Aspect-Oriented Modularization of Assertion Crosscutting Objects

Takashi Ishio[1], Toshihiro Kamiya[2], Shinji Kusumoto[1] and Katsuro Inoue[1]

[1]Graduate School of
Information Science and Technology,
Osaka University
1-3 Machikaneyama, Toyonaka,
Osaka 560-8531, Japan
{t-isio, kusumoto, inoue}@ist.osaka-u.ac.jp

[2]Ubiquitous Software Group,
Information Technology Research Institute,
National Institute of
Advanced Industrial Science and Technology
Akihabara Dai Bldg. 1-18-13 Sotokanda,
Chiyoda-ku, Tokyo, 101-0021, Japan
t-kamiya@aist.go.jp

Abstract

Assertion checking is a powerful tool to detect software faults during debugging, testing and maintenance. Although assertion documents the behavior of one component, it is hard to document relations and interactions among several objects since such assertion statements are spread across the modules. Therefore, we propose to modularize such assertion as an aspect in order to improve software maintainability. In this paper, taking Observer pattern as an example, we point out that some assertions tend to be croscutting, and propose a modulalization of such assertion with aspect-oriented language. We show a limitation of traditional assertion and effectiveness of assertion aspect through the case study, and discuss various situations to which assertion aspects are applicable.

1. Introduction

Design by Contract[23] provides behavioral specifications including preconditions, postconditions and invariants to improve robustness of software. Preconditions protect the called component from illegal calls, and postconditions protect the caller against erroneous implementations, respectively[26]. However, they are hard to handle properties held in interactions among objects because traditional assertions specify behaviors of one object used by arbitrary clients.

We show one variant of Observer pattern[6] for an example. In the Observer pattern, observers and subjects are modeled as many-to-many relationship, in other words, a number of observers may observe one subject and an observer may observe several subjects. After a developer has implemented the observer pattern, the developer may reuse the many-to-many relationship code for one subject-to-many observers relationship because many-to-many implementation covers one-to-many usage. Although the developer may add assertions to observers and subjects in order to prohibit attaching an observer to several subjects, the assertions in the observers and the subjects strongly depend on each other.

This kind of assertions crosscutting objects is caused when a developer assumes some interaction patterns among the objects. Assertion specifying interactions is a promising tool for software maintenance since the behavior out of the interaction patterns expected by the developer may indicate a defect[4]. To use assertions effectively, we need a method to write assertions in a well-modularized manner since assertions crosscutting objects are harmful to the maintainability of software.

We propose to modularize assertions crosscutting objects as an aspect using an aspect-oriented language. In order to show the effect of modularization, we have defined a simple language whose pointcuts are a subset of AspectJ and compared two versions of the observer pattern with the one-to-many constraint in Java and in our language. As a result, modularized assertion simplifies objects and improves the maintainability of the objects.

This paper consists of six sections. In the next section, we present the background of the research. Section 3 describes our proposal modularizing crosscutting assertions to aspects. In Section 4, we show the difference between our approach and traditional assertion. In Section 5 we discuss software quality affected by our approach, situations to which our approach is applicable and related work. We draw conclusions in Section 6.

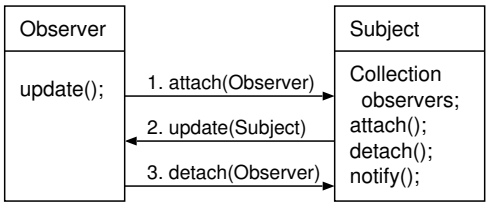

Figure 1. Observer Pattern

2. Motivation

Design by Contract[23] improves robustness of software by specifying the behavior of a component based on preconditions and postconditions for each method of the component. Preconditions protect the called component from illegal calls, and postconditions protect the caller against erroneous implementations, respectively[26].

Practical programming languages such as Java and C++ have `assert` as a language construct, a function of the standard library, or a macro of a preprocessor. The behavior of `assert(expr)` statement is shown as follows.

assert(true) → do nothing
assert(false) → throw a runtime exception

Preconditions and postconditions of a method are regarded as `assert` statements inserted into the beginning of the method and the end of the method, respectively.

