UNIVERSITY OF WATERLOO

CS 842 Research Project Report: Designing by Contract with JML for Aspect-Oriented Programs

Prepared By: Matthew Stephan
Prepared for: Professor Macdonald
Student ID: 20098161
Userid: mdstepha
4B Software Engineering
April 13, 2007

Table of Contents

Table of Figures

1. Motivation

Aspect-Oriented Programming (AOP) [1], the notion of programming by identifying and isolating cross-cutting concerns, is becoming extremely popular in both industrial and academic environments. It isolates these concerns by utilizing the concept of pointcuts and joint points, which identify both static and dynamic places that the concerns are applicable within the code being cross cut. One of the main advantages gained by using AOP is the idea of obliviousness, which entails the ability of the component encapsulating a cross-cutting concern's logic, known as an aspect, to be unknown from the perspective of the base code on which the aspect is being applied to. While this is clearly amicable because it involves a one sided dependency, it also makes testing increasingly difficult. Because of AOP's increasingly wide-spread adoption in industry and academia, especially the use of AspectJ [2], the issue of testing AOP programs is one of paramount importance. Even though [3] argues that it can be viewed as both harder and easier to test AOP systems, the easier-to-test side of the argument is dependent on AOP testing principles that are not yet solidified or well established.

1.1. Obliviousness and Pre and Post Conditions

This obliviousness property makes AOP very cumbersome to test, as noted in both [3] and [4]. One specific factor alluded to in both these papers that contribute to this difficulty in testing is the fact that aspects have the ability to break both preconditions and post conditions set forth by a method due to their ability to insert code, known as advice, before, after, and around a method call. [4] recommends dealing with this issue by testing the amalgamation of the base code and aspect with the original test data used to test the base code irrespective of any aspect interference. While this will likely identify many of the possible faults in the program, the preconditions of the base code method that are a function of the test data may be modified by the aspect. This original test data solution will identify only post conditions that have not been adhered to. Preconditions will not be verified because the original test data in conjunction with an aspect's advice may still allow for a successful test, that is, the satisfaction of post conditions.

Another issue that arises from the solution put forth is that localization of fault becomes very difficult to accomplish in the case where many aspects are being woven into a single method. This is because there is no easy way of identifying which aspect is at fault of breaking the pre or post condition that was violated. Using the original test data does not account for any of the specific aspects being applied, it only identifies faults on a system-wide level. It is possible to create test data that can isolate problems in specific aspects using the notion of integration testing and incrementally testing one aspect, but this would be a time consuming process. Ideally, the system should be able to run holistically with varying test data and still allow for the problem area(s) to be identified precisely.

1.2. Design by Contract and Java Modelling Language

One possible way of ensuring that pre and post conditions are not violated at any point during the runtime of an OOP program is to utilize the design by contract (DBC) [5] paradigm. This is a mode of specification that entails having each method within a program indicate explicitly, or as explicitly as possible, the pre and post conditions that must be adhered to during the execution of the method and that these conditions are verified dynamically. The original DBC works on the basis that every object in an Object-Oriented Programming (OOP) system enters into a binding “contract” with one another [5]. By operating on this principle, it is possible to detect condition violation during execution. It should, thus, be possible to extend this concept to aspects.

One implementation of DBC for Java programs that is on the rise, is the Java Modelling Language (JML) described in [6]. JML can accomplish DBC through the use of annotations at the source code level to indicate the preconditions and post conditions for a method by use of the “requires” and “ensures” keywords, respectively. It allows for both informal and formal specification of these conditions, however only the formal variety can be checked at runtime. As noted in the paper [6], it facilitates information hiding practices, non-null checking, and invariants at the class level. There are also a number of tools available that provide more features such as documentation, unit testing, and static checking.

Something important to keep in mind when discussing JML is when a condition or an invariant is broken, a runtime error occurs and a program ceases to execute. This is typically not the type of behaviour one desires in a system that has gone to production, rather errors should be handled in a more graceful manner. From a non-production/testing perspective, however, JML specification and verification could be utilized in conjunction with aspects for regression, unit, and other types of tests that can benefit from it.

