

# Moxa: An Aspect-Oriented Approach to Modular Behavioral Specifications

Kiyoshi Yamada<sup>\*</sup>  
School of Information Science  
Japan Advanced Institute of Science and  
Technology  
kyamada@psg.cs.titech.ac.jp

Takuo Watanabe  
Programming Systems Group  
Department of Computer Science  
Tokyo Institute of Technology  
takuo@acm.org

## ABSTRACT

Design-by-Contract (DbC) is a software development method that utilizes assertions in a principled manner, and is beneficial for building reliable software systems. However, in our experience of applying DbC to the development of a working application, we faced a difficulty in dealing with assertions that have properties spanning over the natural program structure. Such crosscutting properties make each assertion bulky and thus can be obstacles for developing, maintaining or extending large-scale systems. Our solution to this problem is to introduce a new modularization mechanism based on assertion aspect, a new notion in aspect-oriented technology, to capture the crosscutting properties. We have designed a behavioral interface specification language Moxa that provides the mechanism. To examine our idea before developing Moxa tools, we have re-written the specification of our motivating application using AspectJ as a vehicle for prototyping modules for assertion aspects. Then we have compared it to our original, traditional DbC-based specification written in the Java Modeling Language (JML). The result shows that the use of assertion aspects clarifies the large, complex specification and greatly simplifies each assertion in the specification.

## Keywords

Design by Contract, Assertion Aspect, Java Modeling Language, AspectJ.

## 1. INTRODUCTION

An assertion is a programming language construct that specifies an assumption on the execution state at a certain program code position. Embedding assertions into the code

<sup>\*</sup>Contact address: Programming Systems Group, Department of Computer Science, Tokyo Institute of Technology, 2-12-1 Oookayama, Meguroku, Tokyo 152-8552, Japan

of a software module is a pragmatic method for testing, debugging and documentation. *Design by Contract* (DbC)[7] is a software developing method that utilizes assertions in a principled manner. In DbC, the “contract” between a class and its clients is a set of conditions (pre-/postconditions of the methods and a class invariant) typically represented as assertions embedded in the class code. The contract provides the detailed interface specification of the class.

DbC is especially beneficial for developing reliable software systems[7]. The authors have an experience of applying DbC to the actual development of a working application in which reliability is the prime factor to be considered. The application — AnZenMail client — is a secure and reliable e-mail client implemented in Java. It is a part of the AnZenMail system[8], an experimental testbed for cutting-edge security enhancement technologies. The AnZenMail system has been developed by a group of researchers involved in the research project “*Research on Implementation Schemes for Secure Software*” supported by Japanese Ministry of Education, Culture, Sports, Science and Technology.

The primary purpose of applying DbC was to ensure the code quality of the AnZenMail client. To ensure the code quality of the AnZenMail client, we first wrote a formal specification of its important component, called the Maildir Provider, that should handle received e-mails and mail folders in reliable way. We used the Java Modeling Language (JML)[5] to describe its specification with DbC-style assertions. With this specification, we checked the component thoroughly using the JML tools and then we could find bugs in the code and the assertions. This process, which was actually performed incrementally and repeatedly, enabled us to gradually obtain the solid code and the firm specification of the component. The final specification consists of approximately 3,500 lines of assertions.

While we were carrying out the above process, we often got the following problem: changes made to an assertion in a class caused the propagation of changes in the assertions within other classes. In principle, DbC assertions in a class are independent from ones in other classes. But in real life, while we were working with some large, seemingly unrelated classes, we often encountered the above phenomenon. This can be a serious obstacle for developing, maintaining or extending a large-scale software with DbC. We have observed that there are properties that span over the assertions in several program modules (classes or methods). The problem comes from the fact that the coverage of such properties

```

1 public class Foo {
2   /*@ public behavior
3     @ requires P;
4     @ ensures Q;
5     @ signals (FooException e) R(e);
6   @*/
7   public void foo() throws FooException {...}
8 }

```

Figure 1: JML Annotation

does not fit the inherent structure made from the program modules. In other words, they *crosscut* the modules.

