JML and Aspects: The Benefits of Instrumenting JML Features with AspectJ

Henrique Rebêlo
Sérgio Soares
Department of Computing and Systems
University of Pernambuco
Recife, Pernambuco, Brazil
{hemr,sergio}@dsc.upe.br

Ricardo Lima
Paulo Borba
Informatics Center
Federal University of Pernambuco
Recife, Pernambuco, Brazil
{rmfl,phmb}@cin.ufpe.br

Márcio Cornélio
Department of Computing and Systems
University of Pernambuco
Recife, Pernambuco, Brazil
{marcio}@dsc.upe.br

ABSTRACT

The Java Modeling Language (JML) is used to specify designs of Java classes and interfaces. To this end, JML has a rich set of features for specifying methods, including specification inheritance. Thus, the most fundamental motivation for employing JML is to improve functional software correctness of Java applications, and helps to reduce corrective maintenance effort of those applications. Previously, we presented a new JML compiler (ajmlc) that generates aspects (AspectJ) for contract enforcement. This paper describes the main reasons to instrument JML features with AspectJ, with particular emphasis on issues related to instrumentation code size — we also defined guidelines to use ajmlc that always generate compact instrumented code than the classical JML compiler (jmlc). In addition, we discuss the analogy between JML and AspectJ, and how the ajmlc also deals with Java ME applications, which is not possible with jmlc. Moreover, we implemented other JML features such as the new the new assertion semantics based on "strong validity" presented elsewhere. The paper includes studies to compare the final code generated by ajmlc with the one produced by jmlc. Results indicate that the overhead in code size produced by our compiler is very small when using the proposed guidelines, 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, JML new assertion semantics, aspect-oriented programming, AspectJ, AspectJ weaving

1. INTRODUCTION

The Java Modeling Language (JML) [19, 18] is a formal behavioral interface specification language for Java. The JML compiler (jmlc) reads a Java program annotated with JML specification and produces instrumented bytecodes. Such additional code checks the correctness of the program against restrictions imposed by the JML specifications.

In a previous work [24], we proposed a new JML compiler known as ajmlc (AspectJ JML compiler). The ajmlc employs AspectJ [14, 15] (generative programming) to implement JML features (specifications) without generics. The AspectJ compiler translates such features into an instrumented bytecode, which performs a runtime checking of those features. The ajmlc generates code compliant with both Java SE and Java ME [23] applications. Similarly, Jose [10] is a tool that uses AspectJ to instrument Java programs. Jose tool adopts a different specification language to specify Java programs. Thus, its semantics is different from that of JML. However, the instrumentation provided by the Jose tool is not complaint with Java ME applications, whereas ajmlc does. Moreover, the language JCML [9] is a subset of the JML language targeting Java Card applications. Different from ajmlc, the JCML compiler does not use Aspect Oriented Programming. As with the original JML compiler, it instruments JML features using standard Java code with a few restrictions imposed by the Java Card platform.

Based on our previous results and the new results addressed by this paper, we try to answer properly the following research questions:

- Does AOP represent the JML features (specifications) conveniently?
- When is it beneficial to aspectize JML features in relation to both source and bytecode instrumentation? When it is not?
- How to check JML features during runtime?
- How to modify properly the code generation of the JML compiler for generating aspects that support runtime assertion checking of JML features in Java ME applications (in a constrained environment)?
- What is the relationship between the generated aspects?
  What is the order of aspects generation to provide an effect like the *wrapper approach* present in the classical JML compiler (jmlc)?

The main contributions of this paper are: (1) answering the above research questions throughout the paper; (2) describing the analogy between JML and aspects (AspectJ); (3) extending ajmlc to support the new assertion semantics; (4) generating instrumented bytecode to verify constrained methods during class initialization (this feature is not supported by jmlc); (5) generating instrumented bytecode when necessary (this feature is also not supported by jmlc); (6) Conducting study between ajmlc with two different AspectJ weavers and the original JML compiler to investigate the overhead in the code size.

This paper is organized as follows. The next section 2 presents the background of Java Modeling Language (JML). Section 3 describes the ajmlc compiler as well as its new issues addressed. Some results of conducted studies between our and original JML compiler are discussed in Section 4. Section 5 discusses related work. The last section contains the conclusions and points directions for future work.

