Implementing Java Modeling Language Contracts with AspectJ

Henrique Rebêlo
Sérgio Soares

Ricardo Lima
Leopoldo Ferreira

Márcio Cornélio

Department of Computing Systems
Pernambuco State University
Recife, Pernambuco, Brazil
{hemr,ricardo,marcio,sergio,lpf}@dsc.upe.br

ABSTRACT

The Java Modeling Language (JML) is a behavioral interface specification language (BISL) designed for Java. It was developed to improve functional software correctness of Java applications. However, the JML compiler explores the *reflection* technique and data structures not supported by Java ME applications. In order to eliminate such a problem, this paper proposes 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 includes a comparative study to demonstrate the quality of the final code generated by our compiler. The size of the code is compared against the code generated by an existent JML compiler. Moreover, we evaluate the amount of additional code required to implement the JML assertions in Java applications. Results indicate that the overhead in the code size produced by our approach is very small, which is essential for Java ME applications.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Aspect-Oriented Programming*; D.3.2 [**Programming Languages**]: Languages Classifications—*JML*

General Terms

Languages, Experimentation

Keywords

Design by contract, JML language, JML compiler, aspect-oriented programming, AspectJ

1. INTRODUCTION

Several specification languages have been designed to be annotated directly in the source code [6, 1, 11]. Most of these languages adopt the Design by Contract (DBC) [12] technique. This is the case of Java Modeling Language (JML) [11]. The JML compiler (jmlc) reads a Java program annotated with JML specification and produces an instrumented bytecode. Such additional code checks the correctness of the program against restrictions imposed by the JML specification.

Java ME [14] is intended for devices with limited resources such as handheld mobile devices. Many existing Java Standard Edition (Java SE) applications can be used in Java ME applications, but this code usually is not scaled down to fit limited hardware. Additionally, the Java ME API is a subset of the Java SE API geared toward handheld devices. Moreover, the implementation of the JML compiler explores reflection technology [13] to look inside Java objects at runtime. The lack of support for reflection represents an additional limitation for using JML with Java ME applications.

In order to overcome this limitation, this paper proposes an approach to implement a new JML compiler with support for Java ME applications. The strategy explores AspectJ [9] to implement JML's contracts (assertions). In fact, the approach leads to an instrumented bytecode compliant with both Java SE and Java ME applications.

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's contracts.

Jose [7] tool provides another specification language for applying DBC technique for Java. It also instruments their contracts through AspectJ. Moreover, in contrast with our work, Pipa [15] is a specification language that extends JML language to specify AspectJ classes and interfaces.

The contributions of this paper are:

- A novel JML compiler compliant with Java ME and Java SE applications;
- The usage of aspect-oriented programming in a different context than software development;
- An alternative JML compiler for applications in general (not only Java ME);
- A case study to investigate the proposed approach against the JML compiler proposed by Cheon [3].

This paper is structured as follows. Section 2 discusses the Java Modeling Language. Section 3 presents a novel JML

Preprint of:
Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, Sérgio Soares, and Leopoldo Ferreira. Implementig Java Modeling Language Contracts using AspectJ. Proceedings of the 23rd Annual ACM Symposium on Applied Computing , Fortaleza, Brazil, pp. 228-233, 2008.

compiler based on AspectJ. Some results of a comparative study between our and original JML approach 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) [11] is a behavioral interface specification language designed to Java. JML adopts design by contract (DBC) [12] and Hoare style [8]. JML specifications are composed of (pre-) postconditions and invariant assertions annotated in Java code in the form of comments.

JML compiler (*jmlc*) translates the annotated Java code into instrumented bytecode to check if the system respect the specification.

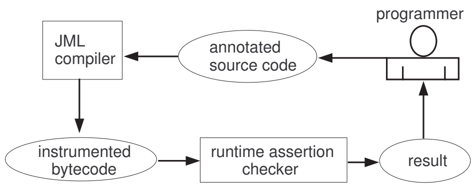

Figure 1: An overview of the JML environment

Figure 1 depicts an overview of 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> `@*/`. Figure 2 presents a JML specification, where the keywords **`requires`**, **`ensures`** and **`signals`** are respectively used to specify the precondition, the (normal) postcondition and the exceptional postcondition of the method. Moreover, the keyword **`invariant`** denotes a predicates that will always keep its truth value at the beginning and at the ending of every methods and the ending of constructor's execution. JML supports both instance and static invariants.

```
public class foo {
  //@ invariant S;
  /*@ requires P;
    @ ensures Q;
    @ signals (FooException) R;
    @*/
   public void foo() throws FooException {...}
}
```

Figure 2: Example of JML specification

2.1 The JML compiler

The JML compiler (*jmlc*) [3] was developed at Iowa State University. It is an assertion checker compiler, which converts JML annotations into runtime assertions.