To overcome the problem, we introduced a new modularization mechanism for assertions that aims to separate the crosscutting properties. The mechanism is based on *assertion aspect*, a new notion in aspect-oriented technology. So far, we have designed a new behavioral interface specification language *Moxa*, an extension of JML, that provides the mechanism.

Before developing Moxa tools, we have examined our idea by re-writing the specification of the Maildir Provider using AspectJ[4] as a vehicle for prototyping modules for assertion aspects. Then we have compared it to the original specification in JML. The result shows that the new modularization mechanism greatly simplifies the assertions in each program module by eliminating subexpressions commonly exist in the assertions.

The rest of the paper is organized as follows. The next section briefly introduces the Java Modeling Language. Section 3 explains the development of the Maildir Provider, our motivation example. In Section 4, we introduce assertion aspect and the language Moxa. Section 5 describes how to examine our idea using AspectJ and gives the comparison to the specification given in Section 3. Section 6 compares to the related work and gives the discussions. Section 7 concludes the paper.

## 2. JAVA MODELING LANGUAGE

The Java Modeling Language (JML)[5] is a behavioral interface specification language tailored to Java. JML supports DbC style assertions for describing behavioral specifications. Figure 1 shows a skeleton of the specification in JML. In JML, a specification consists of assertions written within special annotation comments that starts with “@” sign (see lines 2 to 6 in Figure 1). The keywords **requires**, **ensures** and **signals** are respectively used to specify the pre-condition, the (normal) post-condition and the exceptional post-condition of the method.

In order to describe assertion expressions in JML, we use Java boolean expressions enhanced with quantifiers and special expressions. For example, `\forall` and `\exists` represent  $\forall$  and  $\exists$  respectively. We can use the special expressions `\old(expression)` and `\result` in post-condition expressions. The former denotes the value of the *expression* evaluated just before the method execution, and the latter denotes the result of the method. We can also use method calls within assertions when such methods are declared as side-effect free using `pure` modifier.

After describing the assertions within a program code, we used the following JML tools to check if the code is correct. First, we compiled the code with **jmlc** to embed assertion

checking code into the target class files. Then, using **JUnit**, we ran extensive unit tests to detect assertion violations. To set up initial states for the unit testing with proper code coverage tracking, we used **coverage**.

## 3. THE MOTIVATION EXAMPLE

### 3.1 The Maildir Provider

As a part of AnZenMail client, we developed the *Maildir Folder Service Provider* (*Maildir Provider* for short). This is a JavaMail[9] component that manages *maildir* mailboxes on file systems. Maildir is the name of the mailbox format that is used in the *qmail*[1] mail server. It specifies the structure for directories of incoming e-mail messages and can provide reliable hierarchical mailboxes by using sophisticated algorithms for handling message files.

JavaMail API provides a platform-independent and protocol-independent framework for constructing e-mail or other messaging applications in Java. The API consists of two layers: an abstraction layer that provides classes and interfaces used by the applications, and an implementation layer that contains *service providers*. A service provider is a component (a set of classes) that provides the functionality of a particular protocol or message store. Because service providers are pluggable component, we can easily extend any JavaMail based applications by plugging new service providers. Sun distributes services providers for standard e-mail protocols such as SMTP, POP3 and IMAP with their reference implementation of the JavaMail API. The Maildir Provider is a service provider for maildir message stores (mailboxes). These service providers are plugged into the AnZenMail client.

### 3.2 Specifying the Maildir Provider

Because the Maildir Provider provides the functionality of managing local mailboxes, its reliability is essential to ensure the reliability of the entire application. To ensure the reliability of our implementation of the Maildir Provider, we will validate the following properties:

- (a) The implementation conforms to the interface and behavior defined in JavaMail API.
- (b) The directory structure of a mailbox managed by the implementation is always consistent.
- (c) Messages stored in a mailbox managed by the implementation should never be lost even when the application stops within the code of the implementation.

