Modularizing Crosscutting Contracts with AspectJML

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

It is claimed in the literature that the contracts of a system present crosscutting structure during its realization. In this context, there has been attempts to improve separation of crosscutting contracts, e.g. by aspect-oriented programming and design by contract languages, but none give programmers textual separation of contracts/specifications and modular reasoning at the same time.

In this demonstration we show how our language, AspectJML, a simple and practical aspect-oriented extension to JML, allows the separation of crosscutting contracts while maintaining the key benefits of a design by contract language, like documentation and modular reasoning. AspectJML's quantified statements, written in terms of AspectJ pointcut language, allow one to select join points in which the contracts are written in a modular and convenient way. Also, all the crosscutting contracts are well documented in the class they apply to, thus allowing the reasoning about crosscutting contracts in a modular fashion.

This demonstration will proceed by discussing several examples that highlights the main features of the AspectJML language and show the use of AspectJML's compiler, ajmlc. We will conclude with a discussion of ongoing work on design, implementation and runtime checking of both Java and AspectJ programs with AspectJML/ajmlc.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Modules and Packages

General Terms Design, Languages

Keywords Design by contract, aspect-oriented programming, crosscutting contracts, JML, AspectJ, AspectJML

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

ACM 978-1-4503-2773-2/14/04.
http://dx.doi.org/10.1145/2584469.2584476

1. Background and Problem Statement

Design by Contract (DbC) is a technique for developing and improving functional software correctness [11]. The key mechanism in DbC is the use of the so-called "contracts". A contract formally specifies an agreement between a client and its suppliers. Clients must satisfy the supplier's conditions before calling one of the supplied methods. When these conditions are satisfied, the supplier guarantees certain properties, which constitute the supplier's obligations. When a client or supplier breaks a condition (contract violation), a runtime error occurs. The use of such pre- and postconditions to specify software contracts dates back to Hoare's 1969 paper on formal verification [4].

It is claimed in the literature [2, 3, 5, 8–10, 16, 17, 19] that the contracts of a system tend to be crosscutting since they cannot be properly modularized. In this context, numerous mechanisms have been developed to instrument, modularize, and document contracts at source code level, including procedures, design by contract languages, pointcuts and advice [5]. The latter two are well-known aspect-oriented programming (AOP) mechanisms and according to the literature are the best ones to properly deal with *crosscutting contracts*. This idea has also been patented [9].

In this context, Balzer, Eugster, and Meyer [1] were the first to investigate the adequacy of aspects to modularize DbC. They conclude that the use of aspects hinders design by contract implementation 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*" [1, Section 6.3]. According to them, contracts should appear in the modules themselves and separating such contracts as aspects contradicts this view [11].

However, plain DbC languages like Eiffel [12] and JML [7] also have problems when dealing with crosscutting contracts. 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.

As observed, the main problem here is a trade-off between the existing techniques like DbC and AOP. None of them give programmers textual separation of contracts/specifications and modular reasoning at the same time.

2. AspectJML

The AspectJML language [14] takes advantages from both design by contract languages like JML [7] and aspect-oriented programming languages like AspectJ [5]. One of our goals is to make programming and specifying with AspectJML feel like a natural extension of programming and specifying with Java and JML. In addition, AspectJML has three more design goals:

- Enable modularization of crosscutting contracts while maintaining the the key DbC principles such as documentation and modular reasoning,
- enable a well-defined interface between the crosscutting contracts and object-oriented code,
- enable typechecking, compilation, and runtime checking of crosscutting contracts.

As listed above, the modularization of crosscutting contracts is a key concept behind AspectJML. We call this feature as *crosscutting contract specification*, or XCS for short. To understand how it works, please consider the JML specifications for the type `Package` in Figure 1. Suppose that this `Package` type is part of a delivery service system, which manages package delivery within a city [13, pp.100-107]. So, in this example there are three scenarios 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 the `dimension` model field is equal to the multiplication of the method argument `width` with argument `height`; however, we cannot write a simple and local quantified statement for these postconditions and apply them to the constrained methods; and

(3) the exceptional postcondition **`signals_only \nothing`** must be explicitly written for all methods in `Package` which forbid exceptions; there is no way to modularize such a JML contract in one place and apply it to all constrained methods.

For instance, to solve our first crosscutting scenario, consider the following JML annotated pointcut:

```
//@ 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) {}
```

As observed, we define an AspectJ's pointcut with crosscutting preconditions. To be fully compatible to Java, the AspectJ constructs, we rely on, are based on the @AspectJ syntax, which are based on metadata annotations. So, in the example, we define a pointcut using the **`@Pointcut`** annotation. The method `sizes` represents a pointcut declaration since it is annotated with a **`@Pointcut`** annotation. This pointcut intercepts all the executions of methods in `Package` class that end with `Size` (see the use of wildcarding). In Contrast to AspectJ, this is the simplest way to modularize crosscutting contracts at source code level. The major difference is that a specified pointcut is always processed when using the AspectJML compiler (ajmlc). In standard AspectJ, a single pointcut declaration, without an associated advice, does not contribute to the execution flow of a program. In AspectJML, we do not need to define an advice to check a specification in a crosscutting fashion.

```
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
 //@ ensures this.width == width;
 //@ ensures this.height == height;
 //@ signals_only \nothing;
 void setSize(double width, double height){
  this.width = width;
  this.height = height;
 }

 //@ requires width > 0 && height > 0;
 //@ requires width * height <= 400; // max dimension
 //@ requires this.width != width;
 //@ requires this.height != height;
 //@ ensures this.width == width;
 //@ ensures this.height == height;
 //@ signals_only \nothing;
 void reSize(double width, double height){
  this.width = width;
  this.height = height;
 }

 //@ requires width > 0 && height > 0;
 //@ requires width * height <= 400; // max dimension
 //@ signals_only \nothing;
 boolean containsSize(double width, double height){
  if(this.width == width && this.height == height){
   return true;
  }
  else return false;
 }
... // other methods
}
```

Figure 1. The JML pre- and postconditions for `Package` class.

