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 M. 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]Harvey Mudd College, Claremont, CA, 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 have found that the realization of design by contract (DbC) is crosscutting and fares better when modularized by AOP. However, previous efforts aimed at supporting crosscutting contract modularity might actually compromise the main DbC principles. For example, in AspectJ-style, reasoning about the correctness of a method call may require a whole-program analysis to determine what advice applies and what that advice does relative to DbC implementation and checking. Also, when contracts are separated from classes a programmer may not know about them and may break them inadvertently. In this paper we solve these problems with *AspectJML*, a new specification language that supports 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, aspect-oriented programming, crosscutting contracts, JML, AspectJ, AspectJML

1. Introduction

Design by Contract (DbC), originally conceived by Meyer [32], is a useful technique for developing a program using specifications. The key mechanism in DbC is the use of behavioral specifications called "contracts". Checking these contracts against the actual code at runtime has a long tradition in the research community [7, 11, 14, 25, 27, 44, 51]. This idea of checking contracts at runtime was popularized by Eiffel [33] in the late 80s. In addition to Eiffel, other DbC languages include the Java Modeling Language (JML) [27], Spec# [4], and Code Contracts [14].

It is claimed in the literature [6, 15, 21, 29–31, 40, 41, 45] that the contracts of a system are de-facto a crosscutting concern and fare better when modularized with aspect-oriented programming [22] (AOP) mechanisms such as pointcuts and advice [21]. The idea has also been patented [30]. 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. Indeed, they go further to 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 [33] and JML [27] also have problems when dealing with crosscutting contracts. Although mechanisms such as invariant declarations help avoid scattering of specifications, the basic mechanisms for pre- and postcondition specification 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 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.

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.

