An Aspect-Oriented Approach to Modular Behavioral Specification

Kiyoshi Yamada
Research Center for Information Security
National Institute of Advanced Industrial Science and Technology
Tokyo, Japan

Takuo Watanabe
Department of Computer Science
Tokyo Institute of Technology
Tokyo, Japan

Abstract

Behavioral interface specification languages, such as Java Modeling Language (JML), can be used to specify the behavior of program modules. We have developed a behavioral interface specification language Moxa, an extension of JML. Moxa provides a new modularization mechanism called assertion aspect that can capture the crosscutting properties among assertions. In this paper, we briefly explain the notion of assertion aspects and the design of Moxa, and then we show an example specification. By comparing the specification to its JML counterpart, we show 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 of a software module is a pragmatic method for testing, debugging and documentation. Design by Contract (DbC) [10] is a software development 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 source code. The contract provides the detailed interface specification of the class.
DbC is especially beneficial for developing reliable software systems [10]. The authors have experience in 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 [11], 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 a reliable way. We used the Java Modeling Language (JML) [7] 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 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 observed 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 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 [5] 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 that commonly exist in the assertions.

The rest of the paper is organized as follows. The next section explains the development of the Maildir Provider — our motivating example — and its specification in JML. In Section 3, we introduce the notion of assertion aspect and our
behavioral specification language Moxa. Then, in Section 4, we compare Moxa to JML by using the same example. Section 5 mentions the related work. Section 6 offers a discussion of the results. We conclude in Section 7.

2 The Motivating Example

2.1 The Maildir Provider

As a part of the AnZenMail client (mentioned in the previous section), we developed the Maildir Folder Service Provider (Maildir Provider for short). This is a JavaMail [12] component that manages maildir style mailboxes on file systems. Maildir is the name of the mailbox format that is used in the qmail [2] 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.

2.2 Specifying the Maildir Provider using JML

We used the Java Modeling Language (JML) [7] to describe the specification of the Maildir Provider. JML is a behavioral interface specification language tailored to Java. JML supports DbC style assertions for describing behavioral specifications.

Figure 1 is an abridged JML specification of the class Folder. This class is a part of the Maildir Provider. In JML, a specification consists of assertions written within special annotation comments that starts with the “@” sign. 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.

2.3 Properties to be Validated

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:

(i) The implementation conforms to the interface and behavior defined in JavaMail
Fig. 1. Specification in JML

API.

(ii) The directory structure of a mailbox managed by the implementation is always consistent.

(iii) 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 (for (i))**: 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 to the behavioral specification defined in JavaMail. In other words, we should check that each public class in the implementation is the behavioral subtype [9] of its corresponding class (or interfaces) in the abstraction layer of 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 (for (ii) and (iii)): 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 (ii) and (iii) shown above, we only need to focus on this class. To validate the property (ii) by focusing on 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 (iii) 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 lose the messages (except for the method handling messages).

2.4 Validating the Implementation

We have tested our implementation using the JML tools. 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 used to specify the folder location and the path showing actual folder location. Another problem is incorrect indexing of the messages in a maildir folder. Actually, we had been able to find most of the problems in the earlier implementation at the specification phase. 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.

3 Assertion Aspects in Moxa

3.1 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 associated to classes and methods. This forces that assertions are grouped into classes. But this is not always appropriate for the modularization of assertions. Figure 1 (in Section 2.2) exemplifies the problem.
public spec S {
  public behavior
  requires Pre1;
  ensures Post1;
  Ta C1.m1(T1 x1, ...);
  Tb C2.m2(T2 x2, ...);
  ...

  public behavior
  requires Pre2;
  ensures Post2;
  Tc C3.m3(T3 x3, ...);
  Td C4.m4(T4 x4, ...);
}

Fig. 2. An Assertion Aspect in Moxa

In this example, the logical formula for each assertion is the product of conditions concerning (1) the states of an instance of this class (chkState*), (2) the names of the instance (chkName*), (3) return values (chkResult*) and so on. Here, we can see that these conditions appear in all methods; they are crosscutting over the methods.

