Aspect-Oriented Programming with Model Checking

Naoyasu Ubayashi
Systems Integration Technology Center,
Toshiba Corporation
Tokyo, Japan
naoyasu.ubayashi@toshiba.co.jp

Tetsuo Tamai
Interfaculty Initiative in Information Studies,
Graduate School, University of Tokyo
Tokyo, Japan
tamai@graco.c.u-tokyo.ac.jp

ABSTRACT

Aspect-oriented programming (AOP) is a programming paradigm such that crosscutting concerns including synchronization policies, resource sharing and performance optimizations over objects are modularized as aspects that are separated from objects. A compiler, called a weaver, weaves aspects and objects together into a program. In AOP, however, it is not easy to verify the correctness of a woven program because crucial behaviors are strongly influenced by aspect descriptions. In order to deal with such problem, this paper proposes an automatic verification approach using model checking that verifies whether the woven program contains unexpected behaviors such as deadlocks. The objectives of this paper are as follows: 1) to verify the correctness of AOP-based programs using model checking, 2) to provide AOP-based model checking frameworks.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Aspect-oriented programming

Keywords

Model checking, Validation, Checking frameworks

1. INTRODUCTION

Aspect-oriented programming (AOP)[16][10] is a programming paradigm such that crosscutting concerns including synchronization policies, resource sharing and performance optimizations over objects are modularized as aspects that are separated from objects. Object-oriented programming (OOP) is suitable for encapsulating functions that objects present. However OOP is not necessarily adequate to describe crosscutting concerns over objects. Adding crosscutting concerns such as optimizations tends to cause tangling of optimizing code over a program. As a consequence, it is difficult to understand the program and maintain it. AOP is proposed in order to alleviate this problem. In AOP, program descriptions are divided into objects and aspects. A compiler, called a weaver, weaves aspects and objects together into a program. Using AOP, crosscutting concerns over objects can be described explicitly. In AOP, however, it is not easy to verify the correctness of a woven program because crucial behaviors including performance and synchronization are strongly influenced by aspect descriptions. Bugs affecting safety and liveness tend to be embedded in aspects that may have complex structures to describe crucial behaviors. In order to deal with such problems, this paper proposes an automatic verification approach using model checking that verifies whether a woven program contains unexpected behaviors such as deadlocks. Model checking has been used in order to verify the correctness of concurrent finite automata including LSI design and communication protocols. Recently, model checking has been applied to verification of concurrent software systems. Model checking will be useful for AOP validation because it is used to check global properties across multiple processes.

The objectives of this paper are as follows: 1) to verify the correctness of AOP-based programs using model checking, 2) to provide AOP-based model checking frameworks. Regarding the first objective, model checking is applied to verify automatically whether a program composed of aspects and objects satisfies requirements specifications. Using model checking, it is possible to find bugs concerning safety and liveness. These bugs may be hidden behind individual aspects and objects. Moreover, bugs may emerge in a program woven by a weaver even if bugs do not exist in individual aspects and objects. These kinds of bugs are caused by weaving policies such as the order weaving is to take. Regarding the second objective, an approach is proposed that separates property descriptions to be verified by model checking as aspects from object descriptions. Verification descriptions can be considered as one crosscutting concern because these descriptions are tangled in a program. It is useful to separate verification descriptions from objects in order to improve readability and maintainability of a program. In this paper, concepts and problems of AOP are shown in section 2. In section 3, an approach using model checking is introduced to address these problems. In section 4, an example applying model checking for verifying a program based on AOP is shown. Section 5 shows AOP-based model checking frameworks. Section 6 is a discussion. Lastly, in section 7, we conclude this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. AOSD 2002, Enschede, The Netherlands
Copyright 2002 ACM 1-58113-469-X/02/0004...$5.00

2. AOP AND RELATED ISSUES

2.1 AOP

In AOP, objects that describe functions requested for systems and aspects that describe crosscutting concerns are woven into a program using a compiler called *weaver*[2]. A point where a hook is placed to combine objects and aspects is called a *join point*. Using join points, a weaver weaves objects and aspects into a program. Subject-oriented programming[11], Composition Filter[1], Role Model[15][21][22] and Adaptive Programming[9] are other researches dealing with separation of crosscutting concerns[8]. There are several AOP languages such as AspectJ[2], an aspect-oriented extension to Java. A program in AspectJ is composed of aspects defined by *aspect* and ordinary Java classes. A join point is defined by *pointcut*. Execution points of method invocation and exception throwing can be specified as join points. Main language elements in AspectJ are statements for *introduction* and *advice*. *Introduction* adds a new method to a class that already exists. *Advice* modifies a method that already exists and appends crosscutting code before/after/ around a specified join point.