Figure 2 illustrates how the previous crosscutting contract specification is placed within the Package type. Also, note that we also provided pointcut-specifications for the other two crosscutting scenarios previously discussed (see the shadowed part). More details can be found at [14] and will be given in the demonstration sessions.

3. Benefits and Demonstration Overview

We begin our demonstration with an overview of contemporary languages like JML and AspectJ to deal with implementation/specification and checking of contracts.

The first benefit of attending our demo sessions will be the explicit recognition of the crosscutting contracts problem when specifying pre- and postconditions. Also, we show that languages such as JML and AspectJ are not enough to properly deal with the crosscutting structure of design by contract.

Second, attendees will see an improved form of a DbC language which enables the programmer to specify crosscutting contracts in a modular way while preserving the main DbC principles such as documentation and modular reasoning. At this stage, the examples will be given in AspectJML [14], which combines features from both JML and AspectJ. The AspectJML features we explain in this demo are implemented in the AspectJML/ajmlc compiler [14].

We believe that these benefits are important for the software engineering community as they can aid in further improving crosscutting contract specifications in DbC languages and runtime assertion checking.

```
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 sizes(double width, double height) {}

 //@ 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) {}

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

 void setSize(double width, double height){...}

 //@ requires this.width != width;
 //@ requires this.height != height;
 void reSize(double width, double height){...}

 boolean containsSize(double width, double height){...}
... // other methods
}
```

Figure 2. The crosscutting contract specifications for the three crosscutting scenarios discussed and illustrated in Figure 1.

3.1 Tool Support

In aspect-oriented programming, development tools like AJDT [6], 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 already provided functionality of AJDT to this end.

For example, consider the crosscutting contract structure of the `Package` class using AspectJML/AJDT (shown in Figure 3). 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.

Figure 4 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 3. All the AspectJ/AJDT IDE validation is inherited by AspectJML.

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

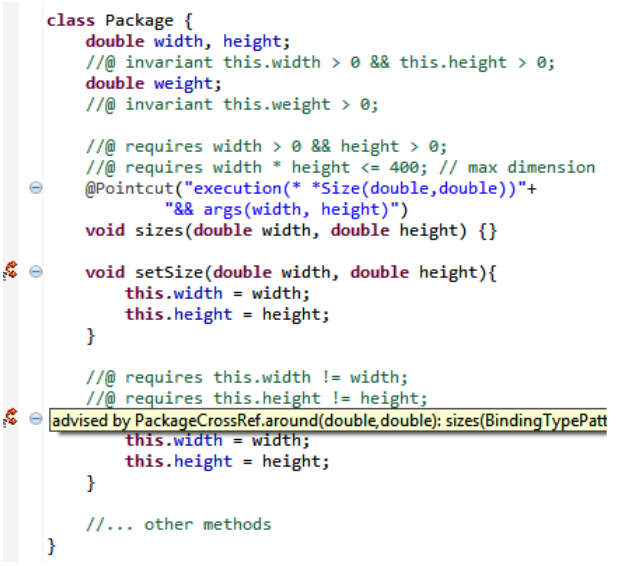

Figure 3. The crosscutting contract structure in the `Package` class using AspectJML/AJDT [6].

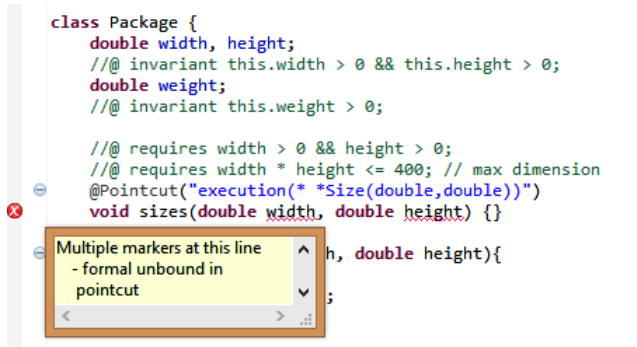

Figure 4. An example of a malformed pointcut declaration in AspectJML.

4. AspectJML Infrastructure

AspectJML is an open source project and is broadly available for download and for modication under GNU General Public License. The source code and examples for the AspectJML/ajmlc compiler [14, 15, 18] can be obtained from:
`http://www.cin.ufpe.br/%7ehemr/JMLAOP/ajmlc.htm`.

5. Presenter Biography

Henrique Rebêlo is one of the original authors and creators of the AspectJML language. He has extensive experience in separation of concerns and design by contract techniques. He is the main developer of the aspect-oriented JML compiler known as ajmlc [14, 15, 18]. This compiler uses aspect-oriented programming (AOP) for enforcing JML contracts at runtime. He was a researcher intern in 2010 at Microsoft Research working on program analysis and program verification. He has given talks on design by contract and AOP at prestigious venues like SPLASH'13, SPLASH'11, SEKE'11, FTFJP'11, SAVCBS'09, ICST'08, and SAC'08.

Acknowledgements

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 is supported by US NSF grants CCF-11-17937 and CCF-08-46059.

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

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

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

[14] H. Rebêlo, G. T. Leavens, M. Bagherzadeh, H. Rajan, R. Lima, D. Zimmerman, M. Cornélio, and T. Thüm. AspectJML: Modular Specification and Runtime Checking for Crosscutting Contracts. In *Proceedings of the Thirteenth International Conference on Modularity*, Modularity '14, New York, NY, USA, 2014. ACM.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.