Assertion checking is powerful, practical, scalable and simple to use. Assertion is effective to detect software faults during debugging, testing and maintenance[25]. Assertion supports developers in understanding the software because it documents the behaviors of a component and effectively prevents developers from depending on implementation details of the component[21].

Several behavioral specification languages and tools including JML[13], jContractor[15], Larch[8] and Contract4J[3] are proposed to use assertion effectively. They provide several convenient functions and predicate to improve expressiveness of assertions. A developer describes properties for each method of a component using these languages. Gibbs et al. have proposed *Temporal Invariants*, or an extension of assertion for temporal properties held in a series of method calls for one component[7]. On the other hand, Yamada et al. have proposed *Moxa*, or an aspect-oriented extension of JML[27]. Moxa provides language constructs to write common properties to several methods and classes. These approaches extend assertions to describe a property related to several methods. However, they are hard to handle properties held in interaction among objects because traditional assertions specify behaviors of only one object used by arbitrary callers.

We show a variant of Observer pattern[6] for an example. Observer pattern is an interaction pattern between Observers and Subjects. Figure 1 shows the structure of the pattern. `Subject` represents an object which has some data, and `Observer` represents an object watching subjects. An observer first attaches itself to a subject by calling the `attach` method. When the state of a subject is updated, the subject notifies the attached observers by calling `update` method of them. The notified observers call some methods of the subject to get updated information. An observer calls the `detach` method of a subject when the observer no longer need notification message from the subject. Observers and subjects are modeled as many-to-many relationship in the pattern. In other words, a number of observers may observe one subject and an observer may observe several subjects.

The assertion is hard to handle inter-object properties since the traditional assertion is described for each class. We discuss a variant of the observer pattern, a model of one subject-to-many observers relationship in order to show a limitation of the traditional assertion. After a developer has implemented an instance of the usual observer pattern, the developer can reuse the many-to-many relationship code for one-to-many relationship since many-to-many implementation covers one-to-many usage. The developer may want to add assertions in order to prevent an observer from being attached to several subjects. However, it is a hard task for the developer to describe the assertion for that purpose in a modularized manner since an observer has no variable which represents how many subjects the observer attaches to. So a developer need to add a field containing an attached subject to Observer and modify Subject to check and update the field when an observer is attached. The scattered code damages modularity and maintainability of the components. We show detailed code in Section 4.

We propose to write such crosscutting assertions in an aspect. In Aspect-Oriented Programming[16], an aspect is a module unit for a crosscutting structure such as above example. The features of our approach are following:

- Our approach is based on aspect-oriented programming. It is important to separate crosscutting assertions from objects since assertion is often regarded as a part of the interface of an object[23]. The crosscutting assertions of objects affect modularity and maintainability of the objects.
- An aspect-oriented approach also enables developers to separate assertions into aspects for each purpose. Developers could not group assertions for each purpose in traditional approaches since traditional assertions are written for each method of a class. While Moxa also supports developers to group common properties for several methods, our approach al-

lows developers to group several properties of classes for each purpose.

- Crosscutting assertions are caused in various situations. For example, developers write a method along with assumptions for the usage of the method. The developers usually write such assumptions in a comment such as "This method `foo` is to be called from the method `bar`". We should assert such assumptions since other developers may accidentally break assumptions when they reuse the method, and violated assumptions often cause a defect. We discuss the applicability of our approach in Section 5.4.
- Developers can deploy an aspect including application specific assertions to legacy components. Heineman pointed out that a service should be provided to enforce local properties specified by components as well as global properties specified by the application[11].

In the next section, we present the details of our approach.

3. Assertion as an Aspect

We propose to modularize crosscutting assertions in aspects. First we discuss language constructs which are useful to describe assertion. After the short discussion, we introduce a new simple aspect-oriented language whose pointcut designators are a small subset of AspectJ[1] to show basic language constructs to write assertions.

Pre-/post-conditions of a method are checked before/after the method is called respectively. Therefore, we regard them as `before`/`after` advices with a `call` pointcut in AspectJ. When assertions are separated from the objects, the separated assertions need a way to access context information including a method caller object, a callee object, their own (member) objects, method parameters and a return value through context exposure provided by AspectJ.

Comparing with an aspect implementing some functionality (or non-functional requirements), an aspect for assertion has following features.