2.2 Issues on AOP

In the design phase of AOP, a programmer can define aspects corresponding to individual concerns without considering weaving processes. In the testing phase, however, the programmer has to consider weaving processes and imagine how the woven program behaves, because targets of testing are not individual objects and aspects but the woven program. There is a gap between the design phase and the testing phase in AOP.

Example: error logging

In general, code that handles error logging tends to be dispersed in many places in a program. If requirements for error logging are changed, it is necessary to change code fragments that are dispersed in the program. As a consequence, unexpected bugs may be inserted into the code. In the following code cited from [2], a description for error logging is separated as the aspect *PublicErrorLogging*. The join point *publicInterface* expresses points where objects instantiated from classes defined in the package *mypackage* receive messages corresponding to public methods. *After* statement adds code that displays an error message or writes a log when the program control returns from a method related to *publicInterface* or throws an exception in the method. *Call* picks up a method or constructor call join point based on the static signature. *Target* picks up all join points where target object is an instance of a type pattern, or of the type of the identifier.

```
package mypackage;
// --------
//  Class
// --------
class Foo { public void m1() {} }
class Bar extends Foo {
    public  void m1()  { super.m1(); }
    private void m2()  {}
}
class Log { void write(Object o) {} }
// --------
//  Aspect
// --------
aspect PublicErrorLogging {
  static Log log = new Log();
  // join point
  pointcut publicInterface ():
    call(public * *(..)) && target(mypackage..*);
  // advices
  after() returning (Object o): publicInterface() {
      System.out.println(thisJoinPoint);
  }
  after() throwing (Error e): publicInterface() {
    log.write(e);
  }
}
```

How can we answer the following questions:

- Is an error logged when an exception is thrown in the method *m1* of the class *Bar* ?
- Is an error logged when an exception is thrown in the method *m2* of the class *Bar* ?

The first property is true. But the second property is false. It is easy to check these properties in this case because there is only one aspect definition and a simple join point specification. However, it is not easy to check complex properties in general AOP as shown in section 4 because it is difficult to imagine behaviors of a program woven by a weaver.

3. VERIFICATION WITH MODEL CHECKING

3.1 Model checking

In this section, a verification approach using model checking is proposed. Model checking is a formal method that checks whether a structure (system) satisfies a property or not[6]. A type of state transition graph called a *Kripke structure* is used as a structure for model checking. If a structure M satisfies a formula ϕ in a state s, M is called a model of ϕ and is expressed as $M, s| = \phi$. Usually, M is defined as a finite state automaton and ϕ is expressed as a temporal logic formula. Temporal logic formalizes sequences of transitions between states in a system. CTL*(Computation Tree Logic) which describes properties of computation trees is a commonly used powerful temporal logic. A tree is formed by designating an initial state in the structure as the root and then unwinding the structure into an infinite tree[6] (Figure 1). A CTL* formula is composed of the following path quantifiers and temporal operators.

- path quantifier: A(for all computation paths), E(for some computation paths)
- temporal operator: X(next time), F(in the future), G(globally), U(until), R(release)

CTL and LTL(Linear Temporal Logic) are useful sub-logics of CTL*. CTL is a branching-time logic and LTL is a linear-time logic. In CTL, temporal operators quantify over the paths that are possible from a given state. In LTL, temporal operators are provided for describing events along a single computation path[6].

Figure 1: Computation tree of error logging

Example: error logging
Let M be a structure and let s be its initial state.

```
M,s |= AG(ThrowingException-Bar-m1
          -> AF ErrorLogging-Bar-m1)
```

This CTL* formula shows the following property: for all paths starting from s, any state satisfies the property that an exception is logged eventually (ErrorLogging-Bar-m1) if the exception is thrown in the method *m1* of the class *Bar*(ThrowingException-Bar-m1). Figure 1 illustrates a structure satisfying the above CTL* formula. A circle expresses a state and an arrow expresses a transition between states.