Jmlc is built on top of the MultiJava compiler [4]. It reuses the front-end of existent JML tools [2] to verify the syntax and semantics of the JML annotations and produces a typechecked abstract syntax tree (AST). The compiler introduces two new compilation pass: the "runtime assertion checker (RAC) code generation"; and the "runtime assertion checker (RAC) code printing". The *RAC code generation* creates the assertion checker code from the AST. It may mutate the abstract syntax tree to add nodes for the generated checking code. Eventually, the *RAC code printing* writes the new abstract syntax tree to a temporary file.

For each Java method three *assertion methods* are generated into a temporary Java source file: one for precondition checking, and two for postcondition checking (for normal and exceptional termination). Finally, instrumented bytecode are produced by compiling the temporary Java source file through the MultiJava compiler.

The instrumented bytecode produced contains *assertions methods* code embedded to check JML's contracts at runtime.

3. A NOVEL JML COMPILER BASED ON ASPECTJ

This section describes a new approach for the implementation of the jmlc compiler based on AspectJ. The Cheon's approach [3] to implement JML has the following limitation to be employed for small devices:

- It adopts Java's reflection [13] to support specification inheritance and separate compilation. Java ME does not support reflection;
- It employs data structures, such as *HashSet* and *Map*, both from the java.util package, which are not supported by Java ME.

In the remaining of the section we present modifications in the JML compiler to overcome these limitations.

3.1 AspectJ Overview

AspectJ[9] is a general purpose aspect-oriented extension to Java. The main language construct is an *aspect*. Each aspect defines a functionality that crosscuts others (crosscutting concerns) in a system. An aspect can declare attributes and methods, and can extend another aspect by defining concrete behavior for some abstract declarations. An aspect can be used to affect both static and dynamic structure of Java programs by using AspectJ's *static/dynamic crosscutting mechanism.* The *static crosscutting mechanism* allows one to introduce new methods and fields to an existing class, convert checked exceptions into unchecked exceptions, and change the class hierarchy. On the other hand, dynamic crosscutting mechanism changes the way a program executes. It intercepts specific points of the program execution flow, called *join points*, adding behavior *before*, *after*, or *around* the join point.

3.2 The implementation strategy

Similar to Cheon [3], we reuse the front-end of the JML Type Checker [2]. We traverse the typechecked AST generating *Aspect Assertion Methods* (AAM) for each Java method. This version of the compiler considers precondition checking, normal postcondition checking and invariants. 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.

3.3 Mapping contracts to Aspects

In this section, we present a mapping of JML into *aspects* [9]. We concentrate on precondition, normal postcondition and invariant. For the mapping rules, consider the following JML sample annotation:

```
public class C extends B{
 //@ invariant θ;
 /*@ also
   @ requires α;
   @ ensures γ;
   @*/
   public void m(){...}
}
```

This sample provides a class `C` that inherits class `B` with an overridden method `m` that has the precondition α, postcondition γ, invariant θ and possible inherited specifications indicated by JML keyword `also`.

3.3.1 Precondition Mapping

In JML preconditions must be satisfied before any code of the method is executed, otherwise an exception must be thrown informing the precondition violation. The AspectJ code that checks the precondition of the method `m` is:

```
1: before() :
2:  execution(void m())) && within(C) {
3:    if(!α){
4:      throw new JMLInternalPreconditionError();
5:    }
6: }
```

This piece of code declares a `before` advice (line `1`) that is responsible to insert (instrument) an extra behavior (lines `3`, `4` and `5`) before some specified points in the Java program. In this example, 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 `3`) and throws `JMLInternalPreconditionError` (line `4`) if it is violated. Assertion violations in the JML semantics are instances of java.lang.Error [3].