- An aspect observes method call events and often accesses contextual information, a caller object and a callee object. So a developer often uses pointcut designators including `call`, `this`, `target` and `args`.
- An aspect has utility methods and fields to collect information through several method calls.
- An aspect sometimes needs to access private members of the object.

We have developed a simple aspect-oriented language specialized to write an aspect for assertion based on the above features. Our language provides several pointcut designators to easily access context information.

We allow a module of our language to include a block written in AspectJ to declare methods for utility functions and advices handling context information. We have defined our language to be converted to AspectJ since our purpose is not to develop a new practical language but to use a simple subset of AspectJ to write assertions.

3.1. Assertion Module

In this language, a developer declares assertion aspect using the keyword `assertion`. The declaration of a module consists of the name of the module, a module level pointcut and a set of advices.

```
assertion name ( params ) : pointcut
  << advice definition >>
end
```

The name of a module is provided just for management, it has no meaning in programming semantics. A module may have a module level pointcut describing a common pointcut among advices included in the module. The pointcut is optional, a developer may omit ": pointcut" fragment. Our language provides following primitive pointcuts to specify context in simple expression:

- `p calls q` represents a method call from `p` to `q`. `p` and `q` usually specify objects declared as parameters of the module. `p` and `q` may be type name or a wild card "`*`" when the developers have no interest in caller/callee objects, respectively. This statement is translated into following pointcut designators: `call(* *.*(..)) && this(p) && target(q)`.
- `p calls q.method(params)`, which is another form of the pointcut, is also allowed to specify the signature of the methods. This form is translated into the following pointcut designators: `call(* *.method(..)) && args(params) && this(p) && target(q)`.
- `if(expr)` represents a condition of the context. The assertions in the context module are enabled when the `expr` is true. This pointcut is exactly same as `if` pointcut of AspectJ.
- `method(signature)` represents a method signature constraint. This pointcut is simply converted to `call(signature)` pointcut. A developer uses this pointcut to specify methods when the developer is not interested in objects.

Although above pointcuts are sufficient to write usual assertions, other pointcut designators are also useful to

write assertions. For example, `cflowbelow` can specify pre/post-conditions for recursive method calls. Examining how powerful pointcuts such as `cflow` and `dflow`[20] affect the expressiveness of assertions is future work.

Parameters in the module declaration are module level variables, so all advices in the module can access the parameters. It allows developers to declare common parameters for each assertion.

3.2. Assertion Advice

An assertion module includes a number of advices. An advice is defined in the following format:

```
def name ( params ) : pointcut
  pre
    <<expression or code block>>
  post
    <<expression or code block>>
end
```

The name, the parameters and the pointcut of an advice are same as the module level declaration. `pre` and `post` specify preconditions and postconditions, respectively. A developer writes a list of boolean expressions and code blocks in AspectJ. A list of expressions separated by "`;`" specifies conditions that must be satisfied. A code block is a procedure executed at the beginning or the end of the method. We assume that the code block updates variables for assertion checking. Here we show an example code as follows.

```
assertion OneSubjectManyObserver
 def attach(Observer o, Subject s):
  * calls s.attach(o)
   pre  o.subject == null;
   post o.subject == s;
        s.getObservers().contains(o);
        {
         Logger.log(o, "connects", s);
        }
 end
end
```

The above code is same as following AspectJ code:

```
aspect OneSubjectManyObserver {
 // condition attach
 before(Observer o, Subject s):
  call(* Subject.attach(Observer)) &&
  target(s) && args(o) {
    assert o.subject == null;
 }
 after(Observer o, Subject s):
  call(* Subject.attach(Observer)) &&
  target(s) && args(o) {
   assert o.subject == s;
   assert s.getObservers().contains(o);
   Logger.log(o, "connects", s);
 }
}
```

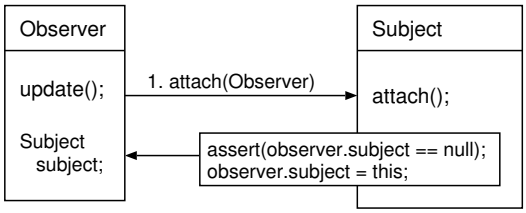

Figure 2. Crosscutting assertion in Java

And an assertion module may include utility methods, member variables (fields), inter-type declarations, internal classes and arbitrary advices in AspectJ. The format is very simple as follows.

```
{
  <<AspectJ Code Block>>
}
```

3.3. Implementation

We have implemented a translator from our language into AspectJ using `Racc`, or a parser generator for Ruby[24]. Our translator converts pointcut declarations to AspectJ style, and just copies code blocks to the output. Therefore, developers can use several pointcut designators which are not directly supported by our translator, e.g. `cflow`. Since our language is translated into AspectJ, the code optimization of AspectJ provides executable code with less-overhead[12].