2. JML: BACKGROUND

The Java Modeling Language (JML) [19, 18] is a specification language to describe the expected behavior of Java modules — Java modules are classes and interfaces. It combines the Design By Contract (DBC) approach [21] of Eiffel [22] and the model-based specification approach of the Larch family [12] of interface specification languages, with some elements of the refinement calculus. Hence, JML specifications contains pre-, postconditions, and invariant predicates based on Hoare-style [13].

Java comments beginning with the symbol @, which are interpreted as JML annotations (see example in Figure 1). One can use JML to specify the behavior of types (type specifications) or methods (method specifications). An `invariant` clause is a type specification. For instance, the predicate `I` denotes an invariant condition that must be true after the execution of all constructors of the class `Foo`. Moreover, this predicate is supposed to be true before and after the execution of all methods of the class `Foo`. On the other hand, `requires`, `ensures`, and `signals` clauses represent method specifications. `Requires` specify the method precondition, whereas `ensures` and `signals` are respectively used to define the normal and exceptional postconditions of the method `foo`.

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

Figure 1: Example of JML specification.

2.1 JML assertion semantics

The current JML compiler implements the assertion semantics based on "strong validity" proposed by Chalin's work [25]. Thus, instead of using the classical two-valued logic in the older approach proposed by Cheon's work [4], the JML compiler uses now a three-valued logic semantics. In this way all logical operators behave similarly in both Java and JML. Using the new semantics, an assertion can be satisfied (true), violated (false) or invalid (when evaluation does not complete successfully). More details about the new assertion semantics, refer to [25].

2.2 The JML compiler

The JML compiler (*jmlc*) [3] was developed at Iowa State University. It is a runtime assertion checking compiler that converts JML annotations into automatic runtime checks.

Design

Jmlc is built on top of the MultiJava compiler [6]. It reuses the front-end of existenting 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 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 AST to a temporary Java source 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). They are invoked before method call (precondition checking), after method call (normal postcondition checking) and when an exception is thrown by the called method (exceptional termination checking). Finally, instrumented bytecode is produced by compiling the temporary Java source file through the MultiJava compiler. The instrumented bytecode produced contains *assertion methods* code embedded to check JML contracts at runtime.

Wrapper approach

The *wrapper approach* [3, 4.1.3] is a strategy used by the JML compiler to implement the assertion checking. Each method is redeclared as private with a new name. Then, a method known as *wrapper method* is generated with the name of the original method. Its surrounds the original method (now with a new name) with the assertion methods. Hence, client method calls the wrapper method, which is responsible for calling the original method with appropriate assertion checks (e.g., precondition checking). The JML compiler is responsible for controlling the order of execution of assertion methods.

Figure 2 depicts the wrapper approach strategy. If a client calls the original method, the call goes to the wrapper method. In this way, the precondition assertion method is the first assertion method called, and then only if the precondition is satisfied, it calls the original method. After calling the original method, if it terminates normally, the normal postcondition assertion method is called; otherwise, the exceptional postcondition assertion method will be called.

3. AJMLC: A JML COMPILER TARGETING ASPECTJ CODE

In this section we present the analogy of JML and AspectJ aspects. We explain the reason to aspectize JML features. The remaining reasons only will be understood in the Sec-

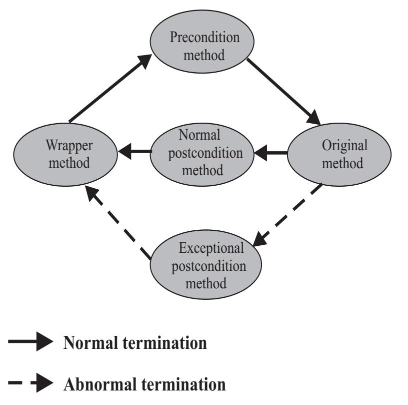

Figure 2: Wrapper approach strategy.

tion 4 with the studies results. Furthermore, we concentrate on new assertion semantics verification recently provided by the ajmlc, the reason that ajmlc can be used with Java ME applications, and among other issues. Some implementation details of our compiler will also be considered. A detailed implementation mechanism has already been available in a previous work [24].

3.1 AspectJ Overview

AspectJ [14, 15] is a general purpose aspect-oriented extension to Java. The aspect-oriented constructs support the separate definition of units of a program which affect (crosscut) other concerns. Such units are called crosscutting concerns. These concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and result in either scattering or tangling code, or both. Thus, this separation of concerns allows better modularity, avoiding tangled code and code spread over several units. Consequently, the system maintainability is also increased. Programming with AspectJ explores both objects and aspects concepts to separate concerns. Object-oriented programming can be used when the concern are well modeled as objects. If it is not the case, concerns that crosscut the objects are separated using units called aspects, and those are composed with the objects of a system by a process called weaving. By weaving AspectJ aspects with standard Java code, we obtain a new AspectJ application.

