens because *CalcAspSol* requires far less code than the original JML runtime to execute instrumented bytecode. In the case of the MiDlet class size, *CalcAspSol* is 76.8% bigger than *CalcPureSol* and 46.6% smaller than *CalcJmlSol*. Such results provide indication that our approach requires less memory space than the original JML compiler. As a proof of concept, we executed the calculator in a real mobile phone. We performed method calls with arguments that lead to precondition violation as specified by contracts. The application answered properly to these calls.

5 Related Work

Feldman [3] presents a DBC tool for Java, called *Jose*. This tool adopts a DBC language for expressing contracts. Similar to our approach, Jose adopts AspectJ to implement contracts. The semantics of postconditions and invariants in Jose are distinct from JML. Jose defines that postconditions are simply conjoined without taking into account the corresponding precondition. Moreover, it establishes that private methods can modify invariant assertions. In the JML semantics, if a private method violates an invariant, an exception must be thrown. Moreover, in order to preserve the JML semantics, we use **after returning** and **after throwing** advices, while the Jose tool only employs the **after** advice.

Pipa[10] is a behavioral interface specification language (BISL) tailored to AspectJ. It extends JML to specify and verify AspectJ programs. Differently, our work uses AspectJ to implement JML contracts in Java programs.

6 Conclusion and Future Work

This paper presents the implementation of a new JML compiler based on AspectJ. The paper explained how to use the aspect-oriented programming technique to implement JML assertions. For the best of our knowledge, this is the first work to use AspectJ with this purpose.

The main contribution of this paper is the use of aspect-oriented programming (AspectJ) for implementing contracts written in JML. The result was a bytecode compliant with both Java SE and Java ME applications. Thus, providing a way to extend the use of JML language to specify and verify Java ME applications. The comparative study conducted indicated that the overhead in size code is very small if compared with the code produced by the jmlc [2]. It is important to notice that code size is a very important metric in the context of Java ME applications.

Currently, we are working to implement the remaining JML constructors, such as exceptional postcondition.

References

[1] L. Burdy et al. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[2] Y. Cheon. *A runtime assertion checker for the Java Modeling Language*. Technical report 03-09, Iowa State University, Department of Computer Science, Ames, IA, April 2003. The author's Ph.D. dissertation.

[3] Y. A. Feldman et al. Jose: Aspects for design by contract80-89. *sefm*, 0:80–89, 2006.

[4] J. V. Guttag and J. J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[5] G. Kiczales et al. Getting started with aspectj. *Commun. ACM*, 44(10):59–65, 2001.

[6] G. T. Leavens. JML's rich, inherited specifications for behavioral subtypes. In Z. Liu and H. Jifeng, editors, *Formal Methods and Software Engineering: 8th International Conference on Formal Engineering Methods (ICFEM)*, volume 4260, pages 2–34, Nov. 2006.

[7] G. T. Leavens et al. Jml reference manual. Department of Computer Science, Iowa State University. Available from url http://www.jmlspecs.org, Apr. 2005.

[8] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[9] V. Piroumian. *Wireless J2me Platform Programming*. Prentice Hall Professional Technical Reference, 2002. Foreword By-Mike Clary and Foreword By-Bill Joy.

[10] J. Zhao and M. C. Rinard. Pipa: A behavioral interface specification language for aspectj. In *Proc. Fundamental Approaches to Software Engineering (FASE'2003) of ETAPS'2003*, Lecture Notes in Computer Science, Apr. 2003.