We specified these properties as JML assertions embedded in the source code of our Maildir Provider implementation. To describe the specification, we took the following approaches:

*Behavioral Subtype Relations (a):* The public classes in our Maildir Provider implementation are defined by inheriting the classes in the abstraction layer of JavaMail API. This obviously implies the correctness of the syntactic interfaces (aka type correctness). Thus we should only check that the behavior of our Maildir Provider implementation conforms the behavioral specification defined in the JavaMail. In other words, we should check that each public class in the implementation is the behavioral subtype[6] of its corresponding class (or interfaces) in the abstraction layer of the

```

1  ...
2  public abstract class Folder {
3  /*@ public behavior
4  @ requires
5  @   chkState_connected(...) && chkName(...) && ...;
6  @ ensures
7  @   chkState_eq(...) && chkName_eq(...) && ...
8  @   && chkResult_getMessageCount(\result);
9  @ also public behavior ...
10 @*/
11 public abstract /*@ pure @*/
12 int getMessageCount()
13 throws MessagingException;
14
15 /*@ public behavior
16 @ requires
17 @   chkState_open(...) && chkName(...) && ...;
18 @ ensures
19 @   chkState_eq(...) && chkName_eq(...) && ...
20 @   && chkResult_getMessage(\result);
21 @ also public behavior ...
22 @*/
23 public abstract /*@ pure @*/
24 Message getMessage(int msgnums)
25 throws MessagingException;
26
27 /*@ public behavior
28 @ requires
29 @   chkState_closed(...) && chkName(...) && ...;
30 @ ensures
31 @   chkState_open(...) && chkName_eq(...) && ...;
32 @ also public behavior ...
33 @*/
34 abstract void open(int mode)
35 throws MessagingException;
36
37 /*@ public behavior
38 @ requires
39 @   chkState_open(...) && chkName(...) && ...;
40 @ ensures
41 @   chkState_closed(...) && chkName_eq(...) && ...;
42 @ also public behavior ...
43 @*/
44 abstract void close(int mode)
45 throws MessagingException;
46 ...
47 }

```

Figure 2: Assertion Declarations in JML

JavaMail. Here, the behavioral subtype relation is a subtype relation where the instance of the super class in this relation can be replaced by the instance of its subclass safely. To validate this property, we write the behavior of the classes and interfaces defined by the abstract layer of JavaMail as the pre- and post-condition using JML.

*Consistency of Maildir Folders (b)(c)*. In our Maildir Provider implementation, the module-private class `MaildirManager` implements the operations on mailboxes and message files. The Maildir Provider always operates directories and message files in the file system through this class. Thus, to validate the properties (b) and (c) shown above, we only need to focus on this class. To validate the property (b) by focusing the class `MaildirManager`, we wrote post-conditions that represent the property: all the messages in the folder should not be affected, except for the messages handled the methods.

Before that, to make the validation of the property (c) easier, we made a simple program transformation on the methods of the class `MaildirManager`. The program transformation splits one complex method into multiple simple methods. After this transformation, each method satisfies the

property that the number of operations on the file system is at most one. Thus we can explicitly represent the inherent state transition caused by the invocation of the methods by using DbC style assertions. We validated the property that all the methods of the class `MaildirManager` do not corrupt or lost the messages (except for the method handling messages).

### 3.3 Validating the Implementation

We have tested our implementation using the JML tools described in Section 2. We could find some problems on the earlier versions of our Maildir Provider implementation. One of the problems is double escaping at the conversion between the URL name using to specify the folder location and the path showing actual folder location. Another problems is incorrect indexing of the messages in a maildir folder. Actually, we had been able to find the most of the problems in the earlier implementation at the specification phase. The unit testing could find a few, but hard-to-find, problems. The size of the final Java code of the Maildir Provider and the final JML specification (without the code) are 2,500 and 3,500 lines.

## 4. ASSERTION ASPECTS IN MOXA

### 4.1 Crosscutting Properties

In the specification described in the previous section, assertion expressions become complicated and bulky. This makes it difficult to develop the code and the specification incrementally with keeping the consistency of among assertions and code. Moreover, it becomes difficult to synchronize modification between a method and corresponding assertions. The source of these problems is the mismatch of modularization structures between the assertions and the code. In JML, we write assertions as annotations of methods. This forces that assertions are grouped into program modules. But this is not always appropriate for the modularization of assertions.