The main construct of the AspectJ [14, 15] language is called *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 affect both static and dynamic structure of Java programs. The static structure might be changed by introducing new methods and fields to an existing class, as well as converting checked exceptions into unchecked exceptions, and changing the class hierarchy. The dynamic structure is changed by intercepting specific points, called join points, of the program execution flow and adding behavior before, after, or around the join point.

3.2 Ajmlc design

Similarly to Cheon [3], we reuse the front-end of the JML compiler, known as JML Type Checker [2]. Then, we modify the code generation part of the original JML compiler[1] to introduce other two new compilation passes: the *Aspect RAC* code generation; and the *Aspect RAC* code printing. The former produces assertion checker code from the typechecked AST, whereas the latter writes the assertion checker code to a temporary Aspect source file. We traverse the typechecked AST generating *Aspect Assertion Methods* (AAM) for each Java method in a temporary Aspect source file: one for precondition checking, and another for both kinds of postconditions in JML (normal and exceptional). Eventually (when necessary) we also generate AAM for both kinds of invariants in JML (instance and static). These AAM are compiled through the AspectJ compiler (ajc or abc [1]), which weaves the AAM with the Java code. The result, unlike jmlc, is an instrumented bytecode compliant to both Java SE and Java ME applications.

3.3 Ajmlc runtime environment

The instrumented bytecode produced by the ajmlc contains not only its normal content (usually generated by javac), but also has embedded code (assertion methods) to checks JML's features during runtime.

In order to run and check those assertion methods of the bytecode generated by ajmlc, we use part of the AspectJ runtime environment library (answer to research question 3 discussed in Section 1). We need only part of the AspectJ library, because only a few AspectJ constructs must be used during runtime. This is due to compatibility needed by the instrumented code to deal with Java ME applications. For more information about ajmlc with Java ME applications and its required AspectJ library refer to section 3.7 and 4.

3.4 The analogy between JML and Aspects

As pointed out by Filman and Friedman [11], the Figure 3, is an example of quantification. Aspect-oriented programming languages such as AspectJ [14, 15] allow programmers to define quantified programmatic assertions.

To this end, AspectJ provides property-based crosscutting to affect from small to a large number of Java modules (e.g., classes, interfaces, and methods). To perform such property-based crosscutting, by using AspectJ, one can use a feature known as wildcarding ($*$) in pointcut designators. Consider the following example:

```
execution(* T.*(..))
```

This AspectJ construct identifies executions to any method (with any return and any parameters type) defined on type `T`.

The invariants analogy

The behavior of quantification can be addressed similarly by using JML invariants. For example the JML instance invariants must be satisfied by all instance methods of the current type and also subtypes (quantification). In Figure 3, we have a behavior of instance invariant checking by using aspects (note the use of wildcards). On the other hand, if we use pure JML, the following clause replaces both `before` and `after` AspectJ advices [14, 15] depicted in Figure 3.

```
//@ instance invariant i == 10;
```

[1]Part of the code of the original JML compiler that we used to implement the ajmlc was based on the JML 5.5 version available to download at `http://sourceforge.net/projects/jmlspecs`.

As mentioned before, this JML clause defines a quantification property (`i == 10`) that must hold by all instance methods in type `T` and also in its subtypes (see Figure 3).

Behavioral subtyping analogy

Regarding specification inheritance in JML, instance methods with specification cases (e.g., pre- and postconditions) must be satisfied by the current type and also subtypes. For example, suppose a scenario with a JML precondition declared in method `m` of type `T` and we have an subtype of `T` (that extends it) called `S` that overrides the method `m` with other specification cases. Thus, if we have an object of type `S`, we must satisfy the current specification cases of method `m` in combination with the inherited ones (from type `T`), resulting in disjunction for preconditions and conjunction for postconditions [17]. Thus, we can also specify this behavior know as behavioral subtyping using aspects — aspects that checks conditions (assertions) defined in the specification cases locally by the method `m` in combination with the inherited conditions (from the type `T`).

