

# Formal Verification of a Bounded Buffer with Three Separate Concerns

Torsten Nelson

Donald Cowan

Paulo Alencar

University of Waterloo

{torsten,dcowan,alencar}@csg.uwaterloo.ca

## Abstract

This paper shows an example of a simple system with multiple concerns – a bounded buffer. The buffer is decomposed into three concerns: one which handles the FIFO behavior of the buffer, one which handles the buffer’s behaviour when empty, and one which handles it when full. We specify the buffer using an implementation language (Java) and a formal specification language (labeled transition systems). Semantically equivalent compositions are generated for each of the descriptions. Finally, we use a model checking tool to test different compositions of the system for the presence of deadlock.

## 1 Introduction

During the Advanced Separation of Concerns workshop at OOPSLA 2000, the “Formalization focus group” began work on the formalization of a multiple-concern bounded buffer to illustrate the need for specification-level support for crosscutting concerns. Our work involves the continuation of the work started by that focus group.

In this work we implement a bounded buffer composed of three crosscutting concerns, and integrate the concerns using a simple composition operator. The composed system is not exactly like the original, tangled version, and the difference may cause problems. We then model each concern using a labeled transition system, compose the concerns using a process equivalent to the one used to compose the Java concerns, and then use a model checking tool to look for errors.

Multiple reader and writer threads can access the bounded buffer. The access is mutually exclusive. If a reader thread attempts to read from an empty buffer, the thread blocks. The same happens when a writer thread attempts to write to a full buffer.

## 2 Java Implementation

### 2.1 Single Concern Implementation

The object-oriented implementation of the buffer in Java is shown in figure 1. By analyzing the implementation, we find three crosscutting concerns, which are highlighted in the figure. The first is the concern that actually inserts and removes elements from the buffer, implementing the FIFO policy. The second concern deals with the buffer’s behavior when it is empty. The final concern deals with the behavior of a full buffer.

### 2.2 Multiple Concern Implementation

Separating the buffer into concerns isolates each concern in a single package. Figure 2 shows two of the three concerns – FIFO policy and full buffer policy.

```

class BlockingBuffer {
    final int size = 1000;
    int first, last;
    int[] buf = new int[size];

    public synchronized int read() {
        while (first == last) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }

        int element = buf[first];
        first = (first + 1) % size;

        notify();

        return element;
    }
}

public synchronized void write(int element) {
    int nextposition = (last+1) % size;

    while ( nextposition == first) {
        try {
            wait();
        } catch (InterruptedException e) {}
        nextposition = (last+1) % size;
    }

    last = nextposition;
    buf[last] = element;

    notify();
}
}

```

- Empty buffer policy  
 - Full buffer policy  
 - FIFO policy

Figure 1: Java implementation of the bounded buffer

```

concern FIFOPolicy {
    class BlockingBuffer {
        final int size = 1000;
        int first, last;
        int[] buf = new int[size];

        public int read() {
            int element = buf[first];
            first = (first + 1) % size;
            return element;
        }
        public void write(int element) {
            int nextposition = (last+1) % size;
            last = nextposition;
            buf[last] = element;
        }
    }
}

concern FullPolicy {
    class BlockingBuffer {
        int size, first, last;

        public synchronized int read() {
            notify();
        }
        public synchronized void write(int element) {
            int nextposition = (last+1) % size;
            while ( nextposition == first) {
                try {
                    wait();
                } catch (InterruptedException e) {}
                nextposition = (last+1) % size;
            }
        }
    }
}
}
}

```

Figure 2: Implementations of the *FIFOPolicy* and *FullPolicy* concerns

### 2.3 Composing the Multiple Concerns

Any approach to composing concerns must address three issues: correspondence, behavioral semantics, and binding. For our example, we will use a single composition operator, “*followed-by*”, (described in our previous work [2]) that has the following properties:

- **correspondence:** lexical - elements in different concerns correspond if they have the same name
- **behavioral semantics:** corresponding data represents shared memory; variables with the same name are representations of the same variable. Corresponding methods are executed in the order given by the “*followed-by*” operator. The operator has semantics roughly equivalent to that of operators of implementation-level languages, such as AspectJ’s *before* advice and HyperJ’s *order before*.
- **binding:** static - all composition is performed at compile time

The composition consists of a single `BoundedBuffer` class with `read` and `write` methods. The data members of the class are the union of the data members of all three concerns. The body of each method is formed by copying the bodies of the corresponding method of each concern, in the order given by the composition expression.