http://dx.doi.org/10.1145/2577080.2577084

JML Contracts

```
01  class Package {
02   double width, height;
03   //@ invariant this.width > 0 && this.height > 0;
04   double weight;
05   //@ invariant this.weight > 0;
06
07   //@ requires width > 0 && height > 0;
08   //@ requires width * height <= 400; // max dimension
09   //@ 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
124  privileged aspect GiftPackageContracts {...}
125
126  privileged aspect CourierContracts {...}
127
128  aspect 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 [35].

2. Design by Contract and Modularity

In this section we discuss three existing problems in modularizing crosscutting contracts in practice. The first two problems are related to AOP/AspectJ [21, 22], and the third problem is related to design by contract languages like JML [27].

2.1 A Running Example

Figure 1 illustrates a simple delivery service system [35] that manages package delivery. It uses contracts expressed in JML [27] (lines 1-66) and AspectJ [21] (lines 67-126). In addition, the system

includes a crosscutting concern, tracing, modularized with AspectJ (lines 128-132).

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

The AspectJ code that corresponds to the JML+Java code is shown in lines 67-126. The main motivation in applying an AspectJ-like language is that we can explore some modularization opportunities that are otherwise not possible in a DbC language like JML. For instance, in the PackageContracts aspect, the second **before** advice (lines 92-96) checks the common preconditions, which are scattered in the JML side, for any method with name ending in Size and taking two arguments of type **double**. Similarly, the **after-returning** advice (lines 104-109) checks the common postconditions for both methods setSize and reSize. This advice only enforces the constraints after normal termination. In JML, postconditions normal postconditions are only required to hold when a method returns normally [27]. A third example is the **after-throwing** advice (lines 111-114), which forbids any method in Package or subtypes from throwing 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 [27].

2.2 The Modular Reasoning Problem

If we consider plain JML/Java without AspectJ, the example in Figure 1 supports modular reasoning [26, 28, 34, 39]. For example, suppose one wants to write code that manipulates objects of type Package. One could reason about Package objects using just Package's contract specifications (lines 1-50) in addition to any specification inherited from its supertypes [13, 26, 28].

Consider the Java and AspectJ implementation of the delivery service system (without the JML specifications). This is represented by the right-hand-side of Figure 1. In addition to the classes in the base/Java code, Figure 1 defines three aspects for contract checking and one aspect for tracing. In plain AspectJ, advice declarations are applied by the compiler without explicit reference to aspects from a module or a client module; therefore by definition, modular reasoning about the module Package does not consider the advice declared by these four aspects. The aspect behavior is only available via non-modular reasoning. That is, in AspectJ, a programmer must consider every aspect that refers to the 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, 20, 36, 39, 49].

2.3 Lack of Documentation Problem

In a design by contract language the pre- and postconditions and invariant declarations are typically placed directly in or next to the code they are specifying. Hence, contracts increase system documentation [3, 34, 37]. In AspectJ, however, the advising code (that checks contracts) is separated from the code it advises 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, 34, 37].

Consider now the tracing concern (Figure 1), modularized by the aspect Tracing. It prints a tracing message after the successful execution of any method in the 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 this advice runs after those advice (in the PackageContracts aspect) responsible for checking the Package's invariant. Without either documentation or the use of AspectJ's **declare precedence** [21] to enforce a specific order on aspects, it is quite difficult–perhaps impossible–to understand the order in which pre- and postconditions will be executed until they are actually executed.

Another problem caused by the lack of documentation implied by separating contracts as aspects is discussed by Balzer, Eugster, Meyer's work [3]. They argue that programmers become aware of contracts only when using special tools like AJDT [23], they are more likely to forget to account for 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 to crystallize our thinking about the goals of a DbC language and about the parts of such languages that provides good documentation, modular reasoning, and contracts in general without obliviousness. One straightforward way to avoid the previous two problems discussed is to use a plain DbC language like JML [27].

We make two observations about the JML specifications in Figure 1. First, 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 (and its subclasses), we 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 [19, 21] that can arise in specifications. For example, consider the JML specifications illustrated in lines 1-66 of Figure 1. In this example there are three ways in which crosscutting contracts are not properly modularized with plain JML constructs:

(1) We cannot write preconditions constraining 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) only once and apply them to these or other methods with the same design constraint;

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

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

[1] Even though inheritance is not exactly a crosscutting structure [19, 21], most DbC languages avoid repeating contracts for overriding methods.

2.5 The Dilemma

It is clear that we face a dilemma with respect to crosscutting contracts. If we use AspectJ to modularize them, the result is a poor contract documentation and compromised modular reasoning. If we go back to a DbC language such as JML, we face the scattered nature of common contracts shown previously. 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 while also specifying 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 problems discussed previously.

3. The AspectJML Language

AspectJML extends JML [27] with support for crosscutting contracts [31]. 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 the *crosscutting contract specification* mechanism, or XCS for short.

XCS in AspectJML is based on a subset of AspectJ's constructs [21]. However, since JML is a design by contract language tailored for plain Java, we need special support to use the traditional AspectJ syntax. To simplify the adoption of AspectJML, the included AspectJ constructs are based on the alternative @AspectJ syntax [5].

The @AspectJ (often pronounced as "at AspectJ") syntax was conceived as a part of the merge of standard AspectJ with AspectWerkz [5]. This merge enables crosscutting concern implementation by using constructs based on the 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, allowing the modularized code using AspectJ to work better with conventional Java IDEs and other tools that do not understand the traditional AspectJ syntax. In particular, this restriction applies to the JML common tools, including the JML compiler on which `ajmlc`, the AspectJML compiler, 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 the 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 how AspectJML supports crosscutting contract specification. The presentation is informal and based on our running example.

3.1 XCS with Pointcuts-Specifications

The combination of pointcuts and specifications is AspectJML's 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* [21]. Optionally, a pointcut can also include some of the values in the execution context of intercepted join points. In AspectJML, we combine these AspectJ pointcuts with JML specifications.

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

 @AfterReturning("trace()")
 public void afterReturningAdvice(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 our AspectJML compiler (ajmlc). 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, 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 we do not require them. This makes AspectJML simpler and a programmer only needs to know AspectJ's pointcut language [21] in addition to the main JML features.

Specifying crosscutting preconditions

Recall our first crosscutting contract scenario described in Section 2.4. It consists of two preconditions for any method, in `Package` (Figure 1) with a name ending with `Size` that returns **`void`** and takes two arguments of type **`double`**. 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 sizes(double width, double height) {}
```

The pointcut `sizes` matches all the executions of methods ending with "`Size`" of class `Package` like `setSize` and `setWeight`. 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. We bind the parameter names in the pointcut's expression (within the annotation **`@Pointcut`**) using the argument-based pointcut **`args`** [21].

