AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts

Henrique Rebêlo[λ], Gary T. Leavens[θ], Mehdi Bagherzadeh[β], Hridesh Rajan[β], Ricardo Lima[λ], Daniel Zimmerman[δ], Márcio Cornélio[λ], and Thomas Thüm[γ]

[λ]Universidade Federal de Pernambuco, PE, Brazil
{hemr, rmfl, mlc}@cin.ufpe.br
[θ]University of Central Florida, Orlando, FL, USA
leavens@eecs.ucf.edu
[β]Iowa State University, Ames, IA, USA
{mbagherz, hridesh}@iastate.edu
[δ]University of Washington Tacoma, USA
dmz@acm.org
[γ]University of Magdeburg, Germany
thomas.thuem@ovgu.de

Abstract

Aspect-oriented programming (AOP) is a popular technique for modularizing crosscutting concerns. In this context, researchers found that the realization of the design by contract (DbC) is crosscutting and fares better when modularized by AOP. However, previous efforts aimed at supporting crosscutting contract modularly instead hindered it. For example, in AspectJ-style, to reason about the correctness of a method call may require a whole-program analysis to determine what advice applies and what that advice does in relation to DbC implementation and checking. Also, when contracts are separated from classes, a programmer may not know about them and break them inadvertantly. In this paper we solve these problems with AspectJML, a new language for specification of crosscutting contracts for Java code. We also show how *AspectJML* supports the main DbC principles of modular reasoning and contracts as documentation.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Programming by contract, Assertion Checkers; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Assertions, Invariant, Pre- and postconditions, Specification techniques

General Terms Design, Languages, Verification

Keywords Design by contract, crosscutting contracts, AOP, JML, AspectJ, AspectJML

1. Introduction

Design by Contract (DbC), originally conceived by Meyer [30], is a useful technique for developing a program using specifications. The key mechanism in DbC is the use of the so-called "contracts". Writing out these contracts in the form of specifications and verifying them against the actual code either at runtime or compile time has a long tradition in the research community [7, 11, 13, 23, 25, 44, 50]. The idea of checking contracts at runtime was popularized by Eiffel [31] in the late 80's. In addition to Eiffel, there are other design by contract languages, such as the Java Modeling Language (JML) [25], Spec# [4], and Code Contracts [13].

It is claimed in the literature [6, 14, 20, 27–29, 40, 41, 45] that the contracts of a system are de-facto a crosscutting concern and fare better when modularized with aspect-oriented programming [21] (AOP) mechanisms such as pointcuts and advice [20]. The idea has also been patented [28]. However, Balzer, Eugster, and Meyer's study [3] contradicts this intuition by concluding that the use of aspects hinders design by contract specification and fails to achieve the main DbC principles such as documentation and modular reasoning. Also, they go further and say that "*no module in a system (e.g., class or aspect) can be oblivious of the presence of contracts*" [3, Section 6.3]. According to them, contracts should appear in the modules themselves and separating such contracts as aspects contradicts this view [32].

However, plain DbC languages like Eiffel [31], JML [25] also have problems when dealing with crosscutting contracts. Although a few mechanisms, such as invariant declarations help avoid scattering of specifications, the basic pre- and postcondition specification mechanisms do not prevent scattering of crosscutting contracts. For example, there is no way in Eiffel or JML to write a single pre- and postcondition and apply it to several of methods of a particular type. Instead such a pre- or postcondition must be repeated and scattered among several methods.

To cope with these problems this paper proposes AspectJML, a simple and practical aspect-oriented extension to JML. It supports the specification of crosscutting contracts for Java code in a modular way while keeping the benefits of a DbC language, like documentation and modular reasoning.

In the rest of this paper we discuss these problems and our AspectJML solution in detail. We also provide a real case study to show the effectiveness of our approach when dealing with crosscutting contracts.

JML Contracts

```
1  class Package {
2   double width, height;
3   //@ invariant this.width > 0 && this.height > 0;
4   double weight;
5   //@ invariant this.weight > 0;
6
7   //@ requires width > 0 && height > 0;
8   //@ requires width * height <= 400; // max dimension
9   //@ ensures this.width == width;
10  //@ ensures this.height == height;
11  //@ signals_only \nothing;
12  void setSize(double width, double height){
13   this.width = width;
14   this.height = height;
15  }
16
17  //@ requires width > 0 && height > 0;
18  //@ requires width * height <= 400; // max dimension
19  //@ requires this.width != width;
20  //@ requires this.height != height;
21  //@ ensures this.width == width;
22  //@ ensures this.height == height;
23  //@ signals_only \nothing;
24  void reSize(double width, double height){
25   this.width = width;
26   this.height = height;
27  }
28
29  //@ requires width > 0 && height > 0;
30  //@ requires width * height <= 400; // max dimension
31  //@ signals_only \nothing;
32  boolean containsSize(double width, double height){
33   if(this.width == width && this.height == height){
34    return true;
35   }
36   else return false;
37  }
38
39  //@ signals_only \nothing;
40  double getSize(){
41   return this.width * this.height;
42  }
43
44  //@ ...
45  //@ signals_only \nothing;
46  void setWeight(double weight) {
47   this.weight = weight;
48  }
49 ... // other methods
50 }
51
52 class GiftPackage extends Package {
53  //@ ...
54  //@ signals_only \nothing;
55  void setWeight(double weight) {
56   ...
57  }
58  ... // other methods
59 }
60
61 class Courier {
62  //@ ...
63  void deliver(Package p, String destination) {
64   ...
65  }
66 }
```

AspectJ Contracts

```
67 privileged aspect PackageContracts {
68  pointcut instMeth():
69   execution(!static * Package+.*(..));
70
71  pointcut sizeMeths(double w, double h):
72   execution(void Package.*Size(double, double))
73    && args(w, h);
74
75  pointcut setOrReSize(double w, double h):
76   execution(void Package.setSize(double, double))
77    || execution(void Package.reSize(double, double))
78    && args(w, h);
79
80  pointcut reSizeMeth(double w, double h):
81   execution(void Package.setSize(double, double))
82    && args(w, h);
83
84  pointcut allMeth(): execution(* Package+.*(..));
85
86  before(Package obj): instMeth() && this(obj) {
87   boolean pred = obj.width > 0 && obj.height > 0
88    && obj.weight > 0;
89   Checker.checkInvariant(pred);
90  }
91
92  before(double w, double h): sizeMeths(w, h){
93   boolean pred = w > 0 && h > 0
94    && w * h <= 400; // max dimension
95   Checker.checkPrecondition(pred);
96  }
97
98  before(Package obj, double w, double h):
99   reSizeMeth(w, h) && this(obj){
100   boolean pred = obj.width != w && obj.height != h;
101   Checker.checkPrecondition(pred);
102 }
103
104 after(Package obj, double w, double h) returning():
105   setOrReSize(w, h) && this(obj){
106    boolean pre = obj.width == w
107     && obj.height == h;
108    Checker.checkNormalPostcondition(pred)
109 }
110
111 after() throwing(Exception ex): allMeth() {
112  boolean pred = false;
113  Checker.checkExceptionalPostcondition(pred);
114 }
115
116 after(Point obj): instInv() && this(obj) {
117  boolean pred = obj.width > 0 && obj.height > 0
118   && obj.weight > 0;
119  Checker.checkInvariant(pred);
120 }
121 // other advice for checking contracts
122}
123
124aspect GiftPackageContracts {...}
125
126aspect CourierContracts {...}
127
128aspect Tracing {
129 after() returning(): execution(* Package.*(..)) {
130  System.out.println("Exiting"+thisJoinPoint);
131 }
132}
```

Figure 1. The JML and AspectJ contract implementations of the delivery service system [33].