3.2 Main model checkers

SPIN[13] and SMV[17] are well used model checkers. Java PathFinder(Jpf)[12], Bandera[4][7] and ESC/Java[20] are specialized tools for checking properties of Java programs. In SPIN, a target system is modeled using the formal programming language PROMELA and checked based on LTL. Although SPIN is one of the most popular model checkers, there is the problem that it is difficult to describe a model using PROMELA because the syntax is specific to SPIN. Modern model checkers such as Jpf and Bandera treat a Java source program as a model in order to alleviate this problem. Using these tools, programmers are released from writing models specific to model checking. Methods of specifying property specifications are categorized into two approaches. One is an approach that specifies an LTL formula directly from a command line, and another is an approach that embeds assertions in a model. The former is useful in specifying global properties although programmers are requested to understand the notion of LTL thoroughly. In the latter case, programmers are not requested to understand LTL. It is easy to specify whether a property is valid at a specific code region, but it is difficult to specify global properties. These two approaches should be selected corresponding to checking objectives. Table 1 summarizes profiles of these checking tools.

Table 1: Checking Tools

Tool	Language	Property specification
SPIN	PROMELA	LTL, assertion
Java PathFinder	Java	assertion
Bandera	Java	LTL
ESC/Java	Java	assertion

Figure 2: Producer/Consumer problem

3.3 AOP verification with model checking

This paper proposes an approach to verify the correctness of AOP-based programs using model checking. Model checking is applied to verify automatically whether a program composed of aspects and objects satisfies properties specified as a temporal logic formula that describes requirements specifications such as the above (error logging example), safety, liveness, fairness and so on. Using model checking, properties that crosscut objects can be verified. Bugs affecting requirements specifications, safety, liveness and fairness may be hidden in individual aspects and objects. Moreover, bugs may emerge in a program woven by a weaver even if bugs do not exist in individual aspects and objects.

4. APPLICATION TO AOP

In this section, model checking is applied to an example program based on AOP. The example is a concurrent program that treats a Producer/Consumer problem. It is not trivial to verify the correctness of the example although the program is very simple.

4.1 Example

As shown in Figure 2, the example is a program in which a producer puts data into a buffer and a consumer gets the data from the buffer. The buffer is used cyclically. The producer puts data in if the buffer is not full. Otherwise, the producer waits until the buffer is not full. The consumer gets data if the buffer is not empty. Otherwise, the consumer waits until the buffer is not empty. The producer sets the halt-flag when it puts all data into the buffer. The exception *HaltException* is thrown if the consumer wants to get data but the buffer is empty and the halt-flag is set. Both the number of producers and the number of consumers are restricted to one for simplicity in the example.

Appendix A describes the example using AspectJ[1]. As shown in Figure 3, the aspect *ConcurrentAspect* (the aspect for concurrent processing) and the aspect *TraceAspect* (the as-

[1]Appendix A is rewritten from the program cited in [12] to AspectJ.

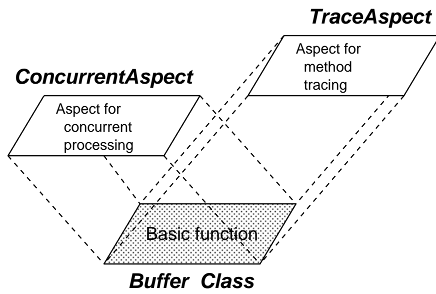

Figure 3: Buffer description based on AOP

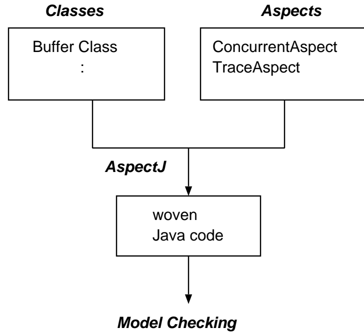

Figure 4: Model checking procedure

pect for message tracing) are separated from the class *Buffer* in the program. Only essential functions are described in the class *Buffer*—there are only two methods, *put* and *get*. Crosscutting concerns on the buffer are separated in Appendix A. Jpf style assertions are described in the class *Consumer* (line 64 – 69). In Jpf, assertions for model checking are described using the class named *Verify*[12]. Deadlocks can be detected whether assertions are declared or not.