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.

Specifying crosscutting postconditions

We discuss now how to properly modularize crosscutting postconditions in AspectJML. JML supports two kinds of postconditions: normal and exceptional. Normal postconditions constrain methods that return without throwing an exception. To illustrate AspectJML's design, we discuss scenarios (2) and (3) from Section 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 sizeChange(double width, double height) {}
```

This pointcut constrains the executions of the `setSize` and `reSize` methods in `Package` to ensure that, after their executions, the fields `width` and `height` have 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 packageMeths() {}
```

The above specification forbids the execution of any method in `Package` (or a subtype, such as `GiftPackage`) to throw an exception. If any intercepted method throws 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 specification in a pointcut declaration. For example, assume that the `Package` type in Figure 1 does not include the `containsSize` method or its JML specifications. In this scenario, we can write a single pointcut to modularize the recurrent pre- and postconditions of methods `setSize` and `reSize`. Therefore, instead of having separate JML annotated pointcuts for each crosscutting contract, we specify them in a new version of the pointcut `sizes`:

```
//@ 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 sizes(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`.

Pointcut expressions without type signature patterns

In AspectJ, a pointcut expression can be defined without using a type signature pattern. A type signature pattern is a name (or part of a name) used to identify what type contains the join point. For example, the following AspectJ pointcut:

```
pointcut sizes(): execution(* *Size(double, double));
```

selects any method ending with "`Size`" and has two arguments of type **`double`**. In AspectJ, this pointcut matches any type in a system. Since we omit the type signature pattern, any type is candidate to expose the join points of interest. In AspectJ, although not required, we can also use a wildcard (`*`) to represent a type signature pattern that intercepts any type in the system (i.e., **`execution`**`(* *.*Size(`**`double, double`**`))`.

However AspectJML has a different semantics compared with AspectJ. For example, recall the previous pointcut method `sizes` in AspectJML:

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

this pointcut method still selects the same methods ending with "`Size`" and that has two arguments of type **`double`**. The main difference is that even with the absence of the target type, AspectJML restricts the join points to the type (`Package` in this case) enclosing the pointcut declaration (see Figure 3). AspectJML works in this manner to avoid the obliviousness problem (see Section 4 for more details).

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 [16, 52] 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, 20, 39, 47–49]. Instead of ruling this completely out, the design of AspectJML allows the specifier to use specifications that constrain unrelated inter-types, but in an explicit and limited manner (see Section 4 for more details about non-obliviousness in AspectJML).

As an example, recall the 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 `deliver` method in type `Courier` also has this constraint. Note that the type `Courier` is not a subtype of `Package`. They are related in the sense that the method `deliver` depends on the `Package` type due to the declaration of a formal parameter. Consider further that `Courier` contains many methods that are not dependent on `Package` in any way. Consider the following type declaration:

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

This type declaration illustrates how we specify crosscutting contracts for interfaces. In @AspectJ, 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 refers only 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 need only to implement the interface `ExceptionSignallingConstraint`. Such an interface acts like a marker interface [18]. This is important to avoid obliviousness and maintain modular reasoning.

Note that the inner class is marked with the **`@InterfaceXCS`** annotation. This is to distinguish from any other inner class that could be also declared within our crosscutting contract interface. Without this mechanism, the AspectJML compiler will not be able to find the crosscutting contracts for the interface `ExceptionSignalling`.

Collected XCS examples

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

3.2 AspectJML Expressiveness

So far we have used the **`execution`** and **`within`** pointcut designators to select join points. This conforms with the supplier-side

[2] Inter-types here are not the AspectJ feature [21] that allows adding methods or fields with a static crosscutting mechanism. Instead, they are unrelated modules in a system; that is, types that are not related to each other but can present a common crosscutting contract structure.