2. The Problems and Their Importance

In this section we discuss the existing problems in modularizing crosscutting contracts in practice. The first two problems are AOP/AspectJ [20, 21] based, and the last, but not least, problem is related to a design by contract language like JML [25].

2.1 A Running Example

As a running example, Figure 1 illustrates a simple delivery service system [33], which manages package delivery. It uses contracts expressed in JML [25] (lines 1-66) and AspectJ [20] (lines 67-

126). In addition, we also include a tracing crosscutting concern modularized with AspectJ (lines 128-132).

In JML specifications, preconditions are defined by the keyword **`requires`** and postconditions by **`ensures`**. JML's (**`signals_only \nothing`**) specification denotes an exceptional postcondition which says that no exception (e.g., runtime exceptions included) can be thrown. For example, all methods declared in class `Package` are not allowed to throw exceptions. The invariants defined in the `Package` class restricts package's dimension and weight to be always greater than zero.

JML's counterpart in AspectJ is shown on lines 67-126. The main motivation in applying an AspectJ-like language is that we can explore some modularization opportunities that, otherwise, are not possible in a DbC language like JML. For instance, in the `PackageContracts` aspect, the second **`before`** advice declared (lines 92-96) checks, the common preconditions, which is scattered in the JML side, for all the methods with the name ending with `Size` and also take two arguments with **`double`** type. Similarly, the **`after-returning`** advice (lines 104-109) checks the common postconditions for both `setSize` and `reSize` methods. This advice only enforce the constraints after normal termination. In JML, the postconditions are called normal postconditions since they must only hold when a method returns normally [25]. A third example is illustrate in with the **`after-throwing`** advice (lines 111-114). It forbids any method in `Package` or subtypes to throw any exception. This is illustrated in the JML counterpart with the scattered specification (**`signals_only \nothing`**). This Second kind of postcondition in JML is called an exceptional postcondition [25].

2.2 The Modular Reasoning Problem

If we consider plain JML/Java without AspectJ, the example in Figure 1 supports modular reasoning [24, 26, 32, 39]. For example, suppose one wants to write code that manipulates objects of type `Package`. One could reason about `Package` objects using just that type's contract specifications (lines 1-50) in addition to the ones inherited from any supertypes [12, 24, 26].

Now let us consider the Java and AspectJ implementation of the delivery service system (without the JML specifications).

As observed, in addition to the classes in the base/Java code, Figure 1 defines four aspects. Three for contract checking and one for tracing. In plain AspectJ, the advice declarations are applied by the compiler without explicit reference to the aspect from a module or a client module; so by definition, modular reasoning about, for example, the `Package` module does not consider the advice declared by these four aspects. Hence, the aspect behavior is only available via non-modular reasoning. That is, in AspectJ, a programmer must consider every aspect that refers to `Package` class in order to reason about the `Package` module. So the answer to the question "What advice/contract applies to the method `setSize` in `Package`?" cannot (in general) be answered modularly. Therefore, a programmer cannot study the system one module at a time [2, 3, 19, 35, 39, 49].

2.3 Lack of Documentation Problem

In a design by contract language like Eiffel [31], JML [25], Spec# [4] or Code Contracts [13], the pre- and postconditions or invariant declarations are typically placed directly in or next to the code they are specifying. Hence, contracts increase system documentation [3, 32, 36]. In AspectJ, however, the advising code (which checks contracts) is separated from the code they advise and this forces programmers to consider all aspects in order to understand the correctness of a particular method. In addition, the physical separation of contracts can be harmful in the sense that an oblivious programmer can violate a method's pre- or postconditions when these are only recorded in aspects [3, 32, 36].

Consider now the tracing concern (Figure 1), modularized by the aspect `Tracing`. It prints a message after the succeed execution of any method in `Package` class when called. For this concern, different orders of composition with other aspects (that check contracts) lead to different behaviors/outputs. As a consequence, the **`after-returning`** advice (line 129) could violate `Package`'s invariants and pass undetected if the advice runs after those advice (in the `PackageContracts` aspect) responsible for checking the `Package`'s invariant. So, without either documentation or the use of AspectJ's **`declare precedence`** [20], to enforce a specific order on aspects, make the understanding of which order a specific pre- or postcondition would be executed quite difficult or not determined until they are executed [20].

Another problem by the lack of documentation implied by separating contracts as aspects is discussed by Balzer, Eugster, Meyer's work [3]. They argue that as programmers become aware of contracts only when using special tools, like AJDT [22], they are more likely to forget adapting the contracts when changing the classes.

2.4 Lack of Support for Crosscutting Contract Specification in DbC Languages

Balzer, Eugster, and Meyer's study [3] helped crystallize our thinking about the goals of a DbC language, in particular about the portion of such languages that provides good documentation, modular reasoning, and non-contract-obliviousness. Therefore, if we want to avoid the previous two problems discussed above, we can use a plain DbC language like JML [25].

Furthermore, let us explain two points about the JML specifications in Figure 1. The first is that a DbC language like JML can be used to modularize some contracts. For example, the invariant clauses (declared in `Package`) can be viewed as a form of built-in modularization. That is, instead of writing the same pre- and postconditions for all methods in a class, we just declare a single invariant that modularizes those pre- and postconditions. Second, specification inheritance is another form of modularization. In JML, an overriding method inherits method contracts and invariants from the methods it overrides[1].

However DbC languages (like JML) do not capture all forms of crosscutting contract structure [18, 20] that can arise in the specifications. As examples of such cases, consider the JML specifications illustrated on lines 1-66 in Figure 1. In this example, there are three ways in which crosscutting contracts that are not properly modularized with plain JML constructs:

(1) The preconditions that constrains the input parameters on the methods `setSize`, `reSize`, and `containsSize` (in `Package`) to be greater than zero and less than or equal to 400 (the package dimension). The main issue is that we cannot write them only once and apply to these or others methods that can have the same design constraint,

(2) The two normal postconditions of the methods `setSize` and `reSize` of `Package` are the same. They ensure that the both `width` and `height` fields are equal to the corresponding method parameters. However, one cannot write just a simple and local quantified form of these postconditions and apply them to the constrained methods, and

(3) The exceptional postcondition clause (**`signals_only \nothing`**) has to be explicitly written for all the methods that forbid exceptions to be thrown. This is the case of the declared methods in `Package` and `GiftPackage` classes. There is no way to modularize such a JML contract in one single place and apply to all constrained methods.

[1] Even though inheritance is not exactly a crosscutting structure [18, 20], a DbC language avoids repeating contracts for overriding methods.

2.5 The Dilemma

As observed, the main problem here is a trade-off. If we decide to use AspectJ to modularize such crosscutting contracts, the result would be a poor contract documentation and a compromised modular reasoning of such contracts. If we decide to go back to a design by contract language, such as JML, we would face the scattered nature of common contracts, as explained above. This dilemma leads us to the following research question: Is it possible to have the best of both worlds? That is, can we achieve good documentation and modular reasoning, and also specify crosscutting contracts in a modular way?

In the following, we discuss how our AspectJML DbC language provides constructs to specify crosscutting contracts in a modular and convenient way and overcomes the above problems.

3. The AspectJML Language

AspectJML extends JML [25] with support to handle crosscutting contract concern [29]. It allows programmers to define additional constructs (in addition to those of JML) to modularly specify pre- and postconditions and check them at certain well-defined points in the execution of a program. We call this *crosscutting contract specification* mechanism, or XCS for short.