4.2 Checking crosscutting concerns

It is not easy to understand the behavior of the woven program as a whole because the program is composed of six classes and two aspects. Actually, violations are detected at the assertions specified in the class *Consumer* when the example is verified using model checking. From the viewpoint of temporal logic, there is at least one transition path that starts from the initial state and does not reach states satisfying the assertions—states in which the number of data put in from the producer equals the number of the data received by the consumer—in the example's state transition space. In this case, there is a bug at $if(halted)$ (line 32 of the aspect *ConcurrentAspect*). The following is a counter example shown by a model checker[2] (Figure 4).

1. The producer puts three values into the buffer in the positions 0, 1, 2. After that, it calls the method *wait*.
2. The consumer gets the first value and then notifies the producer. After that, the consumer gets the second and the third values.
3. The producer puts the fourth value into the position 0, the fifth value into the position 1 and the sixth value into the position 2. Then the producer sets the halt-flag.
4. When the consumer calls the method *get*, the exception *HaltException* is thrown because $if(halted)$ is true. The consumer cannot get the remaining three values and the assertions specified in the class *Consumer* are violated.

[2] This counter example is based on [12].

The correct code is not $if(halted)$ but $if(usedSlots == 0)$. It is difficult to verify the correctness of the example because critical processes are described as aspects. As mentioned here, model checking is very useful in order to alleviate this problem. In fact, the above counter example is not always detected at runtime. If the consumer gets the remaining three values before the producer sets the halt-flag, the assertions specified in the Consumer class are valid. It is very difficult to remove these kinds of bugs completely just by runtime testings.

5. AOP-BASED MODEL CHECKING FRAMEWORK

In this section, an AOP-based model checking framework is proposed in order to use model checking tools efficiently. Using this checking framework, properties to be checked that crosscut over objects can be described as an aspect and separated from a program body. The AOP-based model checking framework can be used not only with model checking but also with runtime testing.

5.1 Basic concepts

In Appendix A, assertions for model checking are embedded in the class *Consumer*. Using this way, we must change source code whenever new checking properties are added. Properties can be regarded as aspects because they crosscut over objects. Properties to be checked are described as an aspect and separated from the body of the program in Appendix B. The aspect *VerifyAspect* in Appendix B is an alternative to line 64 – 69 of the class *Consumer* in Appendix A. This approach has the following advantages:

- Readability is improved because checking properties can be separated from original functions
- Checking properties that crosscut over objects can be encapsulated into an aspect
- It is not necessary to modify program code whenever new checking features are added. It is only necessary to weave aspects that describe the checking features
- Aspects are defined corresponding to checking viewpoints. Both a unit testing that weaves only one aspect and an integrated testing that weaves multiple aspects are possible.

One may argue that it is better to have a specification as a part of the program text rather than as a separate aspect. The argument may be true when there is only one checking property. In the case of multiple checking properties,

Table 2: Join points for AOP-based model checking framework

No.	Checking feature	Pre/post -condition	Join point	Advice
1.	method invocation	pre-condition post-condition	call(Signature) call(Signature)	before after
2.	state transition	pre-condition post-condition	set(Signature) set(Signature)	before after
3.	predicate dispatch	pre-condition post-condition	if(BooleanExp) if(BooleanExp)	before after

however, it is not easy to understand relationships between assertions and checking properties. When multiple tools are used in order to check a program, it is difficult to understand the program because different kinds of assertions that are specific to checking tools exist in the same code. Using aspects, properties needed for checking can be captured through join points. In other words, a checking property cannot be separated as an aspect unless AOP languages are unable to provide hooks that add checking code to target programs.

5.2 Join points needed for checking

AspectJ[3] supports the *Design by Contract* style popularized by Eiffel[18]. In this style, explicit pre-conditions test whether callers of a method call it properly and explicit post-conditions test whether methods properly do the work they are supposed to do properly. Pre- and post-conditions corresponding to method invocation join points are very useful in model checking. However, it is insufficient to test only using these pre- and post-conditions because properties including safety, liveness and fairness cannot always be described by them. These properties may be affected in the middle of method executions. It is necessary to catch other features including occurrences of state transitions and changes of specified conditions. In this section, AOP-based model checking frameworks that include not only method invocation features but also features including state transitions and changes of specified conditions are shown. Table 2 shows join points for AOP-based checking framework using AspectJ language constructs. There are three patterns that can be considered to set join points for checking. The first pattern is a join point corresponding to method invocation. This join point is expressed using *call* in AspectJ. The second pattern is a join point corresponding to the field settings. This join point expressed using *set* in AspectJ can be used to catch a state transition. Lastly, the third pattern is a join point corresponding to a predicate dispatching. This join point expressed using *if* in AspectJ can be used to catch a change of a specified condition. Although these join points alone cannot cover all checking features needed for model checking, they are useful in separating crosscutting concerns of model checking properties from individual objects and aspects.