Other analogies

We discussed two points in JML and AspectJ that their behavior work in the same way. We also showed and argued that those points identified in JML are in fact quantification points that can be implemented using AspectJ. However, there are other quantification points in JML that certainly can be expressed with AspectJ. Examples of such quantification points [19] that can be found in JML not limited to:

- instance and static constraint specifications (is a JML type specifications like invariants);
- refinement;
- model-programs;
- non-functional properties;
- so forth.

AspectJ and JML a perfect match

Based on the above argumentations, we showed that JML has properties that cutting across several modules — the concern know as contract enforcement present in JML which is classified as a crosscutting concern [20, 7]. Since AspectJ provide means to deal with crosscutting properties, we conclude that AspectJ can implement properly various JML features (answer to research question 1 discussed in Section 1).

Feldman's work [10] provides another evidence that AspectJ can be used to implement contract enforcement concern [20, 7]. Section 5 presents the main points related to such a work.

3.5 Expression evaluation with new assertion semantics

Ajmlc was restructured to deal with the new assertion semantics proposed by Chalin's work [25] and implemented by the current JML compiler. Considering this semantics, a clause can be entirety executable or not. In this way, we generate into aspects two try-catch blocks:

- one to handle non-executable exceptions discovered at runtime;

```
public class T {
    int i = 10;

   public void m() {...}
   public void n() {...}
   public void o() {...}
}

privileged Aspect_T{
  before (T current) :
    execution(!static * T.*(..)) &&
    within(T+) &&
    this(current){
      if (!(current.i == 10)) {
        throw new RuntimeException("");
      }
    }

  after (T current) :
    execution(!static * T.*(..)) &&
    within(T+) &&
    this(current){
      if (!(current.i == 10)) {
        throw new RuntimeException("");
      }
    }
}
```

Figure 3: Example of AspectJ quantification.

```
public class T{
  public int x, y;
  //@ requires b && x < y;
  public void m(boolean b)
  { ... }
}
```

Figure 4: Simple method with a precondition.

- another to handle all other exceptions, such as *NullPointerException* raised during assertion checking by a method.

In order to see an example of this approach, consider a method `m` declared in a type `T` with a simple precondition (see Figure 4).

An AspectJ `before advice` [14] is generated by the ajmlc to instrument such a precondition. This `before` advice contains the above mentioned two try-catch blocks. In Figure 5, we can observe the resulting instrumentation code generated by ajmlc. Note that the presence of the `JMLEvaluationError`, which is a new JML assertion error [25] responsible for handling invalid assertion evaluations.

3.6 Ordering of advice executions into an aspect

One AspectJ aspect can have several advices (e.g., `before`) to apply to a particular named or anonymous pointcut. As the advices are declared into the same aspect, we should take into account their order declaration. In this way, the advice that appears first lexically inside the aspect executes

```
public boolean T.checkPre$m$T(boolean b){
  return ((b) && (x < y));
}

before (T current, boolean b) :
   execution(void T.m(boolean)) &&
   within(T) &&
   this(current) && args(b) {
     boolean rac$b = true;
     try {
       rac$b = current.checkPre$m$T(b);
       if(!rac$b){
         throw new
         JMLInternalPreconditionError("");
       }
     } catch (JMLNonExecutableException
       rac$nonExec) {
         rac$b = true;
     } catch (Throwable rac$cause) {
         if(rac$cause instanceof
            JMLInternalPreconditionError) {
           throw (JMLInternalPreconditionError)
                 rac$cause;
         }
         else {
           throw new JMLEvaluationError("");
         }
     }
   }
```

Figure 5: Evaluation of precondition in the new assertion semantics.

first. "The only way to control precedence between multiple advice in an aspect is to arrange them lexically [16]." Thus, ajmlc generate AspectJ advices carefully in order to respect the JML semantics (answer to research question 5 discussed in Section 1). The order of generation is as follow:

1. generate a `before` advice to check static invariants;
2. generate a `before` advice to check instance invariants;
3. generate `before` advice to check preconditions of existing methods (including constructors);
4. generate `after returning` advices and `after throwing` advices or `around` advices (if we have old expressions) to check postconditions (normal and exceptional postconditions) of existing methods (including constructors);
5. generate a `after returning` and a `after throwing` advices to check instance invariants;
6. generate a `after returning` and a `after throwing` advices to check static invariants;

The above ordering to generate AspectJ code is extremely important to keep the classical ordering of contract checking posed by JML semantics. For example, a method to be executed must obey some conditions in a certain order:

1. check invariants (static and instance invariants) before method execution;
2. check preconditions before method execution;
3. check postconditions after method execution (normal postconditions when the method terminates normally and exceptional postconditions when the method terminates abnormally);
4. check invariants (static and instance invariants) after method execution.