XCS in AspectJML is based on a subset of AspectJ's constructs [20] that we include in JML. However, since JML is a design by contract language tailored for plain Java, we would need special support to use the traditional AspectJ's syntax. To simplify the adoption of AspectJML, the AspectJ constructs we include, to handle crosscutting contracts, are based on the alternative @AspectJ syntax [5].

The @AspectJ (often pronounced as "at AspectJ") syntax was conceived due to the merge of the standard AspectJ with AspectWerkz [5]. This merge enables crosscutting concern implementation by using constructs based on metadata annotation facility of Java 5. The main advantage of this syntactic style is that one can compile a program using a plain Java compiler. This implies that the modularized code using AspectJ works better with conventional Java IDEs or other tools that do not understand the traditional AspectJ syntax. In particular, this applies to the so-called "common" JML compiler on which ajmlc is based [8, 42, 43].

Figure 2 illustrates the @AspectJ version of the tracing crosscutting concern previously implemented with the traditional syntax (see Figure 1). Instead of using the **`aspect`** keyword, we use a class annotated with an **`@Aspect`** annotation. This tells AspectJ/ajc compiler to treat the class as an aspect declaration. Similarly, the **`@Pointcut`** annotation marks the empty method `trace` as a pointcut declaration. The expression specified in this pointcut is the same as the one used in the standard AspectJ syntax. The name of the method serves as the pointcut name. Finally, the **`@AfterReturning`** annotation marks the method `afterReturningAdvice` as an **`after returning`** advice. The body of the method is used to modularize the crosscutting concern (the advising code). This code is executed after the matched join point's execution returns without throwing an exception.

In the rest of this section, we present the main elements of the crosscutting contract specification support in our language. The presentation is informal and running-example-based.

3.1 XCS with Pointcut-Specifications

This is the simplest way to modularize crosscutting contracts at source code level. Recall that a *pointcut designator* enables one to select well-defined points in a program's execution, which are known as *join points* [20]. Optionally, a pointcut can also include some of the values in the execution context of intercepted join points. In AspectJML, we can compose these AspectJ pointcuts combined with JML specifications.

```
@Aspect()
class Tracing {
 @Pointcut("execution(* Package.*(..))")
 public void trace() {}

 @AfterReturning("trace()")
 public void afterReturingAdvice(JoinPoint jp) {
  System.out.println("Exiting"+jp);
 }
}
```

Figure 2. The tracing crosscutting concern implementation of Figure 1 using @AspectJ syntax.

The major difference, in relation to plain AspectJ, is that a specified pointcut is always processed when using the AspectJML compiler (ajjmlc). So, in standard AspectJ, a single pointcut declaration does not contribute to the execution flow of a program unless we define some AspectJ advice that uses such a pointcut. In AspectJML, fortunately, we do not need to define an advice to check a specification in a crosscutting fashion. Although it is possible to use advice declarations in AspectJML (as we discuss in subsection 3.2), we do not require them. This makes AspectJML simpler and a programmer only needs to know AspectJ's pointcut language in addition to the main JML features.

Specifying crosscutting preconditions

Recall our first crosscutting contract scenario described in Subsection 2.4. It consists of two preconditions for any method, in `Package` class (Figure 1), with name ending with `Size` that returns **`void`** and takes a double argument of **`double`** type. For this scenario, consider the JML annotated pointcut with the following preconditions:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
@Pointcut("execution(* Package.*Size(double, double))"+
 "&& args(width, height)")
void sizeMeths(double width, double height) {}
```

The pointcut `sizeMeths` matches all the executions of size-like methods of class `Package`. As observed, this pointcut is exposing the intercepted method arguments of type **`double`**. This is done in @AspectJ by listing the formal parameters in the pointcut method. Then we bind the parameter names in the pointcut's expression (within the annotation **`@Pointcut`**) using the argument-based pointcut **`args`** [20].

The main difference between this pointcut declaration and standard pointcut declarations in @AspectJ is that we are adding two JML specifications (using the **`requires`** clause). In this example the JML says to check the declared preconditions before the executions of intercepted methods.

We recommend that the above JML-annotated pointcut is declared in the class `Package`. This guideline will ensure that we keep the modular reasoning and documentation benefits [3] when reasoning about any method or type's specifications. (However, because AspectJML uses AspectJ's pointcut declarations, a programmer could place such pointcuts in any class, if that helped to better modularize other crosscutting specification concerns.)

Specifying crosscutting postconditions

We discuss now how to properly modularize crosscutting postconditions in AspectJML. In JML, there are two kinds of postconditions: normal and exceptional postconditions. Normal postconditions constrain methods that return without throwing an exception.

To illustrate AspectJML's design, we discuss scenarios (2) and (3) from Subsection 2.4.

For scenario (2), we use the following specified pointcut:

```
//@ ensures this.width == width;
//@ ensures this.height == height;
@Pointcut("(execution(* Package.setSize(double, double))"+
 "|| execution(* Package.reSize(double, double)))"+
 "&& args(width, height)")
void setOrReSize(double width, double height) {}
```

The above constrains the executions of `setSize` and `reSize` methods in `Package` to ensure that after their executions, the fields `width` and `height` have their values equal to the ones passed as arguments.

To modularize the crosscutting postcondition of scenario (3), we use the following JML annotated pointcut declaration.

```
//@ signals_only \nothing;
@Pointcut("execution(* Package+.*(..))")
void allMeth() {}
```

The above specification forbids the executions of any method in `Package` (or a subtype, such as `GiftPackage`) to throw an exception. If any intercepted method ends up by throwing an exception (even a runtime exception), a JML exceptional postcondition error is thrown to signal the contract violation. In this pointcut, we do not expose any intercepted method's context.

Multiple specifications per pointcut

All the crosscutting contract specifications discussed above consist of only one kind of JML specification per pointcut declaration. However, AspectJML can include more than one kind of JML specifications in a pointcut declaration. As an example, let us assume that the `Package` type in Figure 1 does not have the `containsSize` method along with its JML specifications. In this scenario, we can write a single pointcut to modularize the recurrent pre- and postconditions of methods `setSize` and `reSize` of `Package` type. Therefore, instead of having separate JML annotated pointcuts for each crosscutting contract, we specify them in a new version of the pointcut `sizeMeths`:

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
//@ ensures this.width == width;
//@ ensures this.height == height;
@Pointcut("execution(* Package.*Size(double, double))"+
 "&& args(width, height)")
void sizeMeths(double width, double height) {}
```

This pointcut declaration modularly specifies both preconditions and normal postconditions of the same intercepted size methods (`setSize` and `reSize`) of `Package`.

Specification of unrelated types

Another issue to consider is whether or not AspectJML can modularize inter-type[2] crosscutting specifications. All the crosscutting contract specifications we discuss are related to one type (intra-type) or its subtypes. However, AspectJ can advise methods of different (unrelated) types in a system. This quantification property of AspectJ is quite useful [51] but can also be problematic from the point of view of modular reasoning, since one needs to consider all the aspect declarations to understand the overall system behavior [2, 19, 39, 47–49]. Instead of ruling this out, the design of ApsectJML allows the specifier to use specifications that constrain unrelated inter-types. This puts the decision about when to use such features in the hands of the AspectJML user.

[2] When we refer to inter-types here is not that AspectJ feature [20] to add methods or fields with static crosscutting mechanism. Instead we are referring to unrelated modules in a system. That is, types that are not related to each other, but can present a common crosscutting contract structure.