1.3. Report Outline

The following report addresses the problem of testing preconditions and post conditions in an AOP system by considering a solution implemented with JML DBC that extends to aspects. It begins by discussing related work in Section 2 and continues in Section 3 by going through the proposed solution, that is, the way JML can be applied/extended to AOP. Section 4 then analyzes the existing JML tools with respect to the way they can be extended to assist with a JML AOP solution. Section 5 presents the possible problems that such a solution may encounter and the shortcomings that may arise. Section 6 follows with a presentation of future work. The paper finishes in Section 7 by drawing conclusions.

2. Related Work

This section presents related work to that of this report.

2.1. Design by Contract Solutions in AOP

It is important to make the distinction between the idea proposed in this paper and the concept of DBC being implemented using AOP. Solutions such as Contract4J [7] and the type of solution discussed in [11], are not DBC solutions for aspect programs but rather DBC implementations accomplished in AOP for use in Java programs. [11] is a patent that presents a high-level framework for solutions that will implement DBC using AOP.

Contract4J [7] is one such solution. It facilitates DBC by using AspectJ to deal with the cross-cutting concern of method specification. It allows for two different modes of method specification; annotations and method naming. The annotations option contains annotations of pre and post conditions that are in the form of boolean expressions that are evaluated at runtime.

Similar to JML, if these boolean expressions are not satisfied, the program throws a runtime error and stops executing. The naming/signature option uses a principle similar to JavaBeans [8], in which condition tests are inline instance methods that are evaluated against the expected conditions. Again, program execution is halted in the case of a violated condition. Another such solution is Moxa [9] that differs from Contract4J in that it is more focused on assertions that are cross cutting rather than on the individual method level. Both of these differ from this report in that they are implementations of DBC using AOP technologies.

2.2. Pipa

Pipa [10] is a JML extension that is made to specify AspectJ aspects. It adds a few language constructs, such as “proceeds” and “assignable”, facilitating specification of aspects similar to the way JML does for methods and classes. It also discusses the idea of translating a Pipa program into a JML one allowing for the JML toolset to be utilized. While this may appear similar to this report, there is an important difference. Pipa does not consider DBC in its specification approach. It is more concerned with specifying aspects alone and, most importantly, does not consider the idea of aspects breaking a base program’s preconditions or post conditions, that is, DBC at a higher level. While Pipa is useful for programs developed in a more aspect-aware fashion, this report’s solution is suited more for programs whose base code is truly oblivious and thus focuses at the compilation level on the contract between an advice’s pre and post condition and the pre and post condition of the base code being effected.

2.3. Cona

Cona [12] is mainly a solution like the ones described above that implement DBC using aspects, but tries to go one step farther and incorporate DBC for aspects as well. [12] provides an execution sequence and blame assignment evaluation that is used in this paper. Cona differs from the solution presented in this report, however, because, as described above, it uses aspects to solve the DBC problem on an OOP level while simultaneously tries to solve the DBC problem for aspects, a problem discovered/considered after the initial design and planning of their system. Furthermore, it differs from the solution put forth in this paper because Cona is an entirely standalone/fresh application while the solution proposed in this report is an extension to JML, an already established specification tool for Java programs. It also appears as if work on the Cona project is somewhat stagnate as indicated by [13], which says that the tool has been in its infancy since 2004.

3. Proposed Solution

This section discusses the idea of a JML extension that facilitates DBC specification for aspect-oriented programs. It ensures that both pre and post conditions are adhered to for both aspect and base code alike while maintaining the obliviousness characteristic of AOP. The section begins by providing some insight on the way the research was studied and the analysis was performed. The section then gives information on the specifications to be used within the solution and in what manner they will appear. It then continues by focusing on the case where an aspect is interacting with base code. Lastly, it handles the more complex case of multiple aspects applying to a single method, thus involving some aspect-to-aspect interaction. While no specific implementation is given, enough analysis and information is provided such that, given time, an implementation can be realized.

3.1. Method of Analysis/ Basis of Research

The following solution was derived by the author of this report by looking at the various related works and combining that with the knowledge acquired in class from the papers read. The specifications component came as a result of the author's experience with using JML and aspects, separately. The aspect specification was based on this, some work from [10] as noted in the section, and analysis performed by the author.