These ordering is respected by generated aspects to check JML features during runtime — such aspects ordering have an analogy with the Cheon's *wrapper appraoch* [4], because they have the same effect during runtime checking (ordering to call the assertion methods, such as precondition checking method).

3.7 Ajmlc and Java ME applications

The main benefit in using ajmlc is that one can specify and verify during runtime Java ME applications [23] with JML. To this end our compiler only generates aspects that avoids AspectJ constructs that are not supported by Java ME, such as `cflow` pointcut [14, 16] (answer to research question 4 discussed in Section 1).

3.8 Ajmlc optimizations

Concerning Java ME applications, we introduced several optimizations in ajmlc in order to generate small instrumentation code as much as possible due to constrained environments like Java ME platform.

Compiling empty classes

The jmlc compiler assumes a standard configuration for classes. Thus, even if one defines an empty class, basic instrumentation is generated [19, 18] for:

1. class verification
   - Static and non-static invariant/constraint checking;
   - Static and non-static constraint pre-state expressions checking.
2. default constructor verification
   - Assertion checking wrapper;
   - Precondition checking;
   - Normal postcondition checking;
   - Exceptional postcondition checking.
3. other methods (e.g., for dynamic calls using reflection)

In this way, the jmlc compiler generates 11.0 KB (source code instrumentation) and 5.93 KB (bytecode instrumentation) even for a empty class like:

```
public class Empty { }
```

In contrast to jmlc compiler, our compiler does not generate code for empty classes.

Code instrumentation

Code size is an important issue for Java ME applications. Our compiler avoids code generation as much as possible. Table 1 compares the jmlc and ajmlc compilers when no specification is provided.

Limitation of the jmlc compiler solved by the ajmlc compiler

The current implementation of the jmlc compiler has one limitation:

JML clauses	jmlc generates	ajmlc generates
requires	yes	no
ensures	yes	no
signals	yes	no
invariant	yes	no

Table 1: Difference between jmlc and ajmlc during the generation code.

```
public static x;
//@ static invariant x > 0;

public static void m() { x = -3; }

static{
  m();
}
```

Figure 6: Example of non checked type invariant when it is called.

1. When constrained methods are called into static blocks during the class initialization, jmlc does not check the constrains and the method is always executed even if the condition is `false`.

Figure 6 shows an example where the method `m` is constrained with the invariant (`x > 0`) and a call (`m()`) that violates the invariant is made inside a static block. As a result, no assertion violation is raised. Cheon's compiler [3] does not generate instrumented bytecode properly to deal with this limitation. However, the ajmlc always verifies constrained methods when called into static blocks. This benefit is automatically gained just by using aspects to instrument the JML features.

4. STUDY

In our previous work [24], we evaluated our compiler (ajmlc) by using a Java ME application. This was fundamental to investigate our proposed approach in a Java ME environment. In this section we evaluate our compiler employing three Java applications. Such applications was extracted from the JML literature, as described bellow.

Scenario

We have compiled three Java programs annotated with JML using both ajmlc (our compiler), and the jmlc compiler (Cheon's compiler [3]). Such programs are described in three works: (1) the hierarchy classes `Animal`, `Person`, and `Patient` [17]; (2) the class `IntMathOps` [19], and (3) the class `StackAsArray` [3]. Moreover, we have used our ajmlc with two different weaving processes: using the standard AspecJ compiler (ajc) [14]; and the abc compiler [1], which is a complete implementation of AspectJ with some optimizations.

As a important point for our study, we removed the JML specifications from the class `Person`. This choice is to show that our compile only generate instrumented code when necessary.