Generally, assertion descriptions in JML have some characteristics as follows. In Figure 2, showing a simplified assertion description example in JML for the class `Folder` defined in the JavaMail API, the logical formulas of assertion declarations consists of the logical products of the conditions that describe the independent aspect of the object such as the state of the object (shown in the figure as predicate `chkState*(...)`), the name of the object (predicate `chkName*(...)`), the return value of the method (predicate `chkResult*(...)`), and others. Moreover, such a conditions is appeared in the logical formulas of others, (for example, `Message Folder.getMessage(int msgnum)`, `void Folder.open(int)`, `void close(int)`, and so on). So we can say that some properties in the assertions are crosscutting over the assertion descriptions.

### 4.2 Assertion Aspects

Aspect-oriented Programming (AOP) is a methodology to improve the modularity of program code. Some kind of code fragments, such as logging or performance optimization, are scattered over some program modules and it is hard to extract them as independent modules. Such a notion is called crosscutting concerns. AOP Language provides a mechanism to modularize them into aspects. The AOP language AspectJ has the notions such as pointcut and advice. A

```

1 public spec S {
2   /*@ public behavior
3     @ requires pre;
4     @ ensures post;
5   @*/
6   T C1.m1(T1 arg1, ...);
7   T C2.m2(T2 arg2, ...);
8   ...
9 }

```

Figure 3: Assertion Aspect in Moxa

pointcut is a set of join points that are particular locations on the control flow of the program. An advice is a pair of pointcut and a code fragment executed at the location selected by the pointcut. An aspect is a set of advices.

We call the location on the control flow where we want to test the pre- or post-condition of the constructors or the methods, *pre-* and *post-condition location* respectively and we call them *assertion locations*. Also, we call the set of assertions an *assertion aspect* to avoid confusion with the original notion of aspect.

**Assertion Aspect.** The aspect in Moxa (called Assertion aspect) is a set of advice (namely descriptions of assertions). Figure 3 show the definition of assertion aspect **S** that has a description of advices (lines 2–8). An assertion aspect groups some advices and makes them a module.

In the ordinary assertion declaration technique or languages such as JML, we must keep the structure of the program for the description of the assertions. On the other hand, in Moxa, we can modularize advice descriptions independent of the program structure.

**Pointcut.** The pointcut is a set of join points that are locations on the control flow of a program. Because the assertion description style in Moxa is based on DbC, the notion of join point is just identical to the assertion location. The description of pointcuts (lines 6–7) consist of a set of method signatures and the keywords **requires**, or **ensures** (lines 3 and 4). The figure describes two pointcuts at once that show the pre-condition location of method **m1** and **m2**, and the post-condition location of these methods.

A join point in Moxa corresponds to a location in the ordinary assertion declaration technique where the assertion declaration is inserted. In the ordinary assertion declaration technique, when we want to describe same assertion in two or more assertion locations, we have to describe assertions for each of those assertions locations. On the other hand, in Moxa, we can describe the condition of these assertions only once by an advice whose pointcut selects these assertion locations.

**Advice.** The advice is a pair of a pointcut and a assertion condition. This corresponds to one or more description of assertions in the ordinary assertion declaration technique. The assertion condition is verified when the control flow has reach the location selected by the pointcut.

In Figure 3, lines 2–8 have a description of advices. This description of advices defines two advices because this description of advices selects two pointcuts as we mentioned above. One of the advice show that the pre-condition of the methods **m1** and **m2** are **pre**, and the other one show that the post-condition of these methods are **post**.

```

1 public aspect S {
2   private Tx1 old_x1;
3   ...
4
5   before(C1 this_, T1 arg1, ...)
6     : (call(T C1.m(T1, ...))
7       || call(...) || ...)
8     && args(arg1, ...) && target(this_) {
9     assert pre(this_, arg1, ...);
10  }
11
12  pointcut sig1(C1 this_, T1 arg1, ...)
13    : (execution(T C1.m(T1, ...))
14      || execution(...) || ...)
15    && args(arg1, ...) && this(this_);
16
17  before(C1 this_, T1 arg1, ...)
18    : sig1(this_, arg1, ...) {
19    old_x1 = ...;
20    ...
21  }
22
23  after(C1 this_, T1 arg1, ...)
24    : sig1(this_, arg1, ...) {
25    assert post(this_, args1, ..., old_x1, ...);
26  }
27  ...
28 }

```

Figure 4: AspectJ Representation of Figure 3 (part)

## 5. EXPERIMENT

### 5.1 Assertion Aspects in AspectJ

Before implementing Moxa tools, we examine our idea using AspectJ. We use AspectJ to prototype modules for assertion aspects. For this experiment, we have defined a translation rule from assertion aspects in Moxa to (code) aspects in AspectJ that embed assertion checking code into Java program. This rule corresponds to an assertion aspect compiler (such as `jmlc` in JML tools) that generates runtime assertion checking code. In the next subsection, we evaluate the specification description using this translation rule. Figure 4 is the translation result (code aspects in AspectJ) for the assertion aspect in Figure 3.