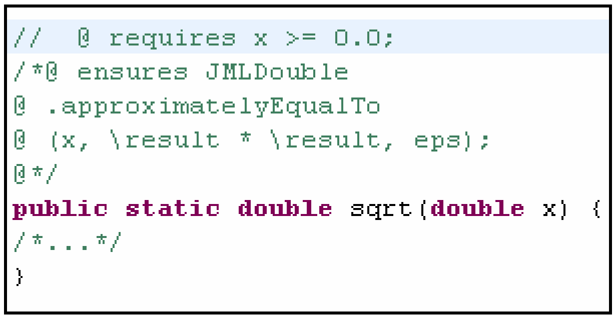
The section evaluating the aspect interaction is based on the author's familiarity with the workings of the AspectJ compilation and runtime semantics acquired from the papers presented in class. The author then performed analysis of the different requirements/implications of a compiler that would support AOP DBC.

3.2. Specifications

This section considers the specifications required to have a DBC version of AOP. It does not consider the runtime checks or behaviour of a compiled program; rather, it is only concerned with how to indicate the various contracts among the different pieces.

3.2.1. Base Code Specification

In order to adhere to the obliviousness constraint in AOP, the base code must be specified in a non-aspect-aware fashion. Given this restriction, there is really no reason why it is necessary to change any of the basic JML specification for standard java/base code. Method and class invariants can continue to cause runtime errors and be specified in the same fashion. As such, the only real consideration the compiler will have to account for, with regards to specification/syntax, is the specification of aspects. Figure 1, taken from [6], shows the typical JML DBC example of the square root function. Regardless of who calls this method, be it an aspect or a base-code method, the precondition is having a positive number and the post condition is having the correct result. Note the "\result" symbol which refers to the result of the method and the JMLDouble variable, which provides type checking, both of which are facilitated by [6].



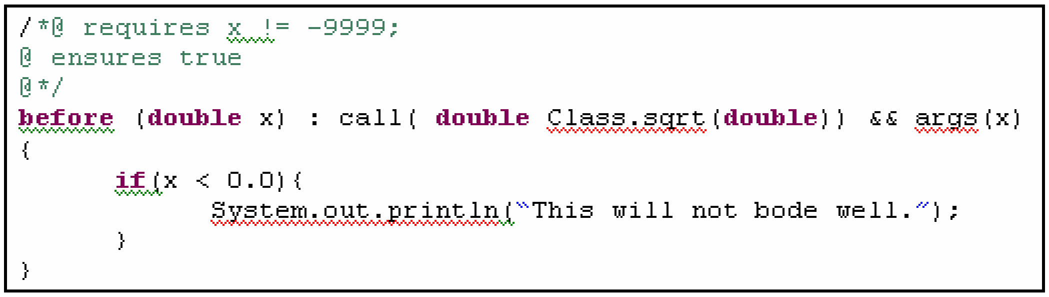
```
//  @ requires x >= 0.0;
/*@ ensures JMLDouble
@ .approximatelyEqualTo
@  (x, \result * \result, eps);
@*/
public static double sqrt(double x) {
/*...*/
}
```

Figure 1: Typical Base Code Example. Copied from [6]

While the example appears trivial, JML can handle many Java types. Also, informal specification allows non-action (documentation) handling of cases when conditions are beyond the scope of JML to formally specify. While this can not be run against during execution, the information specification will still provide valuable information to aspect clients who have the ability/option of looking at the method(s) they are affecting, which is not always the case.

3.2.2. Aspect Specification

At the specification/code level, aspects can be viewed in terms of the advice they supply. Each piece of advice is fundamentally the same as a method in the base code except for the fact that it is run in the context of before, after, or around a base-code method. As such, advice can be treated analogously to base-code methods in that advice can be viewed as a method that has pre and post conditions. Figure 2 showcases a simple before advice that utilizes the same syntax as JML. It continues from the square root example from earlier, outputting a message if the variable is less than 0. In this case, there are no post conditions, as indicated by syntax from [6], and the pre condition for the advice is the number must not be -9999, which may be a special indicator value.