4. Case Study

We have implemented an Observer pattern with the one subject-to-many observers relationship constraint as a case study. One-to-many relationship means that an observer can register to only one subject, and one subject can be observed by multiple observers. Such relationship is found when a subject represents a data model and several views for the model are provided as observers. Developers can simply reuse normal Observer pattern to achieve the purpose, but some constraints are needed to prevent an observer to watch several subjects. Here, we are comparing two implementations, in Java and our language, in the view point of the modularity.

Figure 2 shows the overview of the implementation in Java, and Figure 3 shows code fragments added to the usual

```
// Extended interface for Observer
interface Observer2 extends Observer {
  public Subject getSubject();
  public void setSubject(Subject subject);
}

// Extend an Observer
public class NewObserver extends AnObserver
  implements Observer2 {
  Subject subject;

  public void setSubject(Subject subject) {
    this.subject = subject;
  }

  public Subject getSubject() {
    return subject;
  }
}

// New subject
public class ASubject implements Subject {
      :
      :
  public void attach(Observer o) {
    assert (o instanceof Observer2) &&
     (((Observer2)o).getSubject() == null);
    assert !observers.contains(o);
    this.observers.add(o);
    assert observers.contains(o);
    ((Observer2)o).setSubject(this);
  }

  public void detach(Observer o) {
    assert (o instanceof Observer2) &&
     (((Observer2)o).getSubject() == this);
    assert observers.contains(o);
    this.observers.remove(o);
    assert !observers.contains(o);
    ((Observer2)o).setSubject(null);
  }
}
```

Figure 3. One-to-many relationship in Java

Observer pattern implementation. This implementation is problematic because it damages the encapsulation of Observer. An observer has a reference to a subject and the method `Subject.attach` checks and updates the field. When an observer calls `attach`, the subject checks that the observer is not connected to any subjects using the `subject` field of the observer. After the subject accepts the observer, the subject updates the observer's field. The observer must not modify the field by itself nevertheless it is the field of the observer. It is a bad manner to prevent a component to modify its field and to allow another component to modify the field.

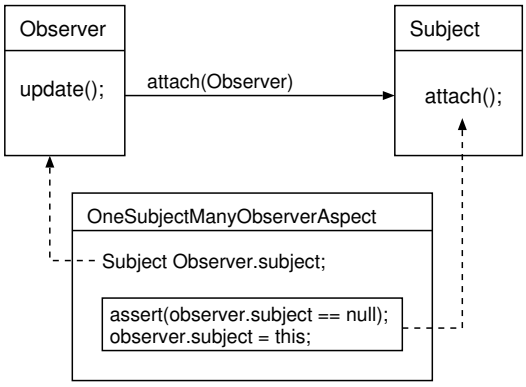

Figure 4. Assertion modularized in an aspect

The broken encapsulation also affects maintainability of the code. Developers need to maintain two versions of the subject and the observer for one-to-many and many-to-many relationships because the code fragments included in the observer and the subject depend on each other to implement the one-to-many constraint. Developers cannot mix one-to-many observers and many-to-many observers for one subject.

Figure 4 shows the overview of a solution in our approach. The aspect has separated crosscutting assertions from subjects and observers. The source code is shown in Figure 5. The field of `Observer.subject` is also moved to the aspect. The aspect introduces the field `subject` into Observer. Its value must be null before the method `attach`, and a reference to subject is set to the field after `attach` is called. The value is cleared after the method `detach` is called.

The aspect modularizes all related assertions, fields and methods. The modularization prevents a developer to accidentally mix these assertions with assertions for other purposes and to misuse methods and fields defined only for the assertions.

An advantage of our approach is that developers can deploy one-to-many relationship for generic observers and subjects. Another advantage is that developers can mix one-to-many observers and many-to-many observers for one subject since the aspect affects only a pair of `AnObserver` and `ASubject`.