There are six possible ways to compose the three concerns. Since the composition operator “*followed-by*” defines a temporal ordering of the concerns, we are only interested in the compositions where the FIFO policy is executed last, because it would be meaningless to read from the buffer before checking whether it is empty, or writing to it before checking whether it is full. Thus, we have two possible compositions:  $\{FullPolicy\ followed\ by\ EmptyPolicy\ followed\ by\ FIFOPolicy\}$ , and  $\{EmptyPolicy\ followed\ by\ FullPolicy\ followed\ by\ FIFOPolicy\}$ . Which composition to choose is uncertain – it seems likely that both will have the same behavior.

Figure 3 shows a java class consisting of the composition of the three concerns using the expression  $\{EmptyPolicy\ followed\ by\ FullPolicy\ followed\ by\ FIFOPolicy\}$ . Notice that the `write` method is slightly different from the original class – the line “`notify();`” is the first rather than the last.

```

class BlockingBuffer {
    final int size = 1000;
    int first, last;
    int[] buf = new int[size];

    public synchronized int read() {
        while (first == last) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        notify();

        int element = buf[first];
        first = (first + 1) % size;
        return element;
    }
}

public synchronized void write(int element) {
    notify();

    int nextposition = (last+1) % size;
    while ( nextposition == first) {
        try {
            wait();
        } catch (InterruptedException e) {}
        nextposition = (last+1) % size;
    }

    last = nextposition;
    buf[last] = element;
}
}

```

- Empty buffer policy  
 - Full buffer policy  
 - FIFO policy

Figure 3: Java implementation of the result of composing the three bounded buffer concerns with composition expression  $\{EmptyPolicy\ followed\ by\ FullPolicy\ followed\ by\ FIFOPolicy\}$

Is this a problem? The answer is not clear from examining the code. Since access to the buffer is mutually exclusive, perhaps it does not matter if the blocked threads are notified before or after the element is actually written to the buffer. However, the fact that the writing thread may block after notifying may prove to be a problem. Testing can be done to try to detect faults, but testing concurrent systems is challenging. An alternative is to use formal modeling tools that can perform automated analysis of the composed system and detect potential problems.

### 3 Labeled Transition System (LTS) Model

The system’s concurrent nature guided our choice of a formal method. Any problems that arise after composition are likely to be related to the multiple accesses to the buffer by reader and writer threads. Therefore, we use Labeled Transition Systems (LTS), as these are suitable for modeling the interactions between con-

current processes. Another reason was tool availability - we used Magee and Kramer's LTS Analyser tool (LTSA)[1], which is capable of performing checks for liveness, safety and fairness violations.

We will use the tool to check whether the system is deadlock free, which is a liveness property. Other properties we could check are whether writes to a full buffer never happen (a safety property) or whether elements can always be written to the buffer when there is space (a liveness property).

### 3.1 Multiple Concern Model

The specifications of the three concerns of a two-element buffer using LTSA are shown in figure 4. Each concern defines three processes: BUFFER, READER, and WRITER. A process has a set of states. Each state defines the transitions (actions) it accepts. For example, the line  $read \rightarrow read.suspend \rightarrow EMPTY$  under state EMPTY in figure 4(a) defines that from this state the process can perform a *read* transition, leading to an intermediate state where the only acceptable transition is *read.suspend*, which leads the process back to state EMPTY. LTSA makes no distinction between labeled states such as EMPTY and NOT-EMPTY, and intermediate states such as the one between the *read* and *read.suspend* transitions. However, our approach makes that distinction. We call the labeled states *principal* states. These are states that will be considered when determining the set of states in the composed system. A *transition sequence* connects two principal states.

Transitions with the same label in different processes are *shared actions*. If a shared action executes, all processes that share the action must perform the transition simultaneously. Thus, a process will block upon reaching a shared action until all processes that share it also reach the equivalent action.

Each concern models only what is needed in order to address its particular needs. For instance, the concern that handles the empty buffer policy (figure 4a) has only two states of interest: buffer empty and buffer not empty. How many actual elements are in the buffer is irrelevant. The processes use non-determinism to deal with their incomplete specifications. For instance, in the empty buffer policy concern (figure 4a), a *read* transition when the buffer is not empty can lead either to state EMPTY or to state NOT-EMPTY, since the buffer may have had a single element, or more than one. During verification, the model checker gives equal weight to each possibility. In other words, if a non-deterministic choice occurs infinitely often, then each possibility will occur infinitely often.

