Assertion with Aspect

Takashi Ishio[†], Toshihiro Kamiya[‡], Shinji Kusumoto[†], Katsuro Inoue[†]
[†] Graduate School of Engineering Science,
Osaka University
1-3 Machikaneyama-cho, Toyonaka,
Osaka 560-8531, Japan
[‡] PRESTO, Japan Science and Technology Agency
1-3 Machikaneyama-cho, Toyonaka,
Osaka 560-8531, Japan
{t-isio, kamiya, kusumoto, inoue}@ist.osaka-u.ac.jp

March 1, 2004

1 Solution Name: Assertion with Aspect

We propose to use aspects to declare assertion. *Assertion* consists of an assertion statement, preconditions and postconditions[3].

2 Motivation

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

Currently, all the assumptions are not described as assertions. For example, programmers often write a method along with assumptions for the usage or the purpose of the method. Programmers usually write such assumptions in a comment, for example, "This method FOO is to be called from method BAR". When another programmer reuses the method or adds a new aspect, he or she may accidentally break an assumption. Since the violated assumption may cause a defect, we should write an assumption as an assertion statement to check the assumption in runtime.

An assertion statement represents an assumption about the method behavior and the states of the related objects such as a method caller object, a callee object and their own objects. However, an assertion statement cannot refer to the control flow information (context of the method) of the program, and cannot explicitly represent the state consisting of multiple objects' states. We propose to write these assumptions as assertion extended for AOP.

In this manuscript, we use an example of a file updater class written in Java. This class has one public method `update_file` and three private methods called from the public method. Programmers have some assumptions in the method and each assumption is described as a comment with the keyword `ASSUMPTION`.

```
class FileUpdater {

  File file;
  OutputStream out;

  public FileUpdater(File f) {
    file = f;
  }

  public void update_file(Data data) {
    open_file();
    write_file(data);
    close_file();
  }
  private void open_file() {
    //ASSUMPTION 1: out is null and f is writable
    out = new FileOutputStream(f);
  }
  private void write_file(Data data) {
    //ASSUMPTION 2: this method MUST be called from update_file
    out.write(data);
  }
  private void close_file() {
    out.close();
    out = null;
  }
}
```

In Section 3, we replace the comments of assumptions to our assertion.

3 What is assertion?

In order to distinguish our proposal from original `assert`, we use a keyword `assume`. The default behavior of `assume(expr)` is the same as `assert(expr)`.

Unlike the traditional assertion, a *predicate method* can be used in an expression of `assume` statement. We use a term *predicate method* as a method which returns a boolean value.

Programmers use arbitrary named predicate methods in `assume` and implement the predicate methods in aspects. We show an example replacing `ASSUMPTION 1` to `assume` with a predicate method as follows:

```
  private void open_file() {
    assume(can_open_file(f, out)); //ASSUMPTION 1: out is null and f is writable
    out = new FileOutputStream(f);
  }
  aspect FileUpdaterAssumptions {
    // This is the implementation of the predicate method can_open_file.
    ASSERT boolean can_open_file(File f, OutputStream out) {
      return (out == null) && f.canWrite();
    }
    when violated: can_open_file(..) {
      System.err.println("can_open_file assertion is violated.");
    }
  }
```

In order to clarify that the method `can_open_file` is a predicate method, `ASSERT` keyword is introduced as a modifier of the method. We use a new language construct `when violated`. This is a kind of advice executed when a predicate method returns `false`. This advice is used to execute a code of cleaning up system resources or of logging an error before the program stops. Although programmers can write an error handling code using `try` statement, our approach allows programmers to add a new error handling using context-specific information.

In order to refer to control flow information in a predicate method, we introduce another new language construct `cflow` to write an assertion statement about the control flow. `cflow(obj.aMethod)` represents a boolean value whether `aMethod` of the object `obj` is contained or not in the call stack.

```
  // call open_file before this method is called
  private void write_file(Data data) {
    assume(can_write_file(this)); //ASSUMPTION 2: this method must
                                  //              be called from update_file
    out.write(data);
  }
  aspect FileUpdaterAssumptions {
    ASSERT boolean can_write_file(FileUpdater obj) {
      return cflow(obj.update_file);
    }
    :
    :
  }
```

The advantages of our approach are following:

First, our approach allows programmers to write an assertion statement at arbitrary execution points. Using AspectJ, programmers can write an aspect to inject `assert` statements into classes. Programmers easily add preconditions and postconditions to a method. However, programmers cannot express join points such as "the beginning of this loop". Our approach allows programmers to use a predicate method in an assertion statement. A predicate method works as a user-defined join points. Moreover, the assertion tells the programmer when the assumption is checked. Such an explicitly defined assertion statement supports programmers' predictability. We discuss the predictability in Section 5.

Second, our approach separates a context-specific assumption from the other assumptions, by using a context-specific implementation of a predicate method based on `cflow` construct. AspectJ also allows programmers to use the control flow information using `cflow` pointcut designator. Our `cflow` expression easily refers to such information. We discuss the comprehensibility in Section 4.

Finally, our approach enhances reusability of a program. Programmers expect that they can freely add and remove assertions (constraints) of a class since assertions do not modify the behavior of the original class. It enables programmers to manage application-specific constraints to improve the reusability of classes. Reusability is important in the software evolution process. We discuss reusability and evolvability in Section 6.