Our language enables a developer to easily write assertions. When a developer uses AspectJ to implement one-to-many observer pattern aspect, the implementation is similarly modularized as our language. The difference from our language is that a developer writes a pair of before and after advices for a method in AspectJ instead of a set of preconditions and postconditions in one advice.

```
assertion OneSubjectManyObserver

 { // AspectJ inter-type declaration
   public Subject AnObserver.subject = null;
 }

 def attach(ASubject s, AnObserver o):
   * calls s.attach(o)
 pre   !s.getObservers().contains(o);
       o.subject == null;
 post  s.getObservers().contains(o);
       { // code executed after s.attach
         o.subject = s;
       }
 end

 def detach(ASubject s, AnObserver o):
   * calls s.detach(o)
 pre   s.getObservers().contains(o);
       o.subject == s;
 post  !s.getObservers().contains(o);
       {
         o.subject = null;
       }
 end
end
```

Figure 5. One-to-many relationship aspect

5. Discussion

In this section, we discuss about behavioral subtyping, modular reasoning, situations to which our approach is applicable and related work.

5.1. Behavioral Subtyping

Behavioral subtyping guarantees that all objects of a subtype preserve all of the original type's invariants[5]. Our approach enables developers to add assertions to a component using an aspect, or an external module. When a developer creates a new subclass of a component, assertion should automatically affect the new subclass for the consistency of the assertion. For example, when an aspect adds an assertion to a class P, a class Q which is a subclass of P is also affected by the assertion.

Our approach allows developers to add preconditions to a component. This feature may break behavioral subtyping since a subtype may have weak preconditions and strong postconditions of the supertype, but cannot have strong preconditions and weak postconditions[19]. However, we decided to allow strong preconditions since a developer sometimes can assume the stronger preconditions based on application constraints[11].

On the other hand, we prohibit removing assertions from objects for safety. Although the behavioral subtyping allows weak preconditions, we prevent a developer from being confused by mixing a code fragment explicitly violating a precondition with an aspect removing the precondition.

5.2. Modular Reasoning

Assertion documents the abstract behavior of a method. Well-modularized assertions provide useful information for a developer to understand interactions among objects.

Assertions separated from a component might reduce the comprehensibility of the component since assertions for a component crosscut a class and several assertion modules. Therefore, Aspect Visualizer[2] and tools finding aspects are important for managing aspects. If a developer finds aspects using Aspect Visualizer, the developer can inspect all related assertions. This is easier than finding all crosscutting assertions in other classes with `grep` or other tools[17]. A program just crashes even if a developer who does not know assertion added by an aspect writes a code fragment which violates the assertion. A developer can get assertion information from the stack trace after a program crashed, so it is not too serious problem.

5.3. Avoiding Side Effects

Assertion aspect should have no side effects for the state of objects. If the side-effect free methods are available, we should enforce aspects to call only such methods. Java with Access Control proposed by Kniesel et al.[18] is promising approach for this purpose since it provides `readonly` context prohibiting side effects and a method statically checking the context.

We allow an aspect to have its own state in order to calculate some statistical value, record a log or other similar purposes. Assertion aspect does not reduce maintainability since a developer does not need impact analysis for an assertion aspect when the aspect has no side effect on other objects.

5.4. Applicability

We have shown the example of the Observer pattern. Here we discuss the applicability of our approach. Our approach is also applicable to following situations:

- *Client specific assertions for a reusable component.* A simple example is a list (e.g. `java.util.List`) containing strings which match a certain regular expression. If several developers want to share such a list in their application, they may develop a new list component. However, many lists are hard to maintain when

each component needs a customized list respectively. This problem is well-known since a developer often needs to handle a set of data with containers, and C++ template mechanism and Java Generics support containers with type constraints. Our assertion approach supports other constraints that cannot be handled in type checking mechanisms.

- *Assertions for experimental/untrusted code.* A developer can add strict assertions to only new code but not to other well-tested code when the developer creates a new client accessing a long-lived component. After the new client is tested, the developer may remove assertions easily by just removing the assertion aspect. A developer can also write assertions for untrusted input from an external component added by a user after the system is released, e.g. a plug-in extending the software.