Results

Considering the scenario described above, Table 2, Table 3 and Table 4 present the results of the compilation size that we obtained by using both compilers. As can be seen, we analyzed instrumented source code size, instrumented bytecode size, and Jar size all in kbytes (KB). Considering our compiler (ajmlc), we used the same AspectJ aspect code (source) generated for both weaving processes (using ajc, and abc compilers). We observed that the ajmlc compiler using the ajc weaver introduces a big overhead in the instrumented bytecode size (see Table 3), whereas in relation to instrumented source code size, ajmlc generates a smaller code (see Table 2). On the other hand, our approach produces a far smaller instrumented bytecode and source code when the abc weaver is employed (see Tables 3 and 2). Concerning Jar size we observed that the final deployed applications for both ajmlc with ajc and abc weavers are smaller to ones related to the jmlc. This happens because the lib Jar size necessary to evaluate assertions during runtime for our compiler is smaller than the lib Jar size for jmlc. It is important to note that we take into account only the JML features available by our compiler. Thus, we removed the from the jmlc runtime library the part that is nor supported yet by our compiler — this gives a more fair comparison. Therefore, the user is free to choose the AspectJ weaver. However, based on the results, we recommend the usage of ajmlc with the abc weaver (for most cases). This choice is particulary important for Java ME applications.

Guidelines

Based on the results presented in Tables 2, 3, and 4, we briefly present steps to use our ajmlc compiler (answer to research question 2 discussed in Section 1). These steps are the guidelines for its usage:

1. If the application is not compiled in its entirety by the JML compiler — if at least $\approx 33\%$ of the application is free of the JML instrumentation effect (as occurs with the Hierarchy application showed above), we recommend to use ajmlc with both ajc or abc weavers. Even if the application is a Java ME application. But, be aware that by always using abc we got better results;

2. If the application is compiled in its entirety by the JML compiler — we recommend only in the case of Java ME applications to use ajmlc with abc weaver;

3. If the user always need to take maximum of the AspectJ optmization — we alwys recommend to use ajmlc with abc weaver.

These guidelines is to provide a way to choose the best AspectJ weaver to use. Although, our compiler always need smaller memory space during deploying (because the discrepancy of the size of the runtime libraries from ajmlc and jmlc).

5. RELATED WORK

JMLC (Java Card Modeling Language) [9] is a is a subset of the JML language. The JCML compiler (jcmlc) generates bytecode compliant with Java Card applications. However, its instrumentation does not employ AspectJ to implement the JML contracts. The jcmlc translates only JML lightweight specifications, whereas our compiler handles both lightweight and heavyweight specifications. The jcmlc does not support inheritance of specifications, which our compiler

Table 2: Instrumentation source code size results

	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
Animal	28.8	4.8	4.8
Person	27.4	0.5	0.5
Patient	26.2	9.6	9.6
IntMathOps	18.2	2.0	2.0
StackAsArray	55.7	9.2	9.2

Table 3: Instrumentation bytecode size results

	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
Animal	13.3	17.0	5.5
Person	**11.7**	**2.3**	**0.7**
Patient	12.7	25.3	7.4
IntMathOps	9.39	5.4	2.3
StackAsArray	21.7	23.2	6.2

does. On the other hand, the jcmlc handles quantifiers such as `forall`, which are not treated by our compiler.

Feldman *et al.* [10] presents a DBC tool for Java, known as *Jose*. This tool adopts a private DBC language for expressing contracts. Similar to our approach, Jose adopts AspectJ for implementing contracts. The semantics of postconditions and invariants in Jose are distinct from JML. Jose states that postconditions are simply conjoined without taking into account the corresponding preconditions. 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. Unlike our compiler, Jose generates bytecode not compliant with Java ME.

Pipa [26] is a behavioral interface specification language (BISL) tailored to AspectJ. It uses the same approach (based on annotations) of JML language to specify AspectJ classes and interfaces, and extends JML with a few new constructs in order to specify AspectJ programs. The Pipa language also supports aspect specification inheritance and crosscutting. Pipa specifies AspectJ programs with pre-, postconditions, and invariants. Moreover, Pipa also can specify aspect invariants and the "decision" whether or not to call the proceed method within the around advice (using the proceed extended annotation). The aim in designing Pipa based on JML is to reuse the existing JML-based tools. In order to make this possible the authors developed a tool (compiler) to automatically transform an AspectJ program with Pipa specifications into a standard Java program with JML specifications. To this end, the authors modified the AspectJ compiler (ajc) to retain the comments during the weaving process. After the weaving process, all JML-based tools can be applied to AspectJ programs. Therefore, the main goal of Pipa is to facilitate the use of JML language to verify AspectJ programs. On the other hand, we use AspectJ to implement JML features and verify Java programs.

6. CONCLUDING REMARKS

In this paper we discussed the benefits to use AOP to instrument JML features. We also discussed the analogy between JML and AspectJ that justify the use of AspectJ to instrument JML features (treated as crosscutting concern). In this way the issues covered throughout the paper provide means to answer the research questions pointed out in Section 1.