5.3 Checking framework

The AOP-based model checking framework is composed of multiple checking aspects. A checking property is expressed as an aspect that describes pre- and post-conditions using join points as shown in Figure 5. The following is an example of a checking aspect in AspectJ.

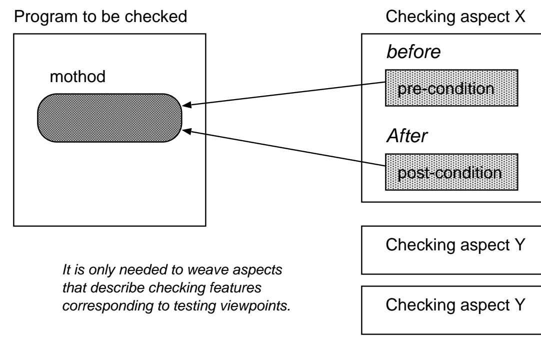

Figure 5: AOP-based checking framework

```
aspect Sample {
   // specify a join point
   pointcut fooPointcut(Foo o):
     call(public void bar()) && target(o);
   before(Foo o): fooPointcut(o){
     // specify a pre-condition
   }
   after(Foo o): fooPointcut(o){
     // specify a post-condition
   }
}
```

A checking aspect is prepared corresponding to a checking property. When AOP languages such as AspecJ are used, a super class of checking aspects can be defined using inheritance mechanisms if there are common properties among them. It is possible to construct checking frameworks similar to object-oriented frameworks. Unfortunately, this checking framework cannot cover all testing features. For example, this framework alone cannot detect deadlocks. In this case, tools such as Jpf that can detect deadlocks, whether assertions are specified or not, should be used together with it. As description styles of assertions and facilities of checking differ corresponding to checking tools, it is necessary to choose tools to be used and describe assertions corresponding to the syntax of the tool. It is only necessary to weave aspects that describe checking features corresponding to checking viewpoints. It is not necessary to change program code even if model checkers are changed.

Using aspects for model checking and aspects for runtime testings properly, integrated checking environments can be constructed. This framework may be applied to XP(eXtreme Programming)[5]. In XP, unit testings are executed using unit testing frameworks such as JUnit that are fitted to Java programming whenever program code is modified. The AOP-based model checking framework can be regarded as unit testing framework targeted on AOP.

6. DISCUSSION

In this paper, an approach that uses model checking in order to verify programs based on AOP is introduced. However, problems such as performance and memory efficiency are unavoidable to model checking. Because model checkers such as Jpf and Bandera adopt source code as a model, the space explosion problem cannot be ignored. For this problem, Bandera extracts a finite state model only related to a checking feature specified by an LTL formula from Java source

code using program slicing. Various approaches should be explored and combined to tackle this problem.

Although the verification approach proposed in this paper is targeted on programming phases, this approach can be applied to the design phase. Applications of AOP to the design phase are called AOD (Aspect-oriented design). While there are development environments such as AspectJ in AOP, environments for AOD are not prepared yet. Design contents should be translated to programming languages such as AspectJ manually. Because compositions of objects and aspects by weavers are enabled in the programming phase, a designer should imagine the result of weaving in order to check whether the result is correct or not in the design phase. Development environments that enable model checking in AOD will be needed in the future. An approach to formal verification of the properties of systems composed of crosscutting concerns is proposed in [19]. In [19], Alloy[14] and Labeled Transition Systems (LTS) are chosen as formal language. Alloy, a subset of Z, is suitable for checking systems with relationship between data elements. LTS is suitable for checking behaviors of concurrent systems.

The approach proposed in this paper applies model checking to a result of weaving objects and aspects. However, there could be an approach to the effect that objects and aspects should be checked separately because they are separated modules from the viewpoint of *separation of concerns*. Practically, it is important to check individual aspects and objects when programs woven from the aspects and objects are large. This is one of the problems to be resolved in the future. For example, an approach such that model checking is applied to a result of weaving an aspect and stub objects generated from the aspect may be one of the solutions to this problem.