```
01 class Package {
02  double width, height;
03  //@ invariant this.width > 0 && this.height > 0;
04  double weight;
05  //@ invariant this.weight > 0;
06
07  //@ requires width > 0 && height > 0;
08  //@ requires width * height <= 400; // max dimension
09  @Pointcut("execution(* *Size(double,double))"+
10   "&& args(width, height)")
11  void sizes(double width, double height) {}
12
13  //@ ensures this.width == width;
14  //@ ensures this.height == height;
15  @Pointcut("(execution(* setSize(double,double))"
16   + "|| execution(* reSize(double, double)))"+
17   "&& args(width, height)")
18  void sizeChange(double width, double height) {}
19
20  //@ signals_only \nothing;
21  @Pointcut("execution(* Package+.*(..))")
22  void packageMeths() {}
23
24  void setSize(double width, double height){...}
25
26  //@ requires this.width != width;
27  //@ requires this.height != height;
28  void reSize(double width, double height){...}
29
30  boolean containsSize(double width, double height){...}
31  double getSize(){...}
32
33  //@ ...
34  void setWeight(double weight) {...}
35 ... // other methods
36 }
37 class GiftPackage extends Package {
38  //@ ...
39  void setWeight(double weight) {...}
40  ... // other methods
41 }
```

Figure 3. The crosscutting contract specifications used so far for the delivery service system [35] with AspectJML.

checking adopted by most runtime assertion checkers (RAC) of DbC languages. Such RAC compilers typically 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 [21]. For instance, we can use the **`call`** pointcut. This would provide runtime checking at the call site. Code Contracts [14] is an example of a DbC language that provides runtime checking at the call site. However, 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`**) [21].

4. AspectJML's Benefits

In this section we discuss the main AspectJML benefits when used for crosscutting contract specification.

Enabling modular reasoning

Recall that our notion of modular reasoning means that one can 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 it references [13, 26, 28, 34, 39].

With respect to whether or not AspectJML supports modular reasoning like a DbC language such as JML, consider the client code, which we will imagine is written by Cathy, shown in Figure 4.

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

Figure 4. `setSize`'s Client code.

To verify the call to `setSize`, Cathy must determine what specifications to use. If she uses the definition of modular reasoning [26, 28, 34, 39], 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 in line 3, which 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 steps, including seven JML pre- and postcondition, and invariant specifications, when using plain JML reasoning. (`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 4).

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

(1) The first invariant definition in 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) The normal postconditions (lines 13-14) located at the pointcut `setOrReSize`; and

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

As before, this task involves only modular reasoning and she can still detect the potential precondition violation related to `Package`'s width. In this case, Cathy needed the same seven specifications, but with two more steps (five in total) to reason about the correctness of the call to `setSize`. 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 [16] does

not occur in this example. Cathy is completely aware of the contracts of `Package` class, though it does take her longer to determine them.

Enabling documentation

This example 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 contract specifications, which is necessary to avoid scattering (repeated specifications). However, using AspectJML, users also have several new possibilities for crosscutting contracts.

Taming obliviousness

Since AspectJML allows pointcut declarations in AspectJ-style, one can argue that a programmer can specify several unrelated modules in one single place. This phenomenon brings into focus again whether AspectJML allows the controversial obliviousness property of AOP [2, 20, 39, 47–49].

The answer is no. AspectJML rules out this possibility. If one tries to write such pointcuts, they will have no effect with respect to crosscutting specification and runtime checking. This happens because AspectJML associates the specified pointcut with the type in which it was declared (see the discussion in the next section and the generated code in Figure 5). Hence, only join points within the given type or its subtypes are allowed. The cross-references generated by AspectJML (see Section 5.1) 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. Those pointcuts can be used to modularize other kinds of crosscutting concerns using the standard AspectJ pointcuts-advice mechanisms [21].

5. Implementation

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/%7ehemr/JMLAOP/ajmlc.htm`. To the best of our knowledge, the AspectJML compiler is the first compiler for runtime assertion checking that supports 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 between the previous ajmlc and the new one is support for AspectJML features like specified pointcuts. Instead of saying JML/ajmlc, we now say AspectJML/ajmlc.

Figure 5 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` method, which checks such preconditions; if it is not true, then a precondition error is thrown. As discussed in Section 4, the exposed object type is `Package`. Hence, this precondition can only be checked to join points of `Package` or its subtypes like `GiftPackage` (see Figure 1).

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

```
/** 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 5. Generated before advice to check the crosscutting preconditions of `Package` in Figure 3.