However, this mapping in the previous code ignores any inherited contract. In JML, subclasses inherit not only fields and methods from their superclasses, but also specifications (*specification inheritance model*). To achieve this, the JML keyword `also`, performs specification inheritance (see the use of `also` at the JML specification example in the beginning of Section 3). Thus, specifications can be added to an overridden method. The semantics of "also" is stated by Leavens (Definition 1) in [10]. According to this definition, the result of precondition inheritance is its disjunction in the form ($\mathtt{pre}'$ || $\mathtt{pre}$), where $\mathtt{pre}'$ is the method precondition defined by the subclass and $\mathtt{pre}$ is the method superclass 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: }
```

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

this code inserts a method into class `C` (the `C.` prefix tells AspectJ where to insert the method definition). This inserted method checks its precondition in conjunction with the proper inherited precondition (`super.checkPre$m()`), if any, as defined by the precondition inheritance definition [10].

The AspectJ's advice becomes:

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

In this code, the advice parameter `current` (line `1`) indicates that we want to expose some information about the execution context. The designator `this` (line `2`) exposes the currently executing object, which is the object executing the method `m`.

3.3.2 Normal Postcondition Mapping

Postconditions are properties in JML that must be true after method execution. In a specification inheritance model, The semantics of "also" given by Leavens [10] (Definition 1) states that postconditions as a conjunction of implications in the form ($\backslash\mathtt{old(pre')}$==>$\mathtt{post}'$) && ($\backslash\mathtt{old(pre)}$==>$\mathtt{post}$). Expressing that when one of the preconditions holds (at beginning of method execution), then the corresponding postcondition must hold. The expression ($\backslash\mathtt{old(pre')}$) corresponds in JML to the evaluation of the precondition before to the method execution.

In JML, there are two kinds of postconditions: normal and exceptional postcondition. Here, we cover only normal postcondition. Exceptional postcondition will be treated in a future work.

The method inserted 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 ($\backslash\mathtt{old}(\alpha)$ ==> γ). Thus, the expression !α (line `2`) is evaluated with values of beginning method execution, according to JML's old expression semantics (pre-state).

The AspectJ's advice that implements this 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 JMLInternalNormalPostconditionError();
7:    }
8: }
```

In the above code, the method name that tests the postcondition (line `5`) contains the class name, 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 to affect executions of `m` in subtypes of `C`. It uses conjunction of normal postcondition testing all the inherited postconditions. We use the `around` advice in order to have total control over the computation [9]. In this way, we can manipulate data before and after executing the `proceed` method.

The constructor `proceed` (line 4) represents the call to the original method from within the advice. The `around` advice is necessary to treat JML `old` expressions by saving all state variables (attributes used in `old` expressions) (line 3) before the method executes. Consequently, we evaluate the postcondition checking method in a proper context allowing references to state values (attributes) in their pre-state (values before method execution). One way to achieve this is to use *inter type declaration* to insert attributes to retain the values of attributes present in class before method execution (line 3). Thus, we can use attributes appropriately in the expression $!\alpha$ (see line 2 of postcondition checking method defined previously).

3.3.3 Invariant Mapping

Class invariants express global properties of a class that must be preserved by all class routines. Moreover, like postconditions, class invariants are conjoined in the specification inheritance model. Leavens (Definition 2) [10] states that inherited specifications, such as invariants, can be explained by constructing an extended specification. The Leavens' definition is as follows:

$$ext\_inv^{T} = \bigwedge\{\ added\_inv^{U}\ \mid U\ \in\ supers(T)\}$$

This definition states that an extended invariant of T is the conjunction of all added invariants in T and its proper supertypes. Moreover, this leads to an instrumentation similar to preconditions. Thus, for each class `C` whose invariant assertion is θ, in order to obey the definition of invariant extended specification, the following method is inserted into `C`:

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

In this code, the returned expression (line 2) is the conjunction of invariants (from the subtype and their supertypes).

In the JML compiler (jmlc) [3], the following steps are performed to invoke a method: (1) invariant test in the beginning of invoked method (pre-state); (2) precondition test; (3) postcondition test; (4) invariant test in the end of invoked method (post-state). If one current step is violated, a proper assertion error is thrown. In order to instrument these steps, 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, without throwing any exception. On the other hand, when the executing method terminates by throwing an exception, the `after throwing` advice is applied to add behavior after it.

```
1: before(C current) :
2:  execution(!static * C.*(..)) && within(C) &&
3:  this(current){
4:    if (!current.checkInv$Instance()){
5:      throw new JMLInvariantError("<@pre>");
6:    }
7: }

8: after(C current)returning(Object o) :
9:  execution(!static * C.*(..)) && within(C) &&
10:  this(current){
11:    if (!current.checkInv$Instance()){
12:      throw new JMLInvariantError("<@post>");
13:    }
14: }
15: after(C current)throwing(Throwable throwable) :
16:  execution(!static * C.*(..)) && within(C) &&
17:  this(current){
18:    if(throwable instanceof σ){
19:      throw(σ)throwable;
20:    }
21:    if(throwable instanceof π){
22:      if (!current.checkInv$Instance()){
23:         throw new JMLInvariantError("<@post>");
24:      }
25:      else{
26:         throw(π)throwable;
27:      }
28:    }
29: }
```

If no assertion violations occur by any steps discussed above, the `after returning` advice is executed. It checks the invariants (line 11) and if it detects any invariant violation then it throws a `JMLInvariantError` (line 12), otherwise returns normally. if an assertion violation occurs, the `after throwing` advice is properly executed. This advice verifies which kind of exception was threw, and if any kind of JML assertion violations is thrown (represented by the letter σ) (line 21), the advice re-throws it. However, if the threw exception is a Java exception (represented by the letter π) (line 21), the advice checks the invariants (line 22) and if they are satisfied, re-throws the Java exception (line 26), otherwise throws a `JMLInvariantError` (line 23).