4 Does Assertion Support Comprehensibility?

4.1 General properties, assumptions, hypotheses about comprehensibility

An assertion can be considered as a kind of document for programmers. A programmer using or modifying a method reads assertions to understand the assumptions of the method.

4.2 How assertion supports comprehensibility

A programmer usually puts assumptions about the usage of a method. Programmers often write such assumptions as comments, not as assertion statements since programmers cannot express some assumptions about control flow and state of multiple objects. Using our approach, programmers can write such assumptions as assertion statements.

Assertion statements are executable. Therefore, programmers can test whether an assertion statement is correct or not, in the other words, whether the assumption is satisfied in the program execution. An assertion is more reliable than a comment since the comment may become obsolete while the software evolves.

4.3 How assertion reduces comprehensibility

When an assertion statement is redundant or complex, the programmer cannot understand the assertion statement.

4.4 Conclusion about comprehensibility

Our approach allows programmers to express assumptions about control flow. When assertions are simple and precise, the assertions aid programmers to understand the method.

5 Does Assertion support predictability?

5.1 General properties, assumptions, hypotheses about predictability

Assertion statements are to check the state of the program and do not modify the state.

5.2 How assertion supports predictability

Programmers usually regard an assertion as a statement without a side effect. Programmers can understand what properties are checked by the proposed assertion statement since the programmers easily find the aspects implementing the predicate method using `grep` or the similar tools.

An assertion also supports predictability of the method since assertions declared as pre-/postconditions express the method functionalities.

5.3 How assertion reduces predictability

When a method for assertions has a side effect, the code may be difficult to understand. For example:

```
// isSorted sorts an array.
ASSERT boolean isSorted(Array array) {
  if (!array.sorted()) array.sort();
  return true;
}

void useSortedArray(Array array) {
  assume(isSorted(array));  // the parameter must be a sorted array.
  doSomething(array);
}

void user() {
  useSortedArray(getUnsortedArray()); // no violation is occurred.
}
```

The method `isSorted` does sort an array instead of checking whether the array is sorted or not. This will be surprising for another programmer reading the method `user()` since it is hard to recognize that `isSorted` sorts an array from its name.

5.4 Conclusion about predictability

Our approach can support predictability when the programmers use assertion to express assumptions.

A Method with a side effect like the above example may cause a problem when it is used in an assertion statement. When the side-effect free methods like `const` in C++ are available, we should enforce programmers to write predicate methods as `const` methods.

6 Does Assertion support evolvability?

6.1 General properties, assumptions, hypotheses about evolvability

In software evolution process, software reuse is important. The unit of the reuse is a method or a class. When programmers want to reuse a functionality of a method, it is better that the programmers can reuse the method without modifications.

6.2 How assertion supports evolvability

Assertions usually help programmers to understand what a method does. Programmers can understand what the properties of the objects are required at the location of each `assume` statement. For example, when the programmers give invalid parameters to a method, assertion can detect it[4].

On the other hand, when a programmer creates a class, the programmer can separate a class from its assertions. When another programmer wants to reuse the class, the programmer decides whether he (or she) adopts each assertion or not. The programmer can decide to add new application-specific (strict) assertions to adapt a generic reusable class which has loose constraints.

6.3 How assertion reduces evolvability

Assertions sometimes prevent to reuse a method or a class when the assertions are too strict. The programmer should rewrite such a method or a class more general than original.

6.4 Conclusion about evolvability

An assertion supports reusing a program. However, it needs to prepare a mechanism or a design guideline to prevent a programmer writing a superfluous assumption.

7 Impact on other software engineering properties

We have described reusability of classes and methods in Section 6.

8 Discussion and Conclusion

We have proposed an approach to use aspects to declare assertions. Since an assertion allows programmers to express their assumptions, an assertion improves the comprehensibility and the predictability of the software.

Although an assertion is not a novel idea, there seems to be few knowledge how a programmer should write assertion in AOP.

The proposed approach allows programmers to write implementation of an assertion statement in an aspect with `assume` statement. This approach enables programmers to manage and extend assertions as aspects. We also enhance an assertion with `cflow` expression to contain the context-specific assumptions.

We have no ways to prevent programmers to implement a predicate method with a side effect. Such an assertion statement may confuse a programmer. Therefore, when a programmer can declare a method as a side-effect free (using a language construct such as `const` in C++), the assertion process tool will enforce the programmer to use methods to reduce side-effect problem.

Current design of the proposed language construct is a crude one. In the future work, we will refine the design and implement a prototype as a preprocessor for AspectJ.

References

[1] AspectJ Team: *The AspectJ Programming Guide.* http://dev.eclipse.org/viewcvs/indextech.cgi/˜checkout˜/aspectj-home/doc/progguide/

[2] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. and Irwin, J.: Aspect Oriented Programming. *In Proc. of the 11th annual European Conference for Object-Oriented Programming (ECOOP97)*, vol.1241 of LNCS, pp.220-242, 1997.

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

[4] Romanovsky, A.: Exception Handling in Component-based System Development. *In Proc. of the 25th Annual International Computer Software and Application Conference (COMPSAC 2001)*, pp. 580-586, 2001