7. CONCLUSIONS

In this paper, 1) a method to verify AOP-based program using model checking and 2) AOP-based model checking frameworks are proposed. Although there are problems that should be resolved in order to practice these approaches, the effectiveness of the approaches is confirmed.

8. REFERENCES

[1] Aksit, M. and Tripathi, A.: Data Abstraction Mechanisms in Sina/ST, *Proc. OOPSLA'88*, pp.265-275, 1988.

[2] AspectJ. http://aspectj.org/.

[3] The AspectJ Programming Guide, 2001.

[4] Bandera. http://www.cis.ksu.edu/ santos/bandera/.

[5] Beck, K.: Extreme Programming Explained: Embrace Change, Addison-Wesley, 1999.

[6] Clarke, E., Grumberg, O., and Peled, D.: Model Checking, The MIT Press, 1999.

[7] Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Pasareanu, C.S., and Zheng, H.: Bandera: Extracting Finite-state Models from Java Source Code, *Proc. ICSE 2000*, 2000.

[8] Czarnecki, K. and Eisenecker, U.W.: Generative Programming, Addison-Wesley, 2000.

[9] Demeter Project. http://www.ccs.neu.edu/research/demeter/.

[10] Elrad, T., Filman, R.E. and Bader A.: Aspect-oriented programming, *Communications of the ACM*, vol.44, no.10, pp.29-32, 2001.

[11] Harrison, W. and Ossher, H.: Subject-oriented Programming, *Proc. OOPSLA'93*, pp.411-428, 1993.

[12] Havelund, K.: Java PathFinder User Guide, 1999.

[13] Holzmann, G.J. and Smith, M.H.: The Model Checker SPIN, *IEEE trans. SE*, vol.23, no.5, pp.279-295, 1997.

[14] Jackson, D.: Aloca: the alloy constraint analyzer, *Proceedings of ICSE 2000*, 2000.

[15] Kendall, E.A.: Role Model Designs and Implementations with Aspect-oriented Programming, *Proc. OOPSLA'99*, pp.353-369, 1999.

[16] Kiczales, G., Lamping, J., Mendhekar A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J.: Aspect-Oriented Programming, *Proc. ECOOP'97*, Lecture Notes in Computer Science, Springer, vol.1241, pp.220-242, 1997.

[17] McMillan, K.L.: Symbolic Model Checking: An Approach to the State Explosion Problem, Kluwer Academic, 1993.

[18] Meyer, B.: Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 2000.

[19] Nelson, T., Cowan, D. and Alencar, P.: Supporting Formal Verification of Crosscutting Concerns, *Proc. REFLECTION 2001*, Lecture Notes in Computer Science, Springer, vol.2192, pp.153-169, 2001.

[20] K. Rustan M. Leino, Greg Nelson, and James B. Saxe: ESC/Java User's Manual, 2000.

[21] Tamai, T.: Objects and roles: modeling based on the dualistic view, *Information and Software Technology*, Vol.41, No.14, pp.1005-1010, 1999.

[22] Ubayashi, N. and Tamai, T.: Separation of Concerns in Mobile Agent Applications, *Proc. REFLECTION 2001*, Lecture Notes in Computer Science, Springer, vol.2192, pp.89-109, 2001.

APPENDIX

A. Producer/Consumer problem [AOP]

Buffer Class

```
 1: class Buffer {
 2:   static final int SIZE = 3;
 3:   Object[] array = new Object[SIZE];
 4:   int putPtr = 0;
 5:   int getPtr = 0;
 6:
 7:   public synchronized void put(Object x){
 8:     array[putPtr] = x;
 9:     putPtr = (putPtr + 1) % SIZE;
10:   }
11:
12:   public synchronized Object get()
13:                       throws HaltException {
14:     Object x = array[getPtr];
15:     array[getPtr] = null;
16:     getPtr = (getPtr + 1) % SIZE;
17:     return x;
18:   }
19: }
```