Ordering of checks

As ajmlc generates AspectJ aspects to check contracts, it also enforces aspect precedence. For instance, if we have advising code for other crosscutting concerns, it can only be allowed to execute after the preconditions are satisfied; otherwise, a precondition violation is thrown.

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

Contract violation example in AspectJML

As an example of runtime checking using AspectJML/ajmlc, recall the client code illustrated in Figure 4. 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 be greater than zero. As a result, this precondition violation occurs during runtime checking when calling such client code.

5.1 Tool Support

In aspect-oriented programming, development tools like AJDT [23], allow programmers to easily browse the crosscutting structure of their programs. For AspectJML, we are developing analogous support for browsing crosscutting contract structure. We use the existing functionality of AJDT to this end.

For example, consider the crosscutting contract structure of the `Package` class using AspectJML/AJDT (see Figure 6). Note the arrows indicating where the crosscutting contracts apply. In plain AspectJ/AJDT this example shows no crosscutting structure information, because it has only 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. We have implemented an option (that is enabled by default) in AspectJML that generates the cross-references information for crosscutting contracts, thus allowing one to visualize the crosscutting structure.

To enable the crosscutting contract structure view, AspectJML generates an **`around`** advice in AspectJ, without effect in the base code, to associate with the corresponding pointcut in AspectJML pointcuts. For instance, considering the `Package` type (Figure 6), AspectJML generates an AspectJ aspect called `PackageCrossRef`. Figure 6 illustrates this in practice. Once compiled, we can see that the method `reSize` in Package is intercepted by the pointcut sizes from `PackageCrossRef`. Through

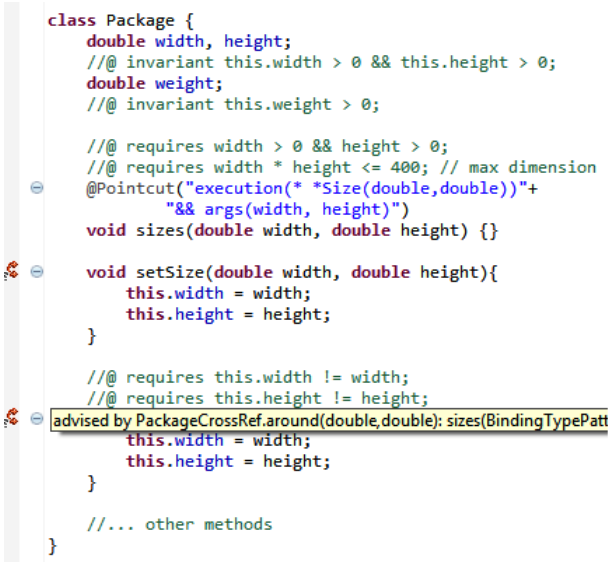

Figure 6. The crosscutting contract structure in the `Package` class using AspectJML/AJDT [23].

the cross-references, we go to the **`around`** advice (in the aspect `PackageCrossRef`) that actually activates the arrow through AJDT. But the cross-reference code we generate for AspectJML allows the programmers from a javadoc-link to point to the right pointcut `sizes` in type `Package`. The generated code looks as follows:

```
/** Generated by AspectJML to enable the crossref for
 * the XCS pointcut {@link Package#sizes(double, double)}*/
  pointcut sizes(double width, double height): ... ;
  Object around(...): sizes(width, height) && ...
```

Figure 7 shows another example where the use of the AspectJ/AJDT helps 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 6. All the AJDT IDE validation is inherited by AspectJML.

Note that the AJDT is just a helpful functionality to assist (beginners) AspectJML programmers to see where the specified pointcuts intercept. Once pointcut language and quantification mechanism are understood, this tool is not required to reason about AspectJML in a modular way (as discussed in Section 4).

6. 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) [17, 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 [17]. 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:
`http://www.cin.ufpe.br/%7ehemr/modularity14/`.

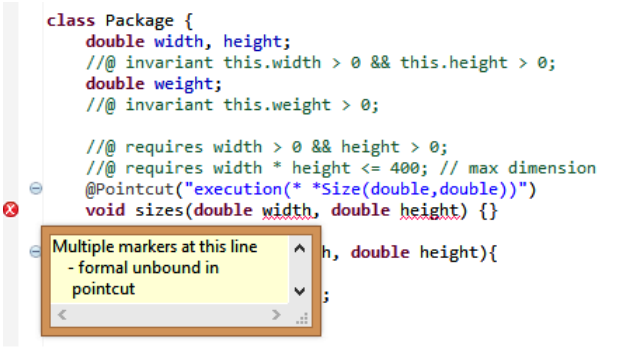

Figure 7. An example of a malformed pointcut declaration in AspectJML.