Another major contribution of this paper is that, unlike jmlc, our compiler (ajmlc) generates instrumented bytecode to verify constrained methods within static blocks during the class initialization. We also present three examples of Java programs annotated with JML to investigate the overhead in code size produced by two different AspectJ weavers. Such results provide an evidence that our approach generates smaller code than the original JML compiler when using the abc AspectJ weaver. Moreover, such results showed that in relation to application Jar sizes, ajmlc with either ajc and abc produces a smaller application size than jmlc. These results are essential when considering Java ME applications. We also presented some guidelines useful to choose properly which AspectJ weaver to employ.

We believe that the usage of aspects to implement a JML compiler introduces a new level of modularity. In other words, our approach is not invasive (the Java source code is not tangled and scattered with the generated assertion methods to check JML features during runtime). This gives more flexibility to extend the compiler with other JML constructs and to optimize the current implementation (since our source code instrumentation is less complexity resulting in a smaller source code instrumentation). In addition, optimizations in the weaven process are automatically inherited by our compiler when using abc.

As a future work, we also plan to address a problem suggested by Cheon [3]: to support assertion checking in a concurrent environment (e.g., multi-threaded program). We also intend to conduct more experiments using weavers that implement optimization techniques for AspectJ, including the work by Cordeiro [8]. Such a work provide some optimizations in the AspectJ abc compiler, which can improve the instrumented code generated by the ajmlc. As another future work, we intend to perform quantitative studies to compare the instrumented code generated by the classical jmlc compiler and the ajmlc compiler. These quantitative studies will respect to important software engineering attributes [5], such as composability, coupling, cohesion, number of attributes and operations. Finally, in addition to quantitative studies and code size conducted in this paper, a performance comparison would also be an interesting future work to investigate, especially in the Java ME context.

Table 4: Jar size results

	jmlc (KB)	ajmlc (ajc) (KB)	ajmlc (abc) (KB)
hierarchy classes	33.6	18.7	10.7
IntMathOps	20.6	7.5	4.7
StackAsArray	25.2	11.7	6.6

7. ACKNOWLEDGMENTS

We would like to thank Professor Gary Leavens for his comments and stimulating discussions on earlier topics of this paper. We would also like to thank Professor Patrice Chalin, and Perry James for their several and helpful discussions about JML and its semantics. Special thanks to Fernando Calheiros, he shared a substantial amount of domain knowledge related to Java ME with AspectJ.

This work was partially supported by CAPES and FINEP, brazilian research agencies.

8. REFERENCES

[1] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM.

[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] Y. Cheon and G. T. Leavens. A contextual interpretation of undefinedness for runtime assertion checking. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 149–158, New York, NY, USA, 2005. ACM.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[6] 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.

[7] C. Constantinides and T. Skotiniotis. Reasoning about a Classification of Cross-cutting Concerns in Object-Oriented Systems. In *Second Workshop on Aspect-Oriented Software Development (Workshop Aspektorientierte Softwareentwicklung der GI-Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung)*, Bonn, Germany, February 21-22, 2002.

[8] E. Cordeiro et al. Optimized compilation of around advice for aspect oriented programs. *Journal of Universal Computer Science*, 13(6):753–766, 2007.

[9] U. Costa et al. Specification and Runtime Verification of Java Card Programs. In *Brazilian Symposium on Formal Methods (SBMF)*, Oct. 2008.

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

[11] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. pages 21–35. Addison-Wesley, 2000.

[12] 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.

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

[14] G. Kiczales et al. Getting Started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[16] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[17] 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.

[18] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[19] G. T. Leavens et al. Jml reference manual. Department of Computer Science, Iowa State University. Available from url http://www.jmlspecs.org, Apr. 2007.

[20] M. Marin et al. A classification of crosscutting concerns. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 673–676, Washington, DC, USA, 2005. IEEE Computer Society.

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

[22] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

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

[24] H. Rebêlo et al. Implementing Java Modeling Language Contracts with AspectJ. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 228–233, New York, NY, USA, 2008. ACM.

[25] F. Rioux and P. Chalin. Effective and Efficient Runtime Assertion Checking for JML Through Strong Validity. In *Proceedings of the 9th Workshop on Formal Techniques for Java-like Programs (FTfJP'07)*, 2007.

[26] 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.