Concurrent Aspect

```
1: aspect ConcurrentAspect
2:       of eachobject(target(Buffer)) {
3:
4:   int usedSlots = 0;
5:   boolean halted = false;
6:
7:   // --- modify ``put'' method ---
```

```
 8:   pointcut bufferPut(Buffer b):
 9:     call(public * put(..)) && target(b);
10:
11:   before(Buffer b): bufferPut(b){
12:     while (usedSlots == Buffer.SIZE)
13:       try{
14:         b.wait();
15:       }catch(InterruptedException ex){};
16:   }
17:
18:   after(Buffer b): bufferPut(b){
19:     if (usedSlots == 0) b.notifyAll();
20:     usedSlots++;
21:   }
22:
23:   // --- modify ``get'' method ---
24:   pointcut bufferGet(Buffer b):
25:     call(public * get(..)) && target(b);
26:
27:   before(Buffer b): bufferGet(b){
28:     while (usedSlots == 0 && !halted)
29:       try{
30:         b.wait();
31:       }catch(InterruptedException ex){};
32:
33:     if (halted){  /*** BUG!! ***/
34:       HaltException he = new HaltException();
35:       b.throw(he);
36:     }
37:   }
38:
39:   after(Buffer b): bufferGet(b){
40:     if (usedSlots == Buffer.SIZE) b.notifyAll();
41:     usedSlots--;
42:   }
43:
44;   // --- add ``halt'' method (Introduction) ---
45:   public synchronized void Buffer.halt() {}
46:
47:   pointcut bufferHalt(Buffer b):
48:     call(public * halt(..)) && target(b);
49:
50:   before(Buffer b): bufferHalt(b){
51:     halted = true;
52:     b.notifyAll();
53:   }
54: }
```

Trace Aspect

```
 1: aspect TraceAspect {
 2:   pointcut publicInterface():
 3:                 call(public * *(..));
 4:
 5:   before(): publicInterface(){
 6:     System.out.println(
 7:       "[Trace public methods] "
 8:           + thisJoinPoint.toLongString());
 9:   }
10: }
```

Main Class, etc.

```
 1: // --- The Main class
 2: class Main {
 3:   public static void main(String[] args) {
 4:     Buffer b = new Buffer();
 5:     Producer p = new Producer(b);
 6:     Consumer c = new Consumer(b);
 7:   }
 8: }
 9:
10: // --- The Attribute class
11: class Attribute {
12:   public int attr;
13:   public Attribute(int attr) {
14:     this.attr = attr;
15:   }
16: }
17:
18: class AttrData extends Attribute {
19:   public int data;
20:   public AttrData(int attr,int data) {
21:     super(attr);
22:     this.data = data;
23:   }
24: }
25:
26: // --- The Producer class
27: class Producer extends Thread {
28:   static final int COUNT = 6;
29:   private Buffer buffer;
30:
31:   public Producer(Buffer b) {
32:     buffer = b; this.start();
33:   }
34:
35:   public void run() {
36:     for (int i = 0; i != COUNT; i++) {
37:       AttrData ad = new AttrData(i,i*i);
38:       buffer.put(ad);
39:       yield();
40:     }
41:     buffer.halt();
42:   }
43: }
44:
45: // --- The Consumer class
46: class Consumer extends Thread {
47:   private Buffer buffer;
48:
49:   public Consumer(Buffer b) {
50:     buffer = b;
51:     this.start();
52:   }
53:
54:   public void run() {
55:     int count = 0;
56:     AttrData[] received = new AttrData[10];
57:     try{
58:       while (count != 10){
59:         received[count] = (AttrData)buffer.get();
60:         count++;
61:       }
62:     }catch(HaltException e){};
63:
64:     // Verify.assert is an assertion for Jpf.
65:     Verify.assert("count != COUNT",
66:                   c.count == Producer.COUNT);
67:     for (int i = 0; i != c.count; i++){
68:       Verify.assert("wrong value received",
69:                     c.received[i].attr == i);
70:     }
71:   }
72: }
```

B. Aspect for model checking

Verify Aspect

```
 1: aspect VerifyAspect {
 2:   pointcut consumerRun(Consumer c):
 3:     call(public void run()) && target(c);
 4:
 5:   after(Consumer c): consumerRun(c){
 6:
 7:     // Verify.assert is an assertion for Jpf.
 8:     Verify.assert("count != COUNT",
 9:                   c.count == Producer.COUNT);
10:     for (int i = 0; i != c.count; i++){
11:       Verify.assert("wrong value received",
12:                     c.received[i].attr == i);
13:     }
14:   }
15: }
```