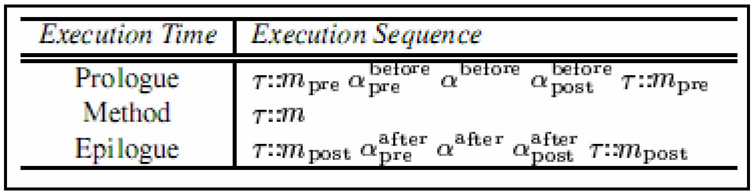
```
/*@ requires x != -9999;
@ ensures true
@*/
before (double x) : call( double Class.sqrt(double)) && args(x)
{
        if(x < 0.0){
                System.out.println("This will not bode well.");
        }
}
```

Figure 2: Aspect Specification Example

There is one important consideration that must be made however when dealing with around advice. This is the fact that around advice need not proceed into a method after executing or can proceed at any time. The way that this should be handled in this case is the same way around advice as dealt with in [10]. [10] handles this by means of adding a "proceeds" predicate that indicates what must be true during the execution of the around advice. It also uses a "then" keyword to indicate what happens before and after the "proceed" call is made. [10] contains a much more detailed description that can be referred to when necessary.

3.3. Aspect Interaction with Base Code

The meanings of the pre and post conditions at a high level vary depending on the type of advice. This section looks at the different types of advice and the way it interacts with the base code both conceptually and logically. Figure 3, taken from [12], provides the technical execution/placement of the pre and post conditions and corresponds with basically the same notion the author of this report came up with before reading [12] except the author of this report failed to consider the preconditions of the method being checked prior to the before advice preconditions. The [12] description of this process is strictly technical, quite brief, and does not provide any conceptual view of it, which are all important considerations, especially in the context of documentation, maintenance, and testing. It does, however, segregate the execution sequence into three useful categories; prologue, method, and epilogue. Alpha represents the advice and “m” represents the base-code method that the advice is being applied to. In the case of around advice, it can be split up into the before and after advices.



Execution Time	*Execution Sequence*
Prologue	$\tau::m_{pre}\ \alpha^{before}_{pre}\ \alpha^{before}\ \alpha^{before}_{post}\ \tau::m_{pre}$
Method	$\tau::m$
Epilogue	$\tau::m_{post}\ \alpha^{after}_{pre}\ \alpha^{after}\ \alpha^{after}_{post}\ \tau::m_{post}$

Figure 3: Execution Sequence. Taken from [12]

The prologue segment of time deals strictly with before advice. The precondition of before advice in the context of a single aspect should be consistent with the preconditions of the base-code method because it executes before it. As noted in the figure, since the call to the method knew not of the before advice, the first thing that should be checked are the preconditions of the method call. The advice should run as normal and then verified against the post conditions of the before advice. In this singular aspect case, these post conditions should be consistent with the preconditions of the base-code method.

After the base-code method executes, the post conditions for that method must be validated. This should be followed by the testing of the preconditions of the after advice method, which, in the case of a single aspect, should be consistent with the post conditions of the base-code method. After the execution of the after advice, the post conditions of the after advice should be checked to ensure it has fulfilled its contract. This should be followed by the rechecking of base-code method post conditions to make sure the after advice did not violate the method’s post conditions, which, as mentioned earlier, is a large issue in testing AOP systems and a reason that some people are hesitant to use AOP.

3.4. Aspect Interaction with Other Aspects

The notion of DBC with AOP becomes much more complex when multiple aspects are thrown into the mix. Rather than the relatively straightforward scenario presented in the previous section, multiple aspects entail a much more complex interaction that will likely be more difficult to codify in a compiler. Also, the notion of aspect precedence must be accounted for.

The way this could be handled is similar to the way advice is chosen to be ran in the first place. While this may not be entirely efficient, it is the most conceptually consistent with the notion of treating advice as simple methods. What this entails is each time advice is run before the base-code method, either as before or around advice, the pre and post conditions of each advice should be run before the next advice is ran. The same goes for after and after-around advice. An issue that arises here is the question of whether or not aspects should be oblivious with respect to one another. If so, they are not "entering contracts" with other aspects, so, why should they expect to bide by the preconditions of those other aspects. The counter to this argument is that this is the way that advice is run currently, at least in AspectJ, so even if there was no pre or post condition enforcement, these aspects would be running in this order. As such, the problem would still exist, but this would at least allow a tester or developer to determine which aspect interaction is causing the unexpected behaviour.