3.2 Aspects in AspectJ

Aspect-oriented Programming (AOP) [6] is a programming technique for modularizing concerns that cross-cut the modules in programs. Some kind of code fragments related to concerns such as logging, synchronization, exception handling or performance optimization, are mingled within functional modules. In other words, they cross-cut the modules. AspectJ [5] is an extension of Java that provides a mechanism for modularizing such tangled code. The key notions of the mechanism are pointcut and advice. A 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 consists of a set of advice.

3.3 Assertion Aspects in Moxa

The notion of aspect in Moxa is different from the one in AspectJ. The difference is that an aspect in Moxa is applied to specifications (logical expressions written as annotations), while an aspect in AspectJ is applied to code. We call aspects in Moxa assertion aspects to avoid confusion with aspects in AspectJ.

Figure 2 shows that how an assertion aspect is defined. In this definition, S is the name of this assertion aspect, C1⋯C4 are class names, m1⋯m4 are method names, x1⋯x4 are identifiers (arguments) and Ta⋯Td, T1⋯T4 are type descriptors. Pre1 and Pre2 (Post1 and Post2) are pre-conditions (post-conditions) respectively.

An assertion aspect is a collection of advice (as in AspectJ). Figure 2 has two
public spec FolderState {
  public behavior
    requires chkState_connected(...)
    ensures chkState_eq(...)
  public int Folder.getMessageCount() 
    throws MessagingException;
  public behavior
    requires chkState_open(...)
    ensures chkState_eq(...)
  public Message Folder.getMessage(int msgnums)
    throws MessagingException;
  public behavior
    requires chkState_closed(...)
    ensures chkState_open(...)
  void Folder.open(*) throws MessagingException;
  public behavior
    requires chkState_open(...) 
    ensures chkState_closed(...) 
  void Folder.close(*) throws MessagingException;
... 
}

Fig. 3. An Assertion Aspect Specifying State Transition of Folders (abridged)

advice: lines 2–6 and lines 8–12.

The advice is a pair of a pointcut and an assertion condition. The pointcut is a set of join points that are locations on the control flow of a program. 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. Because the assertion in Moxa is based on DbC, a join point is normally identical to the assertion location. A descriptions of pointcuts (e.g., lines 5–6) consists of a set of method signatures and positional keywords requires (or ensures). The first advice (lines 2–6) in Figure 2 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 the 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.

3.4 Example

Figure 3 is an assertion aspects that specifies the state transition of the class Folder (described in Figure 1). This assertion aspect captures and modularizes a concern (on state) on folders. In this example, the logical expression in each pre-/post-condition consists of the invocation of a method such as chkState_open. These methods are defined in actual classes (thus they are implementation dependent) and provide actual state information. This makes the assertion aspect FolderState implementation independent.
4 Specifying Maildir Provider in Moxa

In this section, we compare Moxa to JML by using the same example. The target of the specifications is a part of the Maildir Provider; we compared the specifications of its classes defined in the abstract layer of JavaMail (Store, and its super class, Service). The items of comparison are the number of modules (the number of classes in JML and the number of assertion aspects in Moxa), the number of assertions (the number of pre- and post conditions in JML and the number of advice in Moxa), and the number 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 a modularization unit of JML must be matched to the class or interfaces. In the case of Moxa, the number of modules are 3 and 5 for the class Service and Store, respectively. This is because, each crosscutting condition of assertion can be split into different assertion aspects.

**Number of Assertions:** In the case of JML, the number of assertions are 42 and 53 for the class Service and Store, respectively. In the case of Moxa, the number of assertions are 13 and 18 for the class Service and Store, respectively, and each number is smaller than the case of JML. This is because crosscutting conditions over the assertions includes the 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 total number of lines in assertion aspects of the Moxa specification are 152 and 286, for the class Service and Store respectively. Thus, we can see that the average number of lines in an assertion aspect is much smaller than the average number of lines in the JML specification. This comes from the fact that the same logical expression of assertions for some join points are merged into one advice in Moxa using pointcuts.