### 5.2 The Maildir Provider Revisited

In this subsection, we compare the specifications in JML and Moxa. Since we haven't implemented any Moxa tools, the latter is actually an AspectJ program using the AspectJ representation of assertion aspects described in the previous subsection. The common target of the specifications is the Maildir Provider; we compared the specification of its classes defined in the abstract layer of JavaMail (**Store**, and its super class, **Service**). The items of comparison are count of modules (count of classes in the case of JML, and count of aspects in the case of AspectJ), count of assertions (count of pre- and post condition in the case of JML, and count of advice in the case of AspectJ), and count of lines (comments included). The result of comparison is shown in Table 1, and its characteristics are described below.

**Number of Modules.** In the case of JML, the number of modules for each class is 1 because modularization unit of JML must be matched to the class or interfaces. In the case of AspectJ translated from Moxa, the counts of modules are 6 and 12 for the class **Service** and **Store**, respectively. This is because, each crosscutting conditions of assertion can be

**Table 1: Comparison of the Two Specifications**

	JML		Moxa*	
	Service	Store	Service	Store
# of Modules	1	1	6	12
# of Assertions	42	53	18	14
# of Lines	190	149	(229)	(570)
# of Lines / Module	190	149	38	48

\* using AspectJ representation of Assertion Aspects

split into different assertion aspects.

*Number of Assertions.* In the case of JML, the counts of assertions are 42 and 53 for the class `Service` and `Store`, respectively. In the case of AspectJ, the count of assertions are 18 and 14 for the class `Service` and `Store`, respectively, and each number is less than the case of JML. This is because, crosscutting conditions over the assertions includes same logical expressions, and they can be organized into an advice in Moxa.

*Number of Lines in Assertions.* The number of lines in assertion descriptions in JML are 190 and 149 for the class `Service` and `Store` respectively. On the other hand, the number of lines in aspects in AspectJ translated from Moxa are 229 and 570, for the class `Service` and `Store` respectively. The result shows that lines count of JML descriptions are shorter than that of AspectJ. But, the value of lines count of assertions per modules count, namely, the average lines counts of modules JML descriptions are 190 lines and 149 lines for the classes `Service`, and `Store`, respectively. And that of AspectJ are 38 lines and 48 lines for the class `Service`, and `Store`, respectively. This shows that the number of lines per modules of JML descriptions are larger than that of AspectJ, on the contrary.

Here, the total number of lines in Moxa is larger than the number of lines in JML. The reason is that we are currently use the AspectJ prototyping of assertion aspects described above. On the other hand, average number of line counts in the description on the AspectJ translated from Moxa become smaller than that of JML. This comes from the fact that same logical expression of assertions for some join points are merged into one advice in Moxa using pointcuts.

The above result shows that if we implement the Moxa compiler, we can expect that average number of lines per module will be extremely reduced. We can also expect that this can clarify large and complex specifications by modularizing crosscutting properties span over the program modules.

## 6. DISCUSSION

### 6.1 Related Work

Ishio et al.[3] and Diotalevi[2] individually propose the way to describe assertions using AspectJ. They point out the problems of embedding assertion descriptions in the program code and propose the ways to describe assertions as AspectJ aspects separately from program code. Our proposal not only enables assertions described as aspects separately from the programs similar to their way, but also we focus on the crosscutting properties over the assertions and

modularize it by aspect-oriented technology. So the method we propose makes assertion description more compact and simpler than their way.