6.1 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 the 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;
IteratorDsk searchSpecialitiesByHealthUnit(int code);

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

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

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

//@ requires healthUnitCode >= 0;
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 all 5 search-based methods.

Crosscutting postconditions

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

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

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

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

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

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

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

As observed, all the methods in `IFacade` that return an object of type `IteratorDsk` should return a non-null object reference. In standard JML there are two more ways to express this constraint [9]. The first one uses 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 most object references in the system are possibly 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 option is to use the JML type modifier **`non_null`**; however, even this would lead to some (smaller) amount of repeated postconditions.

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

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

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

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

//@ signals_only java.rmi.RemoteException;
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 the 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 is repeated for all methods in the `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 occurs with 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 it to all search-based methods.

6.2 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 is 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

We can properly modularize the crosscutting preconditions of HW 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 preconditions for all the search-based methods in a single place. To select the search-based methods, we use a property-based pointcut [21] that matches join points by using wildcards. Our pointcut matches any method starting with `search` and taking an **`int`** parameter. 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

Consider the modularization of the two kinds of crosscutting postconditions we discussed previously. For normal postconditions, we add 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 returning `IteratorDsk`.

The AspectJML code responsible for modularizing the exceptional postconditions is similar:

```
//@ 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 can throw `RemoteException` and methods 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.

6.3 Reasoning About Change

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 [16, 52] inherited from AspectJ [21]. In addition to the documentation and modularization of crosscutting contracts achieved with AspectJML, another immediate benefit of using our approach is easier software maintenance. For example, 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 `remoteExceptionalMeths` pointcut. This pointcut can be reused whenever we want to apply constraints to methods already intercepted by the pointcut.

Another maintenance benefit occurs during system evolution. On one hand, we may add more methods in the `IFacade` interface to handle system's new use cases. On the other hand, we do not need to explicitly apply existing constraints to the newly added methods. The modularized contracts that apply to all methods also automatically apply to the newly 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 apply to it through the cross-references feature of AJDT [23].

7. Discussion

This section discusses some issues with the AspectJML specification language, including limitations, compatibility, open issues, and related work.

7.1 Limitations of AspectJML

Even though AspectJML has the benefit of modularity when handling crosscutting contracts, there are 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 it 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 also a limitation of AspectJ's pointcut mechanism.

Another limitation of AspectJML is related to the crosscutting contract interfaces. Such interfaces are the ones we use to explicit make a crosscutting contract to be applied to several unrelated types. To enable the modular reasoning and checking of these contracts one just needs to implement such interfaces. The problem is that modular reasoning will be lost if a programmer makes a type implement one of these crosscutting contract interfaces by using AspectJ's **declare parents** feature. This feature would make the contracts modularized in a crosscutting contract interface to be implicit applied. Therefore, a programmer must take care of this risk if they decide to use AspectJ's feature combined with AspectJML.

7.2 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 @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.

7.3 JML Versus AspectJ

We have discussed the main problems of dealing with contracts expressed in both JML and AspectJ. Indeed, this comparison was suggested by Kiczales and Mezini [23]. 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 situations involving scattering of contracts it can be advantageous to provide a form of specified pointcuts to allow crosscutting contract specifications.

7.4 Open Issues

Our evaluation of AspectJML is limited to two systems, the delivery service system [35] 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 Section 7.1).

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 because we can also program in AspectJ using AspectJML.

7.5 Related Work