The author of this report found this solution of imitating/following the advice call stack preferable to the alternative that has all the preconditions of the before/after advice checked at once, followed by the execution of all advice, concluding with the post conditions of the before/after advice. This seemed conceptually incorrect because all of the advices' pre or post conditions would be seen the conjunction of each advice's pre or post conditions, respectively. As such, this would fundamentally translate to a single advice with a single set of preconditions and post conditions, making localization of fault more of challenge.

4. Extension of JML Tools

This section proposes possible extensions to the existing set of JML tools by analysing these tools and seeing the benefit they could provide AOP DBC. The extensions would be done with the intent to provide the same functionality, but more geared toward AOP DBC.

4.1. Unit Testing

The unit testing tool for JML unit testing [6] tests units on the basis of JML specifications rather than standard JUnit tests. A question that needs to be answered that was also raised in [3], however, is the notion of whether or not an Aspect can be considered an autonomous unit. If so, then unit testing is appropriate, if not, it may not be. In the case of the solution put forth in this report, it is technically feasible for the most part to accommodate this because advices are treated as methods, so they can be tested independently. The only place an issue may arise is the area of around advice, in which the “proceed” statement may cause different behaviour with regards to the specifications, and, thus, is dependent on base code.

4.2. Documentation

Documentation of AOP DBC code can be made possible by extending the jmldoc application alluded to in [6]. It performs relatively simple Javadoc transformations on base-code methods to retrieve Javadoc and JML specifications. Doing these transformations on AOP DBC would be analogous to the traditional JML and likely not very difficult. This would go quite a long way in furthering understanding of an AOP program. Since Javadoc can be transformed to hypertext mark-up language (HTML), a linked structure could be used that could show where statically-resolvable aspects are being applied within a program. Developers would be able to trace through the aspects that apply to join points within their base code and ensure that the post conditions (formal or informal) of the aspect align with the preconditions of their method. The inverse, aspect developers ensuring they are abiding by method preconditions, will now also be possible.

4.3. Static Specification Checking

In cases where join points can be statically resolved, static specification checking of aspects can be facilitated in the same way as the extended static checker does from [6]. This means that a large portion of AOP DBC can be statically checked/tested in instances where compilation is not required.   This may be somewhat infrequent though, due to the likelihood of dynamic join points, like Cflow, and the dynamic ordering of aspect advices in the case of multiple aspects acting on a single base-code method.

5. Limitations of the Solution

As noted in class, a solution combining two or more ideas not only yields the benefits of both, it also inherits the weaknesses of both unless otherwise counteracted.

In the case of this solution, all of the shortcomings associated with JML are inherited, some of which were described above. Firstly, the solution does not deal with all AOP languages, only AspectJ since that is what is supported by JML and considered in the report. If another language needs to be supported, than a base other JML would have to be used and, in this case, some of the solution dealing with implementation specifics would not be applicable. Currently, JML can only handle a few primitive types. Much like JML, the solution presented requires a specialized compiler in order for the specification annotations to be picked up and interpreted. Otherwise the program will run as it would without specification. Another problem is that JML relies solely on annotations, something that many people discourage/dislike using. Lastly, a consequence of using JML is the solution is intended only for testing/verification purposes due to the fact that JML throws only runtime errors. If a solution was desired that only reports specification violation without program termination, then a different base would need to be used.

The limitations that arise due to use of AspectJ with JML are not as bad as one would expect, mostly because AspectJ itself is an extension of Java much like the proposed solution/extension is to JML. Obviously, the complexity of comprehension and specification increases, especially in cases where multiple aspects are concerned. The definition of pre and post conditions becomes somewhat ambiguous when dealing with multiple advice applications, however, this may be a result of the lack of a formal way of dealing with advice ordering. This issue was raised in class as it was brought up in the [4] paper, leaving half the class content with the current system of aspects non-explicitly declaring aspect precedence and the other half wanting a more explicit precedence system.

6. Future Work

The most critical future work for this system is to implement a prototype of the proposed solution using the principles and ideas put forth in this report. The specification section in conjunction with work done in [10] and [12] should be more than enough to produce a formal specification and to begin from there. Basing the multiple-aspect interaction on the current way it is done in AspectJ significant lessens the problem of having to decide how to handle multiple advices applying to a single base-code method. From the analysis put forth in this report, it seems to be the ideal solution, at least for the AspectJ DBC

After the initial creation of the prototype, it is likely that optimizations can be made. As described earlier, there is going to be a significant amount of precondition and post condition checks going on. Each time a method is called, there is a check of the preconditions and post conditions for every before advice, every after advice, and for the base-code method as well. Given the already slow nature of JML, as learned from experience, this prototype will likely be very slow as well. Optimizations should be made and observations of these optimizations should be recorded and analyzed perhaps leading to a variation of the solution presented in this report.