Pipa[10] is an extension of JML whose target language is AspectJ. With this specification language, we can describe assertions for the constructs of AspectJ such as advice or introduction. Also in this language, assertions must be modularized by the unit of class or aspects of target language, so it is hard to deal with large-scale or complex assertions in this language. We consider that extension of join points of Moxa to deal with advice or introduction of AspectJ can apply to Pipa, and this enable assertion descriptions divided into some assertion aspects.

### 6.2 Modularization of Assertions

The simple assertion description technique for object-oriented programming language based on DbC such as JML has no mechanisms to control the mapping between assertions and methods. So, specifying pre- and post-conditions are permitted at most once a method, and they must be modularized by the unit of classes. On the other hand, Moxa enables us to describe assertions independently of the program structure considering assertion assignment location consists of a class, a method, and pre- or post-condition locations as pointcut and assertion description as advice. In the technique, for example, following style of assertion declarations are permitted.

- Specifying assertions to a class from one or more assertion aspects.
- Specifying one or more assertions to an assertion location (logical expression of these assertions are associated with logical product).
- Specifying assertions to one or more classes from one assertion aspect.

Using Moxa, we can split the behavior of object or object group into several independent sides, and we can describe each side of behavior into separated assertion aspects. This feature holds the scale and complexity of assertion aspects small. Moreover, the viewpoint of each assertion aspects becomes narrowed to some simple side, hence, expressing and understanding the meaning of an assertion aspect becomes easy. Also, the maintainability and quality of assertion aspects and corresponding programs are improved.

In Moxa, we can describe JML annotations along with assertion aspects, because Moxa is an extension of JML. Therefore, Moxa enables us not only to modularize assertions as assertion aspects independent of the programs structure, but also to specify assertions as annotations embedded into the program. Such a feature is favorable for the incremental development. Concretely, we can specify assertions using annotations for the program code at the early stage of development or modified rapidly. Then, the code becoming stable and crosscutting properties are unveiled, we can extract assertion aspects from annotations.

## 7. CONCLUDING REMARKS

This paper presented the notion of assertion aspects and a new behavioral interface specification language Moxa that provides a modularization mechanism for assertions. The mechanism enables us to separate crosscutting properties

spanning over multiple assertions. It can clarify a large, complex specification and also can greatly simplify the assertions in the specification by eliminating common logical subexpressions. Assertion aspect broadens the scope of AOP technology by providing the separation of specification concerns, instead of code concerns.

## 8. ACKNOWLEDGMENTS

This research has been supported in part by Japanese Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Scientific Research on Priority Areas (*Research on Implementation Scheme for Secure Computing*), No. 12133207 and Grant-in-Aid for Scientific Research (C), No. 15500028.

## 9. REFERENCES

- [1] D. J. Bernstein. qmail: Second most popular MTA on the internet. <http://www.qmail.org>.
- [2] Filippo Diotalevi. Contract enforcement with AOP: Apply design by contract to Java software development with AspectJ. IBM developerWorks, July 2004. <http://www-106.ibm.com/developerworks/library/j-ceaop>.
- [3] Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Assertion with aspect. In *International Workshop on Software Engineering Properties for Aspect Technologies (SPLAT2004)*, March 2004.
- [4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP 2001: Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355, 2001.
- [5] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and William Harvey, editors, *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175–188. Kluwer Academic Press, 1999.
- [6] Barbara Liskov and Jannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.
- [7] Bertrand Meyer. Applying “Design by contract”. *IEEE Computer*, 25(10):40–51, October 1992.
- [8] Etsuya Shibayama, Shigeki Hagihara, Naoki Kobayashi, Shin-ya Nishizaki, Kenjiro Taura, and Takuo Watanabe. AnZenMail: A secure and certified e-mail system. In *Software Security: Theories and Systems*, volume 2609 of *Lecture Notes in Computer Science*, pages 201–216. Springer-Verlag, March 2003. <http://anzen.is.titech.ac.jp/anzenmail>.
- [9] Sun Microsystems Inc. JavaMail API. <http://java.sun.com/products/javamail>.
- [10] Jianjun Zhao and Martin Rinard. Pipa: A behavioral interface specification language for AspectJ. In *Fundamental Approach to Software Engineering (FASE 2003)*, volume 2621 of *Lecture Notes in Computer Science*, pages 150–165. Springer-Verlag, April 2003.