As discussed throughout the paper, there are several works in the literature that argue in favor of implementing DbC with AOP [15, 21, 30, 41]. Kiczales opened this research avenue by showing a simple precondition constraint implementation in one of his first papers on AOP [21]. After that, other authors explored how to implement and separate the DbC concern with AOP [15, 21, 30, 40, 41]. All these works offer common templates and guidelines for DbC aspectization.

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.

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 easy to specify 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 [53] 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.

As with our work with AspectJML, Lam, Kuncak, and Rinard [24] advocate that previous research in the field of aspect-oriented programming has focused on the use of aspect-oriented concepts in design and implementation of crosscutting concerns. Their experience indicates that aspect-oriented concepts can also be extremely useful for specification, analysis, and verification. In this sense, among other things, Lam, Kuncak, and Rinard designed constructs called *scopes* and *defaults* that can be used to improve the locality and clarity of specifications, and, at the same time, reducing the

sizes of these specifications. These constructs cut across the preconditions and postconditions of procedures in a system. Like our work, their language provides a pointcut language to select where to apply constraints. The main difference is that their pointcut is not expressive as ours (since we reused the standard AspectJ pointcut language). For instance, in their language one cannot use wildcarding to select join points. Also, their pointcut language can apply to several modules, but this can break modular reasoning (which we preserve in AspectJML). Finally, their work is intended for specification and verification, whereas we are concerned with specification and runtime checking.

While AOP aims at the modularization of homogeneous *and* heterogeneous crosscutting concerns, feature-oriented programming (FOP) focuses on heterogeneous crosscutting concerns only, for which only lightweight language extensions are required. Contracts and their use for runtime assertions have also been studied in the context of FOP [50, 51]. We discussed specification clones in the base code, but they may also occur in crosscutting modules. Interestingly, some of these specification clones can actually be avoided by language constructs beyond quantification [51, 52]. A further challenge is to identify the crosscutting module that caused the violation of a contract. Behavioral feature interfaces have been proposed for localization by means of runtime assertions [50].

There are several other interface technologies that are related to ours [12, 20, 48–51]. However, none of them can modularize crosscutting contracts and keep DbC benefits such as documentation and modular reasoning at the same time. None of these checks contracts of base code.

8. Summary

AspectJML is a seamless aspect-oriented extension to JML. Specifying with AspectJML feels like a small extension of specifying with JML. AspectJML uses a mechanism called crosscutting contract specification (XCS). XCS enables one to explicit specify crosscutting contracts for Java code in a modular way. Also, runtime assertion checking is used to check the conformance of these crosscutting contracts during runtime.

Using AspectJML results in clean, well-modularized specifications of crosscutting contracts. In AspectJML, programmers do not need to use several syntactic constructs of AspectJ-like languages to enable crosscutting contract modularization. On the other hand, when written as AspectJML, the structure of a crosscutting contract is explicit and easy to understand, due to the benefits of DbC-style documentation.

Acknowledgements

We thank Eric Eide, Eric Bodden, Mario Südholt, Arndt Von Staa, David Lorenz and Mehmet Aksit for 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.

The work of Leavens was partially supported by US National Science Foundation grants CCF-10-17262 and CCF-1017334. Ricardo Lima is also supported by CNPq under grant No. 314539/2009-3. Also, the work of Rajan and Bagherzadeh was supported by US NSF grants CCF-11-17937 and CCF-08-46059.

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] A. Costa Neto, R. Bonifácio, M. Ribeiro, C. E. Pontual, P. Borba, and F. Castor. A Design Rule Language for Aspect-oriented Programming. *J. Syst. Softw.*, 86(9):2333–2356, Sept. 2013.

[13] 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`.

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] P. Lam, V. Kuncak, and M. Rinard. Crosscutting Techniques in Program Specification and Analysis. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 169–180, New York, NY, USA, 2005. ACM.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

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

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

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

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

[37] 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.

[38] H. Rebêlo, G. T. Leavens, and R. M. Lima. Client-aware Checking and Information Hiding in Interface Specifications with JML/Ajmlc. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, pages 11–12, New York, NY, USA, 2013. ACM.

[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 Trans. Softw. Eng. Methodol.*, 20(2):5:1–5:42, Sept. 2010.

[50] T. Thüm, S. Apel, A. Zelend, R. Schröter, and B. Möller. Subclack: Feature-oriented Programming with Behavioral Feature Interfaces. In *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance*, MASPEGHI '13, pages 1–8, New York, NY, USA, 2013. ACM.

[51] 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.

[52] 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.

[53] 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:
`http://www.cin.ufpe.br/%7ehemr/modularity14/`.