7. Conclusion

This report determined the requirements and issues that will arise when attempting to create a JML DBC facility that works with/for aspects, specifically AspectJ. It was determined that very few specification keyword constructs are needed to be added to JML in order to facility specification of aspect pre and post conditions, specifically, it is only required in the case of around advice. Furthermore, it was determined that while multiple aspects applying to a base-code method may be difficult to accommodate from a DBC perspective, it is easier and conceptually correct to accomplish it by mimicking the advice call order done by default in AspectJ.

The pre-existing extensions/tools for JML DBC all seem relatively easy to extend yet all provide extreme benefits to AOP in terms of comprehensibility and testing. Unit testing can be accomplished, assuming it is deemed acceptable by the AOP society to test aspects as independent units. Documentation is hugely beneficial in that HTML documentation through Javadoc can provide traceability links bidirectionally between aspect code and base code. Static checking of aspects is also available in instances where join points can be statically resolved.

Although, it is not entirely clear how helpful static checking would be in the context of AOP system since many join points are dynamic.

The limitations of the work were also presented. The fact the solution is based on JML implies it receives all of its shortcomings. As such, the proposed AspectJ DBC system can formalize only primitive types and is required to specify via annotations. It also inherits the problems associated with aspects. One such problem, the order of checking pre and post condition of advice, is likely one that will be resolved when the issue of AspectJ aspect ordering is fully settled.

Future work is presented outlining the steps to take next. A prototype should be created so further analysis and tweaking can be performed. Soon after, optimization should occur in order to increase quality of the system and to obtain important observations about advice ordering and precedence as it relates to evaluation of pre and post conditions that may lead to more findings.

References

[1] Kiczales, G., et al. “Aspect-Oriented Programming”. *In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag, Finland (1997)

[2] Kiczales, G., et al. “An overview of AspectJ”. In*: ECOOP 2001: Object-Oriented Programming*, Lecture Notes in Computer Science 2072, 2001, pp. 327–355.

[3] Ceccato, M., et al. “Is AOP code easier or harder to test than OOP code?.” In: *Workshop on Testing Aspect-Oriented Programs*, 2005.

[4] Alexander, R., et al. “Towards the Systematic Testing of Aspect-Oriented Programs”. In: *Workshop on Testing Aspect-Oriented Programs,* 2005.

[5] Meyer, B. “Applying design by contract”. *Computer*, 25(10):40-51, October 1992.

[6] Leavens, G., and Cheon, Y. “Design by Contract with JML”. Internet: ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf , September 28, 2006. [Accessed April 12, 2007]

[7] Wampler, D. “Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces”. In: *Fifth AOSD Workshop on ACP4IS, Bonn, 2006.*

[8] Sun Developer Network. “JavaBeans”. Internet: http://java.sun.com/products/javabeans/index.jsp , 2007 [Accessed April 12, 2007]

[9] Yamada, K., et al. “An Aspect-Oriented Approach to Modular Behavioural Specification” *Submitted to ABMB 2005,* Internet: http://www.win.tue.nl/ABMB/3Yamada_Watanabe.pdf , 2005 [Accessed April 12, 2007]

[10] Zhao, J., et al. “Pipa: A Behavioral Interface Specification Language for AspectJ”, In: *Proc. Fundamental Approaches to Software Engineering*, 2003.

[11] Lopes, C., et al. “Design by contract with aspect-oriented programming”, U.S. Patent 6442750, Oct 22, 1999.

[12] Skotiniotis, T., et al. “Cona: Aspects for Contracts and Contracts for Aspects”, In: *OOPSLA'04,* Oct. 24-28, 2004, Vancouver, British Columbia.

[13] Skotiniotis, T. “Welcome to Cona”, Internet: http://www.ccs.neu.edu/home/skotthe/cona/ , 2004. [Accessed April 12, 2007]