The presented invariants instrumentation only checks instance invariants of all non-static methods of the class `C` (lines 2, 9 and 16), because static methods are not allowed to access instance fields. However, in JML, static invariants are also allowed and the mechanism used to instrument it is very similar to the one presented here. However, some modifications are needed: (1) insert a proper method definition that tests static invariants (`checkInv$Static()`); (2) change the `!static` clause (lines 2, 10 and 18) to `static`; (3) call within the advices the proper method that tests static invariants (`checkInv$Static()`); (4) insert the same advices presented without the parameter `current` (lines 1, 8 and 15) to check static blocks that the class `C` may contain and to use the designator `staticinitialization(C)` instead using the designators `execution`, `within` and `this`.

4. COMPARATIVE STUDY

To evaluate the impact of our approach using aspects in constrained environment devices, we have used the same application, a Java ME floating point calculator[2](MiDlet application [14]), in three different ways: using our approach with the AspectJ language (*CalcAspSol*); using the original approach [3] (*CalcJmlSol*); and a pure one, with no bytecode instrumentation (*CalcPureSol*). We compared the three approaches by analyzing the metrics: MiDlet class size; bytecode size; library API size.

The Java ME floating point calculator application presents a calculator screen where the operands and operation are requested and the result shown. It was annotated with JML constructions presented here to ensure two implementation properties: it yields only positive results; it prevents division by zero operation.

We compile the *CalcJmlSol* version by using the JML compiler (jmlc) setting the class path to the Java ME API [14].

[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	—

However, 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*. Considering 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 strong indication that our approach requires less memory space than the original JML compiler. Moreover, the overhead in the size of the bytecode to check JML assertions at runtime is acceptable. In order to validate our JML compiler, we executed the calculator application in a real mobile phone.

5. RELATED WORK

In [7], Feldman presents a DBC tool for Java, called *Jose*. The tool adopts a proprietary DBC language. 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 present. Moreover, it stablishes that private methods are can modify invariant assertions. In JML's semantics, if a private method violates an invariant, an exception must be thrown. Moreover, in order to preserve the JML semantics (see Section 3.3.3), our approach checks invariants at end of every method invoked. We use **after returning** and **after throwing** advices, while the Jose tool only employs **after** advice.

Pipa[15] is a behavioral interface specification language (BISL) tailored to AspectJ. It extends JML language to specify AspectJ classes and interfaces. The goal is to facilitate the use of existing JML-based tools to verify AspectJ programs. Different from Pipa, our work uses AspectJ to implement JML assertions in Java programs.

6. CONCLUSION AND FUTURE WORK

This paper presents the implementation of a new JML compiler based on AspectJ. The compiler translates JML contracts annotated in the Java source file into AspectJ aspects. The result is an instrumented bytecode with embedded aspect checking methods that verify the program against JML specification. The paper explained how to use aspect 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 our work is to extend the use of JML to Java ME applications. The comparative study conducted here indicated that our JML compiler produces a code of better quality than the jmlc [3], when we analyze the size of the code generated for both compilers. Such a result is essential when considering the Java ME environment. Moreover, the overhead in the size of bytecode to check the JML assertions at runtime is fairly acceptable. We executed the calculator application produced by our JML compiler in a real mobile phone.

We plan to conduct experiments using the AspectJ which implements optimization techniques for AspectJ advices [5].

The JML implementation presented here was based on JML-5.4 version[3]. This first version focused on the main JML constructors. The compiler supports precondition, normal postcondition and invariants. We are working to implement the remaining constructors, such as exceptional postcondition.

7. ACKNOWLEDGMENTS

We would like to thank CAPES and FINEP, brazilian research agencies, that partially supported this work.

8. REFERENCES

[1] M. Barnett et al. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362, pages 49–69, 2005.

[2] 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.

[3] 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.

[4] C. Clifton et al. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM Press.

[5] E. Cordeiro et al. Optimized compilation of around advice for aspect oriented programs. *j-jucs*, 13(6):753–766, 2007.

[6] F. de Oliveira Jr. et al. Cml: C modeling language. *j-jucs*, 13(6):682–700, 2007.

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

[8] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

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

[3]The implementation is available at http://sourceforge.net/projects/jmlspecs

[10] 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.

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

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

[13] S. MicroSystems. Java 2 platform, standard edition, v 1.5.0 api specification. Available from http://java.sun.com/j2se/1.5.0/docs/api/ (Date retrieved: August 20, 2007).

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

[15] 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.