As an example, recall our running example in Figure 1. We know that all the methods declared in `Package` and its subtype `GiftPackage` are forbidden to throw exceptions (see the **`signals_only`** specification). Suppose now that the method `deliver` in type `Courier` also has this constraint. Note that the type `Courier` is not a subtype of `Package`. They are independent to some extent. In other words, they are only related in the sense that the method `deliver` depends on `Package` type due to the declaration as a formal parameter. This way, let us also consider that `Courier` has more methods that is not dependent of `Package` at all. So, consider the following type declaration:

```
interface CommonSignalsOnly {
 class CommonSignalsOnlyXCS {
  //@ signals_only \nothing;
  @Pointcut("execution(* CommonSignalsOnly+.*(..))")
  void allMeth() {}
 }
}
```

This type declaration illustrates how we specify crosscutting contracts for interfaces. As we know, pointcuts are not allowed to be declared within interfaces. We overcome this problem by adding an inner class that represents the crosscutting contracts of the outer interface declaration. As a part of our strategy, the pointcut declared in the inner class only refers to the outer interface (see the reference in the pointcut predicate expression). Now, any type that wants to forbid its method declarations to throw exceptions only need to implement the interface `CommonSignalsOnly` in our case. Such an interface acts like a marker interface idiom [17]. This is important to avoid obliviousness and maintain modular reasoning (according to our definition).

Collected XCS examples

All the crosscutting contract specifications used so far in this section (discussed as scenarios in Subsection 2.4) with pointcuts-specifications are illustrated in Figure 3 (the shadowed part illustrates the XCS in AspectJML's pointcuts and specifications).

3.2 XCS with Pointcut-Advice-Specifications

A second way to specify crosscutting contracts, at the source code level is to use aspects and advice declarations in addition to pointcuts and JML specifications.

In order to exemplify the use of pointcut-advice-specifications, recall scenario (1) from section 2.4, and consider the modified version of the `Package` class in Figure 4. We observe an important difference, in the `Package` class, to the previous examples. We can see an inner aspect named `PackageAspect` with a **`pointcut`** and **`before`** advice declarations. The reason to use an inner aspect is because we cannot declare AspectJ advice inside classes. Another observation is that we moved the preconditions to the **`before`** advice. The semantics of precondition checking in AspectJML still remains the same. So, before the executions of the intercepted join points by the pointcut `sizeMeths`, we have the preconditions checked. The main difference is that we have another behavior that will be executed just before the join point's executions. This is illustrated by the **`before`** advice that performs a trace implementation for the intercepted join points.

Therefore, the main advantage of doing this strategy, shown in Figure 4, is that besides checking the specifications in a crosscutting fashion, we can also define another crosscutting implementation for the same constrained methods.

One can argue that, based on the given AspectJML specification in Figure 4, would be more sensible if we move the specifications of the **`before`** advice back to the pointcut definition. Although this makes sense, we are just showing how to provide the same effect using the JML specification attached to an advice declaration. A scenario, however, that this would make more sense is

```
1  class Package {
2   double width, height;
3   //@ invariant this.width > 0 && this.height > 0;
4   double weight;
5   //@ invariant this.weight > 0;
6
7   //@ requires width > 0 && height > 0;
8   //@ requires width * height <= 400; // max dimension
9   @Pointcut("execution(* Package.*Size(double,double))"+
10   "&& args(width, height)")
11  void sizeMeths(double width, double height) {}
12
13  //@ ensures this.width == width;
14  //@ ensures this.height == height;
15  @Pointcut("(execution(* Package.setSize(double,double))"
16   + "|| execution(* Package.reSize(double, double)))"+
17   "&& args(width, height)")
18  void setOrReSize(double width, double height) {}
19
20  //@ signals_only \nothing;
21  @Pointcut("execution(* Package+.*(..))")
22  void allMeth() {}
24
25  void setSize(double width, double height){...}
26
27  //@ requires this.width != width;
28  //@ requires this.height != height;
29  void reSize(double width, double height){...}
30
31  boolean containsSize(double width, double height){...}
32  double getSize(){...}
33
34  //@ ...
35  void setWeight(double weight) {...}
36 ... // other methods
37 }
38 class GiftPackage extends Package {
39  //@ ...
40  void setWeight(double weight) {...}
41  ... // other methods
42 }
```

Figure 3. The crosscutting contract specifications used so far for the delivery service system [33] with AspectJML.

```
class Package {
 @Aspect()
 static class PackageAspect {
  @Pointcut("execution(* Package.*Size(double,double))"+
   "&& args(width, height)")
  void sizeMeths(double width, double height) {}

  //@ requires width > 0 && height > 0;
  //@ requires width * height <= 400; // max dimension
  @Before("sizeMeths(width, height)")
  public void beforeAdvice(JoinPoint jp, double width,
   double height) {
    System.out.println("Entering: "+jp);
  }
 }
 // ... other specified methods
}
```

Figure 4. A crosscutting precondition specification using pointcuts-advice-specifications.

shown in Figure 5. Since this **`before`** advice uses an anonymous pointcut [20], the only way to constrain the join points with specifications is by adding them directly to the advice declaration.

It is important to stress that AspectJML does not check such preconditions within the given **`before`** advice. In addition, the reader should not be confused to think that the above preconditions are for the given advice. Our approach is for specifying crosscutting contracts to not specify AspectJ advice directly. Thus all contract

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
@Before("execution(* Package.*Size(double, double))"+
 "&& args(width, height)")
public void beforeAdvice(JoinPoint jp, double width,
 double height) {
  System.out.println("Entering: "+jp);
}
```

Figure 5. Specifications added to advice with an anonymous pointcut.

```
// written by Cathy
public class ClientClass {
 public void clientMeth(Package p)
  { p.setSize(0, 1); }
}
```

Figure 6. Client code, written by "Cathy."

specifications are for the base code that is advised. Specifying and checking AspectJ advice is an interesting future work.

3.3 AspectJML Expressiveness

So far we just used the **`execution`** and **`within`** pointcut designators to select join points. This is to be in conformance with the supplier-side checking adopted by most DbC/runtime assertion checkers (RAC). That is, such RAC compilers operate by injecting code to check each method's precondition at the beginning of its code, and injecting code to check the method's postcondition at the end of its code. This checking code is then run from within the method's body at the supplier side.

AspectJML also includes other primitive pointcut designators that identify join points in different ways [20]. For instance, we can use the **`call`** pointcut. This would provide runtime checking at call site. Code Contracts [13] is an example of a DbC language that provides runtime checking at client side. But it supports only precondition checking. Since JML also supports client-side checking [38], the **`call`** pointcut enables client-side checking for AspectJML in relation to specified crosscutting contracts.