- *Assertions describing developers' expectation.* Developers sometimes have implicit assumptions such as "When this method `foo` calls the method `bar`, the object holds a particular condition." Although some developers declare this kind of expectation as a comment, other developers may accidentally break such assumptions when they introduce subclasses or aspects[22]. Our assertion may protect the method from the illegal usage by specifying the caller's state. Specifying strict assertion for the usage of a component, developers might assure the behavior and the quality of the component.

- *Assertions for component behavior affecting other components.* A component usually accesses other components to achieve its task. Our assertion can specify a condition about method calls to other components. Since a wrong series of method calls from a component indicates a defect of the component[4], an assertion checking the behavior of an object is useful for developers. Developers can write assertions and utility advices checking the behavior of the object independent from the object code. Temporal invariants are also useful for this purpose[7].

- *Collaborating with unit testing.* JUnit[14] is a well-known unit testing tool for Java. JUnit provides class libraries which supports to write assertions and a tool which executes a test suite and collects the result. Our approach supports to test interactions among objects in addition to unit testing since assertion methods of JUnit can specify only the state of an object but cannot specify the expected behavior of the object, e.g. methods should be called in a test case.

5.5. Related Work

Zhao et al. proposed Pipa, or an extended language of JML which enables programmers to write assertions for advices[28]. Hanneman et al. have shown the usefulness of aspects implementing interactions among objects[9], and published their code[10]. If a developer introduces Pipa into a Observer pattern implemented in AspectJ, the result may be similar to our approach because Pipa code fragments are assertions for each advice and our code fragments are translated into assertion statements in advices. However, the purpose of Pipa is different from ours. Pipa aims to introduce assertion checking for advices, our approach aims to modularize crosscutting assertions. The concept of crosscutting assertion in our approach includes assertion for a set of the components which may be coded only in classes as main functionality (base code).

Yamada et al. proposed Moxa, another extension of JML[27]. Moxa provides an assertion module which enables programmers to write assertions shared by several methods and classes. Our approach focuses on modularizing crosscutting assertions into an aspect, their approach focuses on modularizing common properties into an aspect. The approaches may collaborate with each other.

Gibbs et al. have proposed Temporal Invariants, or an extension of assertion for temporal properties for one component[7]. Temporal invariants can describe temporal properties held in a series of method calls for one component. This approach can replace a process collecting context information (some flags checking control flow) with an expression of the temporal logic. When a developer can write temporal invariants for several components, it would be more useful tool.

6. Summary

Assertion documents the behavior of a component. Assertion checking is a powerful tool to detect software faults during debugging, testing and maintenance. Since traditional assertions are described for each method, the assertions crosscut several modules in order to specify inter-object properties. Crosscutting assertions are harmful to the modularity and maintainability of the components. Therefore we have proposed to modularize such assertion as an aspect using an aspect-oriented language. We have introduced a simple aspect-oriented language to show basic language constructs to write assertions, and developed a translator for the language into AspectJ. We have implemented two version of the Observer pattern in java and our language and have shown that crosscutting assertions in Java are modularized in our language. Aspect-oriented assertion is promising to improve software maintainability, and is applicable to various situations. In the future work, we are plan-

ning to research design by contract for inter-aspect properties and examine how powerful pointcuts such as `cflow` and `dflow` affect the expressiveness of assertions.

Acknowledgement

This work was supported by MEXT.Grant-in-Aid for JSPS Fellows (No.17-9539).

References

[1] The AspectJ Team. http://eclipse.org/aspectj/

[2] Clement, A., Colyer, A. and Kersten, M.: Aspect-Oriented Programming with AJDT. *Proceedings of Workshop on Analysis of Aspect-Oriented Software (AAOS 2003)*, Darmstadt, Germany, July 2003.

[3] Contract4J, http://www.contract4j.org/

[4] Dallmeier, V., Lindig, C. and Zeller, A.: Lightweight Defect Localization for Java. *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, Glasgow, UK, July 2005.

[5] Findler, R. B., Latendresse, M. and Felleisen, M.: Behavioral Contracts and Behavioral Subtyping. *Proceedings of the 9th Foundations of Software Engineering (FSE 2001)*, pp.229-236, Vienna, Austria, September 2001.

