Identifying and Specifying Crosscutting Contracts with AspectJML

Henrique Rebêlo
Universidade Federal de Pernambuco, PE, Brazil
hemr@cin.ufpe.br

Abstract

I propose 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 design by contract language, like documentation and modular reasoning.

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

General Terms Languages, Verification

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

1. Introduction

Design by Contract (DbC) is a useful technique for developing and checking program's correctness against its specification [7]. The key mechanism in DbC is the use of the so-called "contracts". Writing out these contracts in the form of specifications and verifying them against the actual code either at runtime or compile time has a long tradition in the research community. The idea of checking contracts at runtime was popularized by Eiffel [8] in the late 80's.

It is claimed in the literature [2, 3, 6, 10] that the contracts of a system are de-facto a crosscutting concern and fare better when modularized with AOP mechanisms such as pointcuts and advice [3]. The idea has also been patented [6]. However, Balzer, Eugster, and Meyer's study [1] 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 contract inheritance, documentation and modular reasoning. Also, they go further and say that "*no module in a system (e.g., class or aspect) can be oblivious of the presence of contracts*" [1, Section 6.3]. Indeed, as AOP is a form of implicit invocation with implicit announcement (IIA), it also compromises modular reasoning [12] for other kinds of crosscutting concerns, like distribution and persistence [11].

SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2514877

1.1 Problem and Motivation

Balzer, Eugster, and Meyer's study [1] helped crystallize my thinking about the goals of a design by contract language, in particular about the portion of such languages that provides good documentation, modular reasoning, and non contract-obliviousness.

However, there are situations that the quantification property of AOP can be beneficial to a contracted module. For example, recall the class `Point`, of the classical figure editor [3], and let us specify some contracts expressed in JML [5]. The result is illustrated in Figure 1. Both methods `setX` and `setY` of class `Point` have a precondition (denoted by the **`requires`** clause) that states the input parameter must be at least zero; both also have an exceptional postcondition (represented by the **`signals_only`** clause) that forbids the methods by throwing exceptions (which includes runtime exception). Finally, the invariant defined in this example restricts points to the upper right quadrant.

```
class Point {
 int x, y;
 //@ invariant x >= 0 && y >=0;

 //@ requires x >= 0;
 //@ signals_only \nothing;
 void setX(int x) {this.x = x;}

 //@ requires y >= 0;
 //@ signals_only \nothing;
 void setY(int y) {this.y = y;}
 ... // other methods
}
```

Figure 1. The contract specifications of the class `Point` with JML.

In relation to modular reasoning and documentation [4], one can reason about `Point` objects using just that type's and method's specifications, contained in Figure 1. So, with those documented contracts, there is no need to look at unrelated, separated, and non-documented modules [1] as we do when using, for example, AspectJ aspects [3].

At this point, let me make two viewpoints of the specifications in Figure 1. The first one is that a design by contract language like JML can be used to modularize some contracts. Hence, the single invariant clause can be viewed as a form of modularization provided by these languages. Instead of writing the same pre- and postconditions for all methods in a class, we just write a single invariant statement that "modularizes" those pre- and postconditions.

On the other hand, these design by contract languages (e.g., JML) do not capture other forms of crosscutting contracts that can arise in the specifications. For instance, the precondition that constrains the input parameter, of the both set methods, to be at least zero, cannot be written only once and applied to these and

other methods that can have the same design constraint. In the same sense, if we also assume that other methods are forbidden to throw exceptions, we need to explicitly write the same **`signals_only`** clause to these other methods as well. There is no way to express that which crosscuts like an invariant.

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

2. AspectJML

I propose AspectJML, a simple and practical aspect-oriented extension to JML. With just a few aspect-based constructs, AspectJML provides support for specifying crosscutting contracts in a modular and convenient way. The key concept behind AspectJML is what we call *crosscutting contract specifications*, or XCS.

Crosscutting Contract Specifications. Figure 2 illustrates the crosscutting contracts specifications for the `Point` class. As observed, I define a pointcut with all the crosscutting contract specifications. To be fully compatible to Java, the AspectJ constructs, I rely on, are based on the @AspectJ syntax, which are based on metadata annotations. So, in the example, I define a pointcut using the **`@Pointcut`** annotation. The method `setXY` represents a pointcut declaration since it is annotated with a **`@Pointcut`** annotation. This pointcut intercepts all the executions of set-like method declarations in the `Point` class. 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. Hence, we have the specified crosscutting precondition and exceptional postcondition checked in a modular way.

One benefit to use AspectJ syntax is that we can see when a pointcut declaration is well-formed. In other words, we can see the arrows indicating where the specifications will be checked during runtime. In plain AspectJ/AJDT this example show 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. Hence, I have implemented an option in AspectJML that generates the cross-references information for crosscutting contracts when we have only pointcut declarations.

Finally, with AspectJML, all the crosscutting contracts are well documented in the class it applies to. In the conventional approach, the aspects are separated from the type declarations and they do not provide a documented approach.

3. Related Work

As discussed throughout the paper, there are several works in the literature that argue in favor of implementing DbC with AOP [2, 3, 6, 10]. Kiczales opened this research avenue by showing a simple precondition constraint implementation in one of his first papers on AOP [3]. After that, other authors explored how to implement and separate the DbC concern with AOP [2, 3, 6, 9, 10]. All these works offer common templates and guidelines for DbC aspectization.

```
class Point {
    int x, y;
    //@ invariant x >= 0 && y >= 0;

    //@ requires xy >= 0;
    //@ signals_only \nothing;
    @Pointcut("execution(void Point.set*(int)) && args(xy)")
    void setXY(int xy){}

    void setX(int x) {this.x = x;}

    void setY(int y) {this.y = y;}
}
```

Figure 2. The AspectJML specifications of `Point` class.

However, as also discussed, DbC aspectization is more harmful than good [1]. I 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 I conceive the AspectJML specification language that is suitable for specifying crosscutting contracts.

4. Acknowledgments

I would like to thank Professors Gary T. Leavens and Ricardo Lima (my supervisors) for the discussions about the ideas of this work.

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] Y. A. Feldman, O. Barzilay, and S. Tyszberowicz. Jose: Aspects for Design by Contract80-89. *sefm*, 0:80–89, 2006.

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

[4] 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.

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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*, 2011.

[11] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with aspectj. In *Proceedings of the 17th conference on Object-oriented programming (OOPSLA), systems, languages, and applications*, 2002.

[12] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of OOPSLA 2006*, ACM SIGPLAN Notices, pages 481–497, New York, NY, Oct. 2006. ACM.