```
//@ requires width > 0 && height > 0;
//@ requires width * height <= 400; // max dimension
@Pointcut("(execution(* Package.*Size(double, double))"+
           "|| call(void Package.*Size(double, double)))"
         "&& args(width, height)")
void sizeMeths(double width, double height) {}
```

This is an example of a crosscutting precondition specification, in AspectJML, that takes into account both **`execution`** and **`call`** pointcut designators.

AspectJML also supports AspectJ's control-flow based pointcuts (e.g., **`cflow`**) [20].

3.4 AspectJML's Benefits

As mentioned, design by contract is a recurrent concern and several authors claim that it could be better modularized and handled by means of aspect-oriented mechanisms like those we find in AspectJ [6, 14, 20, 27–29, 40, 41, 45]. After that, Balzer, Eugster, and Meyer's study [3] raised important issues that argue against the aspectization of contracts. Issues like documentation and modular reasoning are compromised when using an AspectJ-like language. Indeed, AOP/AspectJ themselves have been focus of a grand debate including modularity and modular reasoning [2, 19, 39, 47–49].

Enabling modular reasoning

Recall that our notion of modular reasoning means that one can soundly verify a piece of code in a given module, such as a class,

using only the module's own specifications, its own implementation, and the interface specifications of modules that its own implementation references [12, 24, 26, 32, 39].

With respect to whether or not AspectJML supports modular reasoning, like a DbC language such as JML, consider the client code, that we will imagine is written by Cathy, as shown in Figure 6. To verify the call to `setSize`, Cathy must determine what specifications to use. If she uses the definition of modular reasoning, she must use the specifications for `setSize` in `Package`. Let us assume that she uses the JML specifications of Figure 1. Hence, she uses:

(1) The pre- and postconditions located at the method `setSize` (lines 7-11),

(2) The first invariant definition on line 3, that constrains the `Package` dimension (`width` and `height`) fields, and

(3) The second invariant (line 5) related to the `Package`'s weight.

Cathy only needs these three specifications when using plain JML. (`Package` has no supertype; otherwise, she would also need to consider specifications inherited from such supertypes.) After obtaining these specifications, she can see that there is a precondition violation regarding the width value of `0` passed to `setSize` (in Figure 6).

Suppose now Cathy wants to perform again the same modular reasoning task, but using the AspectJML specifications in Figure 3 instead of the JML ones of Figure 1. In this case she needs to find the following pieces of checking code:

(1) The first invariant definition on line 3, that constrains the `Package` dimension (`width` and `height`) fields,

(2) The second invariant (line 5) related to the `Package`'s weight,

(3) The preconditions of the pointcut (lines 7-8) `sizeMeths`, since it intercepts the execution of method `setSize`,

(4) Similarly the normal postconditions (lines 13-14) located at the pointcut `setOrReSize`, and

(5) The exceptional postcondition (line 20) of pointcut `allMeth`.

As before, this only involves modular reasoning, and she can still detect the potential precondition violation related to `Package`'s width. In this case Cathy, needed 7 AspectJML specifications to reason about the correctness the call to `setSize`. However, in contrast to modular reasoning scenario 1, she needed two more specifications to complete the reasoning task using AspectJML. So, although AspectJML supports modular reasoning, Cathy must follow a slightly more indirect process to reason about the correctness of a call. This confirms that the obliviousness issue present in AspectJ-like languages [15] does not occur in this example. Cathy is completely aware of the contracts of `Package` class.

Enabling documentation

Regarding documentation, this example of Cathy's reasoning shows that despite the added indirection, reasoning with AspectJML specifications does not necessarily have a modularity difference compared to reasoning with JML specifications. Only the location where these specifications can appear can be different, due to the use of pointcut declarations in AspectJML.

Our conclusion is that an inherent cost of crosscutting contract modularization and reuse is the cost of some indirection in finding specifications, which is necessary to avoid scattering (repeated specifications). But using AspectJML, users also have several new possibilities for crosscutting contracts.

```
/** Generated by AspectJML to check the precondition of
  * method(s) intercepted by sizeMeths pointcut. */
before (Package object$rac, final double width,
  final double height) :
  (execution(* p.Package.*Size(double,double))
  && this(object$rac) && args(width, height)) {
  boolean rac$b = (((width > +0.0D) && (height > +0.0D))
   && ((width * height) <= 400.0D));
  JMLChecker.checkPrecondition(rac$b, "errorMsg");
 }
```

Figure 7. Generated before advice to check the crosscutting preconditions of `Package` in Figure 3.

```
class Package {
    double width, height;
    // @ invariant this.width > 0 && this.height > 0;
    double weight;
    // @ invariant this.weight > 0;

    //@ requires width > 0 && height > 0;
    //@ requires width * height <= 400; // max dimension
    @Pointcut("execution(* Package.*Size(double,double))"+
            "&& args(width, height)")
    void sizeMeths(double width, double height) {}

    void setSize(double width, double height){
        this.width = width;
        this.height = height;
    }
    //@ requires this.width != width;
    //@ requires this.height != height;
advised by PackageCrossRef.around(double,double): delivery.Package.sizeMeths
        this.width = width;
        this.height = height;
    }
    // ... other methods
}
```

Figure 8. The crosscutting contract structure in the `Package` class using AspectJML/AJDT [22].

3.5 Runtime Assertion Checking

We implemented the AspectJML crosscutting contract specification technique in our JML/ajmlc compiler [42, 43] which is available online at: `http://www.cin.ufpe.br/~hemr/JMLAOP/ajmlc.htm`. This is the first RAC to support crosscutting contract specifications.

Compilation strategy

The ajmlc compiler itself was described in a previous work [43]. Unlike the classical JML compiler, jmlc [8, 10], it generates aspects to check specifications. It also has various code optimizations [42] and better error reporting. The main difference of the previous ajmlc to the new one is the support to AspectJML features like specified pointcuts. Instead of saying JML/ajmlc, we now say AspectJML/ajmlc.

Figure 7 shows the **`before`** advice generated by the ajmlc compiler to check the crosscutting preconditions of class `Package` defined in Figure 3.[3] The variable `rac$b` denotes the precondition to be checked. This variable is passed as an argument to `JMLChecker.checkPrecondition`, which checks such preconditions; if it is not true, then a precondition error is thrown.

Ordering of checks

As ajmlc generates AspectJ aspects to check contracts, it also enforces/declares aspect precedence. For instance, if we have advising code for other crosscutting concerns, it can only be allowed to

[3] The ajmlc compiler provides a compilation option that prints all the checking code as aspects instead of weaving them.

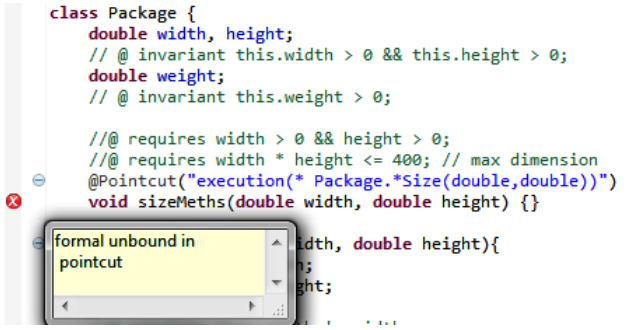

Figure 9. An example of a malformed pointcut declaration in AspectJML.

execute after the preconditions are satisfied; otherwise, a precondition violation is thrown.

The postconditions are only checked after all the advising code's execution. This order prevents undetected postcondition violations, which could happen if postconditions were checked before the execution of the advising code.

Taming obliviousness

In AspectJML specification, it is possible to use crosscutting contract specification mechanisms to write modular specifications. Since AspectJML uses AspectJ's pointcut declarations, one can argue that a programmer can specify several modules in one single place. Intuitively, this would affect several modules in the system. However, AspectJML rule out this possibility, if one tries to write such pointcuts, they will have not effect. This happens because AspectJML associates a pointcut with the type in which it was specified (see the generated code in Figure 7). Hence, only join points within the given type or its subtypes are allowed. The cross-references generated by AspectJML (see Subsection 3.6) can help visualize the intercepted types.

Even though there is no way in AspectJML to specify unrelated modules anonymously, the declared pointcuts can still be used within aspect types that can crosscut unrelated types. The main issue is that the JML specifications have no effect on anonymously intercepted modules.

Contract violation example in AspectJML

As an example of runtime checking using AspectJML/ajmlc, recall the client code illustrated in Figure 6. In this scenario, we got the following precondition error in the AspectJML RAC:

```
Exception in thread "main"
org.jmlspecs.ajmlrac.runtime.JMLEntryPreconditionError:
by method Package.setSize regarding code at
File "Package.java", line 13 (Package.java:13), when
  'width' is 0.0
  'height' is 1.0
  ...
```

As can be seen, in this error output, the shadowed input parameter `width` is displaying `0.0`. But the precondition requires a package's width to greater than zero. As a result, we get this precondition violation during runtime checking when calling such client code.

3.6 Tool Support

In aspect-oriented programming, development tools like Eclipse/AJDT [22], allow programmers to easily browse the crosscutting structure of their programs. In the same sense, for AspectJML, we are developing analogous support for browsing crosscutting contract structure. For this end, we use the already provided functionalities by the Eclipse/AJDT with minor adjustments.

For example, consider the crosscutting contract structure of the `Package` class using AspectJML/AJDT [22]. As observed, we can see the arrows indicating where the crosscutting contracts apply. In plain AspectJ/AJDT, this example show no crosscutting structure information. This is because we just have pointcut declarations without advice. In AspectJ, we need to associate the declared pointcuts to advice in order to be able to browse the crosscutting structure of a system. Hence, we have implemented an option in AspectJML that generates the cross-references information for crosscutting contracts when we have just pointcut declarations.

Figure 9 shows another example where the use of the AspectJ/AJDT helpers an AspectJML programmer to write a valid pointcut declaration. As depicted, the AspectJML programmer got an error from AJDT because he/she forgot to bind the formal parameters of the pointcut method declaration with the pointcut expression by using the argument-based pointcut **`args`**. The well-formed pointcut can be seen in Figure 8. All the AspectJ/AJDT IDE validation is inherited by AspectJML.

It is important to stress that the functionalities of AJDT we provide for AspectJML is just for aid the overall AspectJML approach. Putting in other words, we do not need any IDE support to reason about JML specifications in a modular way, as previously discussed. However, we argue that for beginner AspectJML programmers, a tool support like AJDT helpers them mastering the use of the pointcut language. Moreover, by using this tool support, one does not need to interpret the pointcut expression predicates to see whether or not it applies to some method. The tool gives us the complete list of all applicable pointcuts that should be inspected in relation to their JML specifications.

4. The HealthWatcher Case Study

Our evaluation of the XCS feature of AspectJML involves a medium-sized case study. The chosen system is a real health web-based complaint system, called Health Watcher (HW) [16, 46]. The main purpose of the HW system is to allow citizens to register complaints regarding health issues. This system was selected because it has a detailed requirements document available [16]. This requirements document describes 13 use cases and forms the basis for our JML specifications.

We analyzed the crosscutting contract structure of the HW system, comparing its specification in JML and AspectJML. Our results are available online at [37].

4.1 Understanding the Crosscutting Contract Structure

One of the most important steps in the evaluation is to recognize how the contract structure crosscuts the modules of the HW system. We now show some of these crosscutting contracts present in HW using the standard JML specifications.

Crosscutting preconditions

Crosscutting preconditions occur in the HW system's `IFacade` interface. This facade makes available all 13 use cases as methods. Consider the following code from this interface:

```
//@ requires code >= 0;
public IteratorDsk searchSpecialitiesByHealthUnit(int code);

//@ requires code >= 0;
public Complaint searchComplaint(int code);

//@ requires code >= 0;
public DiseaseType searchDiseaseType(int code);

//@ requires code >= 0;
public IteratorDsk searchHealthUnitsBySpeciality(int code);

//@ requires healthUnitCode >= 0;
public HealthUnit searchHealthUnit(int healthUnitCode);
```

These methods comprise all the search-based operations that HW makes available. The preconditions of these methods are identical, as each requires that the input parameter, the code to be searched, is at least zero. However, in plain JML one cannot write a single precondition for just these 5 search-based methods.

Crosscutting postconditions

Still considering the HW's facade interface `IFacade`, let us focus now on crosscutting postconditions. First, we analyze the crosscutting contract structure for normal postconditions:

```
//@ ensures \result != null;
public IteratorDsk searchSpecialitiesByHealthUnit(int code);

//@ ensures \result != null;
public IteratorDsk searchHealthUnitsBySpeciality(int code);

//@ ensures \result != null;
public IteratorDsk getSpecialityList()

//@ ensures \result != null;
public IteratorDsk getDiseaseTypeList()

//@ ensures \result != null;
public IteratorDsk getHealthUnitList()

//@ ensures \result != null;
public IteratorDsk getPartialHealthUnitList()

//@ ensures \result != null;
public IteratorDsk getComplaintList()
```

As observed, all the methods in `IFacade` that returns `IteratorDsk` should return a non-null object reference. In standard JML, there are other two ways to express this constraint [9]. The first one considers the non-null semantics for object references. In this case we do not need to write out such normal postconditions to handle non-null. However, we can deactivate this option in JML if there are more situations in the system that could be null. In this scenario, whenever we find a method that should return non-null, we still need to write these normal postconditions. So, by assuming that we are not using the non-null semantics of JML as default, these postconditions become redundant. The second is to use the JML type modifier **`non_null`**; however, even this would lead to some (smaller) amount of repeated postconditions.

In relation to exceptional postconditions of `IFacade` interface, we found an interesting crosscutting structure scenario. Consider the following code:

```
//@ signals_only java.rmi.RemoteException;
public void updateComplaint(Complaint q) throws
  java.rmi.RemoteException,...;

//@ signals_only java.rmi.RemoteException;
public IteratorDsk getDiseaseTypeList() throws
  java.rmi.RemoteException,...;

//@ signals_only java.rmi.RemoteException;
public IteratorDsk getHealthUnitList() throws
  java.rmi.RemoteException,...;

//@ signals_only java.rmi.RemoteException;
public int insertComplaint(Complaint complaint) throws
  java.rmi.RemoteException,...;

... // all facade methods contain this constraint
```

As can be seen, these `IFacade` methods can throw the Java RMI exception `RemoteException` (see the methods throws clause). This exception is used as a part of the Java RMI API used by HW system. Even though we list only four methods, all the methods contained in the `IFacade` interface contain this exception in their throws clause. Because of that, the **`signals_only`** clause shown needs to be repeated for all methods in `IFacade` interface. However, in JML one cannot write a single **`signals_only`** clause to constrain all such methods in this way.

Another example of exceptional postconditions is given by the search-based methods discussed previously. All these search-based methods should have a **`signals_only`** clause that allows the `ObjectNotFoundException` to be thrown. As with the `RemoteException`, one cannot write this specification once and apply to all search-based methods.

4.2 Modularizing Crosscutting Contracts in HW

To restructure/modularize the crosscutting contracts of the HW system, we use the XCS mechanisms of AspectJML. By doing this, we avoid repeated specifications, which shown an improvement over standard DbC mechanisms. In the following we show the details of how AspectJML achieves a better separation of the contract concern for this example.

Specifying crosscutting preconditions

In relation to the crosscutting preconditions of HW we discussed, we can proper modularize them with the following JML annotated pointcut in AspectJML:

```
//@ requires code >= 0;
@Pointcut("execution(* IFacade.search*(int))"+
 "&& args(code)")
void searchMeths(int code) {}
```

With this pointcut specification, we are able to locate the precondition for all the search-based methods. To select the search-based methods, we use a property-based pointcut [20], which matches join points by using wildcarding. Therefore, our pointcut matches any method starting with `search` and takes an **`int`** parameter type. So, before the executions of such intercepted methods, the precondition that constrains the code argument to be at least zero is enforced during runtime; if it does not hold, then one gets a precondition violation error.

Specifying crosscutting postconditions

Let us now consider the modularization of the two kinds of crosscutting postconditions we discussed. For normal postconditions, please consider the following code in AspectJML:

```
//@ ensures \result != null;
@Pointcut("execution(IteratorDsk IFacade.*(..))")
void nonNullReturnMeths() {}
```

With this pointcut specification, we are able to explicitly modularize the non-null constraint. The pointcut expression we use matches any method with any list of parameters but must return type `IteratorDsk`.

We now show the AspectJML code responsible for modularizing the exceptional postconditions we previously discussed. Consider the following JML annotated pointcuts expressed in AspectJML:

```
//@ signals_only java.rmi.RemoteException;
@Pointcut("execution(* IFacade.*(..))")
void remoteExceptionalMeths() {}

//@ signals_only ObjectNotFoundException;
@Pointcut("execution(* IFacade.search*(..))")
void objectNotFoundExceptionalMeths() {}
```

These two specified pointcuts in AspectJML are responsible for modularizing the exceptional postconditions for methods that are allowed to throw `RemoteException` and those that can throw `ObjectNotFoundException`, respectively. The first pointcut applies the specification for all methods in `IFacade`, whereas the second one intercepts just the search-based methods.

4.3 Reasoning About Change

As observed, the main benefit of AspectJML is to allow the modular specification of crosscutting contracts in an explicit and expressive way. The key mechanism is the quantification property inherited from AspectJ [20]. Besides the documentation and modularization of crosscutting contracts achieved by using AspectJML, another immediate benefit one can have by using our approach is during software maintenance.

For instance, if we add a new exception that can be thrown by all `IFacade` methods, instead of (re)writing a **`signals_only`** clause, we can add this exception to the **`signals_only`** list of the shown pointcut `remoteExceptionalMeths`. This pointcut can be reused whenever we want to apply constraints to methods already intercepted by the pointcut.

Another maintenance benefit is during system evolution. On one hand, we are adding more methods in the `IFacade` interface to comprise system's new use cases. On the other hand, for the new added methods we do not need to explicitly apply existing constraints to them. In other words, the modularized contracts that apply to all methods also automatically applied to the new added ones, with no cost. Finally, even if the crosscutting contracts are well documented by using JML specifications, the AJDT tool helps programmers to visualize the overall crosscutting contract structure. Just after a method is declared, we can see which crosscutting contracts are applying to it through the cross-references feature of AJDT [22].

5. Discussion

This section discusses some issues involving AspectJML specification language, such as limitation, compatibility, open issues, and related work.

5.1 A Limitation of AspectJML

Even though AspectJML gives us the benefit of modularity when handling crosscutting contracts, we still have some situations that AspectJML cannot currently deal with.

In order to exemplify the main drawback, consider the following JML/Java code:

```
//@ requires x > 0;
public void m(int x){}

//@ requires x > 0;
//@ requires y > 0;
public void n(int x, int y){}

//@ requires y > 0;
public void o(double x, int y, double z){}

//@ requires z > 0;
public void p(double y, int z){}
```

In this code, we can observe that all formal parameters involving the Java primitive **`int`** types should be greater than zero (see the preconditions). In JML, we cannot write this precondition only once and apply for all **`int`** arguments for the above methods. Unfortunately, this also cannot be done with AspectJML. The reason is that we cannot write a pointcut that matches all methods with **`int`** types in any position and associate a bound variable that can be used in the precondition. This is a limitation of AspectJ's pointcut mechanism, so there is fix the problem even with AspectJ.

5.2 AspectJML compatibility

One of the goals of this work is to support a substantial user community. To make this concrete, we have chosen to design crosscutting contract specification in AspectJML as a compatible extension to JML using AspectJ's pointcut language. This takes advantage of AspectJ's familiarity among programmers. Our goal is to make programming and specifying with AspectJML feel like a natural extension of programming and specifying with Java and JML. The AspectJML/ajmlc compiler has the following properties:

- all legal JML annotated Java programs are legal AspectJML programs,
- all legal AspectJ programs are legal AspectJML programs,
- all legal Java programs are legal AspectJML programs, and
- all legal AspectJML programs run on standard Java virtual machines.

5.3 JML Versus AspectJ

In this paper, we discussed the main problems of dealing with contracts expressed in both JML and AspectJ. Indeed, this comparison was suggested by Kiczales and Mezini [22]. They asked researchers to explore what issues are better specified as contract/behavioral specifications and what issues are better addressed directly in pointcuts. In this context, AspectJML goes beyond their question in the sense that it combines both pointcuts and contracts. We showed that DbC is better used with a design by contract language, but for situation involving scattering of contracts it can be advantageous to provide a form of specified pointcuts that allows crosscutting contract specifications.

5.4 Open Issues

Our evaluation of AspectJML is limited to two systems, the delivery service system [33] and the Health Watcher [46]. Although we know of no scaling issues, larger-scale validation is still needed to analyze more carefully the benefits and drawbacks of AspectJML. Library specification and runtime checking studies are another interesting area for future work.

Another open issue, which we intend to address in future versions of AspectJML, is related to the pointcut parameters and methods with common argument types (see Subsection 5.1).

Two more important open issues that could be explored in AspectJML are related to specification and modular reasoning of AspectJ programs [40]. These are interesting points since we can also program in AspectJ using AspectJML.

5.5 Other forms of Aspectized DbC

As discussed throughout the paper, there are several works in the literature that argue in favor of implementing DbC with AOP [14, 20, 28, 41]. Kiczales opened this research avenue by showing a simple precondition constraint implementation in one of his first papers on AOP [20]. After that, other authors explored how to implement and separate the DbC concern with AOP [14, 20, 28, 40, 41]. All these works offer common templates and guidelines for DbC aspectization.

However, Balzer, Eugster, and Meyer argued that DbC aspectization is more harmful than good [3], since one loses all the key properties of a DbC language: documentation, specification inheritance, and modular reasoning. Indeed, they argue that aspect interaction can make even worse the understanding of how contracts are checked, and in what order they are checked.

We go beyond these works by showing how to combine the best design features of a design by contract language like JML and the quantification benefits of AOP such as AspectJ. As a result we conceive the AspectJML specification language that is suitable for specifying crosscutting contracts. In AspectJML, one can specify crosscutting contracts in a modular way while preserving key DbC principles such as documentation and modular reasoning. Tradeoffs among different notions of modularity can be made by the AspectJML user.

The work of Bagherzadeh *et al.* [2] contains "translucid" contracts that are grey-box specifications of the behavior of advice. Although which advice applies is unspecified, the specification allows modular verification of programs with advice, since all advice must satisfy the specifications given. The grey-box parts of translucid contracts are able to precisely specify control effects, for example specifying that a particular method must be called a certain number of times, and under certain conditions, which is not possible with AspectJ or AspectJML. $Ptolemy_x$ [1] is an exception-aware extension to Ptolemy/translucid contracts [2]. As with AspectJML, Ptolemy$_x$ supports specification and modular reasoning about exceptional behaviors. The main difference is that AspectJML is used to specify and reason about Java code. On the other hand, Ptolemy$_x$ is used to specify and reason about event announcement and handling.

Pipa [52] is a design by contract language tailored for AspectJ. As with AspectJML, Pipa is an extension to JML. However, Pipa uses the same approach as JML to specify AspectJ programs, with just a few new constructs. AspectJML uses JML in addition to AspectJ's pointcut designators to specify crosscutting contracts.

There are several other interface technologies that are related to ours [19, 34, 48]. However, none of them can modularize crosscutting contracts and keep DbC benefits such as documentation. None of these checks contracts of base code.

6. Summary

AspectJML is an aspect-oriented extension to JML that enables the explicit specification of crosscutting contracts for Java code. It uses a mechanism called crosscutting contract specification (XCS). With XCS, AspectJML supports specification and runtime checking for crosscutting contracts in a modular way.

Using AspectJML, allows programmers to enable modular reasoning in presence of crosscutting contracts, and to recover the main DbC benefits such as documentation. Also, AspectJML gives programmers limited control over modularity respecting specifications. An AspectJML programmer cannot implicitly add contracts to unrelated modules. Therefore, using AspectJML, programmers get modular reasoning benefits at any time.

Acknowledgements

We thank Eric Eide, Eric Bodden, Mario Südholt, Arndt Von Staa, David Lorenz and Mehmet Aksit for fruitful discussions (we had during the AOSD 2011, more specifically at the Miss 2011 workshop) about design by contract modularization in general.

Special thanks to Mira Mezini, Ralf Lämmel, Yuanfang Cai, and Shuvendu Lahiri for detailed discussions and for comments on earlier versions of this paper.

References

[1] M. Bagherzadeh, H. Rajan, and A. Darvish. On exceptions, events and observer chains. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD '13, pages 185–196, New York, NY, USA, 2013. ACM.

[2] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 141–152, New York, NY, USA, Mar. 2011. ACM.

[3] S. Balzer, P. T. Eugster, and B. Meyer. Can Aspects Implement Contracts. In *In: Proceedings of RISE 2005 (Rapid Implementation of Engineering Techniques*, pages 13–15, September 2005.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: an overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Post Conference Proceedings of CASSIS: Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille*, volume 3362 of *LNCS*. Springer-Verlag, 2005.

[5] J. Boner. Aspectwerks. http://aspectwerkz.codehaus.org/.

[6] L. C. Briand, W. J. Dzidek, and Y. Labiche. Instrumenting Contracts with Aspect-Oriented Programming to Increase Observability and Support Debugging. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 687–690, Washington, DC, USA, 2005. IEEE Computer Society.

[7] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Softw. Pract. Exper.*, 33:637–672, June 2003.

[8] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.

[9] P. Chalin and P. R. James. Non-null references by default in java: alleviating the nullity annotation burden. In *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, pages 227–247, Berlin, Heidelberg, 2007. Springer-Verlag.

[10] Y. Cheon and G. T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In H. R. Arabnia and Y. Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[11] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31:25–37, May 2006.

[12] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996. A corrected version is ISU CS TR #95-20c, `http://tinyurl.com/s2krg`.

[13] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, New York, NY, USA, 2010. ACM.

[14] Y. A. Feldman et al. Jose: Aspects for Design by Contract80-89. *IEEE SEFM*, 0:80–89, 2006.

[15] R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Technical report, 2000.

[16] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. On the impact of aspectual decompositions on design stability: An empirical study. In *Proceedings of the 21st European conference on Object-Oriented Programming*, LNCS, pages 176–200. Springer-Verlag, 2007.

[17] S. Hanenberg and R. Unland. AspectJ idioms for aspect-oriented software construction. In *EuroPlop'03*, 2003.

[18] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 161–173, New York, NY, USA, 2002. ACM.

[19] M. Inostroza, E. Tanter, and E. Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 508–511, New York, NY, USA, 2011. ACM.

[20] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting tarted with AspectJ. *Commun. ACM*, 44:59–65, October 2001.

[21] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin / Heidelberg, 1997.

[22] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 49–58, New York, NY, USA, 2005. ACM.

[23] Y. Le Traon, B. Baudry, and J.-M. Jezequel. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng.*, 32(8):571–586, Aug. 2006.

[24] 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 of *Lecture Notes in Computer Science*, pages 2–34, New York, NY, Nov. 2006. Springer-Verlag.

[25] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 2006.

[26] G. T. Leavens and D. A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report CS-TR-13-03a, Computer Science, University of Central Florida, Orlando, FL, 32816, July 2013.

[27] M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 418–427, New York, NY, USA, 2000. ACM.

[28] C. V. Lopes, M. Lippert, and E. A. Hilsdale. Design By Contract with Aspect-Oriented Programming. In *U.S. Patent No. 06,442,750*, issued August 27, 2002.

[29] M. Marin, L. Moonen, and A. van Deursen. 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.

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

[31] B. Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[32] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, PTR, 2nd edition, 2000.

[33] R. Mitchell, J. McKim, and B. Meyer. *Design by contract, by example*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[34] A. C. Neto, A. Marques, R. Gheyi, P. Borba, and F. Castor. A Design Rule Language for Aspect-Oriented Programming. In *SBLP '09: Proceedings of the 2009 Brazilian Symposium on Programming Languages*, pages 131–144. Brazilian Computer Society, 2009.

[35] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[36] D. L. Parnas. Precise Documentation: The Key to Better Software. In S. Nanz, editor, *The Future of Software Engineering*, pages 125–148. Springer Berlin Heidelberg, 2011.

[37] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. Zimmerman, M. Cornélio, and T. Thüm. Aspectjml: Expressive specification and runtime checking for crosscutting contracts. 2013. Available from: `http://cin.ufpe.br/~hemr/modularity14`.

[38] H. Rebêlo, G. T. Leavens, and R. Lima. Modular enforcement of supertype abstraction and information hiding with client-side checking. Technical Report CS-TR-12-03, 4000 Central Florida Blvd., Orlando, Florida, 32816-2362, Jan. 2012.

[39] H. Rebelo, G. T. Leavens, R. M. F. Lima, P. Borba, and M. Ribeiro. Modular aspect-oriented design rule enforcement with XPIDRs. In *Proceedings of the 12th workshop on Foundations of aspect-oriented languages*, FOAL '13, pages 13–18, New York, NY, USA, 2013. ACM.

[40] H. Rebêlo, R. Lima, U. Kulesza, C. Sant'Anna, Y. Cai, R. Coelho, and M. Ribeiro. Quantifying the Effects of Aspectual Decompositions on Design By Contract Modularization: A Maintenance Study. *International Journal of Software Engineering and Knowledge Engineering*, 2013.

[41] H. Rebêlo, R. Lima, and G. T. Leavens. Modular Contracts with Procedures, Annotations, Pointcuts and Advice. In *SBLP '11: Proceedings of the 2011 Brazilian Symposium on Programming Languages*. Brazilian Computer Society, 2011.

[42] H. Rebêlo, R. Lima, G. T. Leavens, M. Cornélio, A. Mota, and C. Oliveira. Optimizing generated aspect-oriented assertion checking code for JML using program transformations: An empirical study. *Science of Computer Programming*, 78(8):1137 – 1156, 2013.

[43] H. Rebêlo, S. Soares, R. Lima, L. Ferreira, and M. Cornélio. Implementing Java modeling language contracts with AspectJ. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08. ACM, 2008.

[44] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, Jan. 1995.

[45] T. Skotiniotis and D. H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 196–197, New York, NY, USA, 2004. ACM.

[46] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 174–190, New York, NY, USA, 2002. ACM.

[47] F. Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *OOPSLA 2006: Proceedings of the 21st International Conference on Object-oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices, pages 481–497, New York, NY, Oct. 2006. ACM.

[48] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, 20(1):1:1–1:43, July 2010.

[49] K. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *ACM Transactions on Software Engineering and Methodology*, 20(2):5:1–5:42, Sept. 2010.

[50] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying design by contract to feature-oriented programming. In *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 255–269, Berlin, Heidelberg, 2012. Springer-Verlag.

[51] M. T. Valente, C. Couto, J. Faria, and S. Soares. On the benefits of quantification in AspectJ systems. *Journal of the Brazilian Computer Society*, 16(2):133–146, 2010.

[52] J. Zhao and M. Rinard. Pipa: a behavioral interface specification language for AspectJ. In *Proceedings of the 6th international conference on Fundamental approaches to software engineering*, FASE'03, pages 150–165, Berlin, Heidelberg, 2003. Springer-Verlag.

A. Online Appendix

We invite researchers to replicate our case study. Source code of the JML and AspectJML versions of the running example and HW systems, and other resources are available at [37].