[6] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: *Design Patterns*. Addison-Wesley Pub Co., Boston, Massachusetts, 1995.

[7] Gibbs, T. H. and Malloy, B. A.: Weaving Aspects into C++ Applications for Validation of Temporal Invariants. *Proceedings of Conference on Software Maintenance and Reverse Engineering (CSMR 2003)*, pp.249-258, Benevento, Italy, March 2003.

[8] Guttag, J. V., Horning, J. J. and Wing, J. M.: The Larch Family of Specification Languages. *IEEE Software*, Vol.2, No.5, pp.24-36, September 1985.

[9] Hannemann, J. and Kiczales, G.: Design Pattern Implementation in Java and AspectJ. *Proceedings of the 17th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002)*, pp.161-173, Seattle, Washington, November 2002.

[10] Hannemann, J.: *Aspect-Oriented Design Pattern Implementations*. http://www.cs.ubc.ca/˜jan/AODPs/

[11] Heineman, G. T.: Integrating Interface Assertion Checkers into Component Models. *Proceedings of the 6th International Component-Based Software Engineering (CBSE 2003) Workshop*, pp.37-42, Portland, Oregon, May 2003.

[12] Hilsdale, E. and Hugunin, J.: Advice Weaving in AspectJ. Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), pp.26-35, Lancaster, UK, March 2004.

[13] JML Reference, http://www.dc.fi.udc.es/ai/tp/jml/JML/docs/jmlrefman/jmlrefman/

[14] JUnit. http://junit.org/

[15] Karaorman, M., Holze, U. and Bruno, J.: jContractor: A Reflective Java Library to Support Design By Contract. *Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection (Reflection 1999)*, pp.175-196, Saint-Malo, France, July 1999.

[16] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M. and Irwin, J.: Aspect-Oriented Programming. *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, pp.220-242, Jyvaskyla, Finnland, June 1997.

[17] Kiczales, G., Mezini, M.: Aspect-Oriented Programming and Modular Reasoning. *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pp.49-58, St. Louis, Missouri, May 2005.

[18] Kniesel, G. and Theisen, D.: JAC - Access Right Based Encapsulation for Java. *Software Practice and Experience*, Vol.31, No.6, pp.551-576, May 2001.

[19] Liskov, B. H. and Wing, J. M.:A Behavioral Notion of Subtyping. ACM Transactions on Programming Languages and Systems,Vol.16, No.6, pp.1811-1841, November 1994.

[20] Masuhara, H. and Kawauchi, K.: Dataflow Pointcut in Aspect-Oriented Programming. *Proceedings of the 1st Asian Symposium on Programming Languages and Systems (APLAS 2003)*, pp.105-121, Beijing, China, November 2003.

[21] McCamant, S. and Ernst, M. D.: Predicting Problems Caused by Component Upgrades, *Proceedings of the 11th Foundations of Software Engineering (FSE 2003)*, pp.287-296, September 2003.

[22] McEachen, N. and Alexander, R. T.: Distributing Classes with Woven Concerns - An Exploration of Potential Fault Scenarios. *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pp.192-200, Chicago, Illinois, March 2005.

[23] Meyer, B.: *Object Oriented Software Construction*. Prentice Hall, New York, New York, 1988.

[24] Racc: LALR(1) Parser Generator for Ruby. http://www.loveruby.net/en/prog/racc.html

[25] Rosenblum, D. S.: A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, Vol.21, No. 1, pp.19-31, January 1995.

[26] Siedersleben, J.: Errors and Exceptions - Rights and Responsibilities. *Proceedings of Workshop on Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms*, pp.2-9, Darmstadt, Germany, July 2003.

[27] Yamada, K. and Watanabe, T.: Moxa: An Aspect-Oriented Approach to Modular Behavioral Specifications. *Proceedings of Workshop on Software-Engineering Properties of Languages and Aspect Technologies (SPLAT 2005)*, http://www.daimi.au.dk/˜eernst/splat05/, Chicago, Illinois, March 2005.

[28] Zhao, J. and Rinard, M.: Pipa: A Behavioral Interface Specification Language for AspectJ. *In Proceedings of the 6th International Conference on Fundamental Approaches to Software Engineering (FASE 2003)*, pp.150-165, Warsaw, Poland, April 2003.