This result shows that using Moxa, the size of each module in a specification will be reduced. We can also expect that this can clarify large and complex specifications by modularizing crosscutting properties that span over the program modules.

**Locality of Changes:** Table 2 shows the effect of a simple change in the code. Here, we replace the method boolean Service.isConnected() to boolean Service.notConnected(). The table summarizes the effect of this change on the specifications: the number of the modules (classes in JML and assertion aspects in Moxa) we should fix and the number of lines possibly to be affected. In the Moxa specification of the class Service (Store), we should only change 6 (4) modules. Please note that we don’t need to examine the rest of the modules. The number of assertions and the number of lines to be changed dramatically decreases, because of aspect-orientation. This result shows that Moxa provides higher locality in specification.
Table 1
Comparison of the Two Specifications

<table>
<thead>
<tr>
<th></th>
<th>JML</th>
<th>Moxa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Store</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td># of Modules</td>
<td>190</td>
<td>149</td>
</tr>
<tr>
<td># of Assertions</td>
<td>42</td>
<td>53</td>
</tr>
<tr>
<td># of Lines</td>
<td>152</td>
<td>286</td>
</tr>
<tr>
<td># of Lines / Module</td>
<td>51</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 2
Number of Changes in the Specifications

<table>
<thead>
<tr>
<th></th>
<th>JML</th>
<th>Moxa</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service</td>
<td>42</td>
<td>6</td>
</tr>
<tr>
<td>Store</td>
<td>53</td>
<td>4</td>
</tr>
<tr>
<td># of Changes</td>
<td>190</td>
<td>54</td>
</tr>
<tr>
<td># of Lines in Changes</td>
<td>149</td>
<td>40</td>
</tr>
</tbody>
</table>

5 Related Work

Injecting assertion validation code into application modules is a typical application of AOP. There have already been several proposals on describing assertions using AspectJ [8,3,4]. They point out the problems of embedding assertions in the program code and propose ways to describe assertions separately from program code. Especially, Lippert and Lopes [8] investigate that global properties on exception detection and handling can be systematically represented using AspectJ.

Though writing validation code in AspectJ is one possible way to modularize assertions, it is generally complex and error prone task. Moreover, this style of assertion description is specialized to runtime validation. This means that using assertions with other analysis/verification tools is difficult.

Since Moxa has a dedicated syntax, specifications written in this language can be used not only for runtime validation, but also with other tools. Currently we are implementing Moxa processor as a translator to JML. Thus, it is possible to use existing JML tools.

Contract4J [1] is another tool that supports DbC in Java. This tool provides annotation based syntax for assertions and uses AspectJ for injecting validation code.

Pipa [14] is an extension of JML whose target language is AspectJ. With this language, we can describe assertions for the AspectJ constructs such as advice or introduction. However, as in JML, assertions in Pipa are modularized within target language (AspectJ) modules; i.e., classes or aspects. This means that Pipa does not provide modularization of crosscutting properties. Extending Moxa to support
AOP languages is future work.

6 Discussion

6.1 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, the 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 aspect 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.

6.2 From Incremental Refinement to Model-Driven Development

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 in-place 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. This process can be used for incremental refinement of existing code.

For example, suppose that we can extract an assertion aspect (say \(A_1\)) from a specification of an existing system. And suppose that \(A_1\) captures the state transition of modules in the system (as in Figure 3) If \(A_1\) can be refined to \(A_2\) that represents a more reliable state model \(^3\), the we can re-apply \(A_2\) to the original code.

\(^3\) Here, the term model denotes the notion in MDD.
and validate it to refine the code itself. This process can gradually improves the reliability of existing code.

Moreover, assertion aspects may represent other models. A sort of model-driven development (as in [13]) might be possible by using appropriate tools that generate a code skeleton from an assertion aspect. We need more investigation towards this direction.

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 by providing the separation of specification concerns, instead of code concerns.

8 Acknowledgments

We would like to thank the anonymous reviewers for their corrections and suggestions. This research has been supported in part by Japanese Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Scientific Research (C), No. 1750017.

References