The state ERROR in figure 4(c) is a built-in error state that terminates a process. Ideally, after composition all transitions to error states should disappear. Each of the concern specifications was shown to be deadlock-free using the LTSA tool.

### 3.2 Composing the Formal Models

We need to compose the three LTSA concerns in a way that mimics the semantics of the composition operator defined in section 2. While creating the composed Java program using the operator was straightforward, composing the LTSA specifications is a more elaborate process. The composition expression used in this example is  $\{EmptyPolicy\ followed\ by\ FullPolicy\ followed\ by\ FIFOPolicy\}$ .

The composed system consists of three processes: BUFFER, READER, and WRITER. Each process in the composed system is formed by combining the corresponding processes of each concern.

Before composing the concerns, we must establish the correspondence and behavioral semantics for the composition process. In an LTS description, there are two kinds of elements that may correspond across concerns: states and transitions. We will use lexical correspondence for both, meaning that elements with the same names in different concerns are considered equivalent.

The *followed-by* operator is a merge-type composition operator. In the case of LTS descriptions, this implies that all transitions that appear in the concerns must appear in the composed system. The semantics

```

BUFFER = EMPTY,
EMPTY =
  ( read → read.suspend → EMPTY
  | write → read.notify → NOT-EMPTY ),
NOT-EMPTY =
  ( read → NOT-EMPTY
  | read → EMPTY
  | write → read.notify → NOT-EMPTY
  | read.resume → NOT-EMPTY
  | read.resume → EMPTY ).

READER = ( read → READAUX | read.resume → READER ),
READAUX = ( read.suspend → read.notify → READRESUME
  | read.notify → READAUX
  | doread → READER ),
READRESUME = (read.resume → READAUX | read.notify → READRESUME).

WRITER = ( write → WRITER ).

```

(a) Empty buffer policy

```

BUFFER = NOT-FULL,
NOT-FULL =
  ( read → write.notify → NOT-FULL
  | write → NOT-FULL
  | write → FULL
  | write.resume → NOT-FULL
  | write.resume → FULL ),
FULL =
  ( read → write.notify → NOT-FULL
  | write → write.suspend → FULL ).

WRITER = ( write → WRITEAUX | write.notify → WRITER ),
WRITEAUX = ( write.suspend → write.notify → WRITERESUME
  | write.notify → WRITEAUX ),
WRITERESUME = (write.resume → WRITEAUX | write.notify → WRITERESUME).

READER = ( read → READER ).

```

(b) Full buffer policy

```

BUFFER = EMPTY,
EMPTY = ( read → ERROR
  | write → dowrite → ONE ),
ONE = ( read → doread → EMPTY
  | write → dowrite → FULL ),
FULL = ( read → doread → ONE
  | write → ERROR ).

READER = ( read → doread → READER ).

WRITER = ( write → dowrite → WRITER ).

```

(c) FIFO policy

Figure 4: LTS specifications of bounded buffer concerns: (a) Empty buffer policy; (b) Full buffer policy; (c) FIFO policy

of *followed-by* are such that transitions belonging to concerns that come first in the composition expression

must come first in the composed system.

Conceptually, the composition of the three concerns is similar to the execution of all of their processes in parallel. To understand this better, imagine that all three BUFFER processes are running. All three share the *read* and *write* actions. If we execute a *write* transition from the initial state, all three must participate in the action, according to LTSA rules. Following this action we have several choices. The empty buffer concern can perform the *read.notify* action, or the FIFO concern can perform the *dowrite* action. Since we are simulating concurrent processes, both actions can occur at the same time. However, this is not what we want – we are trying to create a single, sequential bounded buffer process. The two actions above must both be executed, but in a well-defined order. The composition, therefore, will be equivalent to *one* of the possible sequential orderings of events in the parallel execution of all processes in the concerns.

The actual composition procedure has two steps: finding equivalent states, and merging transition sequences. Each state in a composed system is formed of a set of states, with one state from each of the concerns. Concern states that correspond must belong to the same set, otherwise the set will not represent a state in the composed system. For example, we cannot have a state in the composed system that is formed by combining state EMPTY from the EmptyPolicy concern (EB), state NOT-FULL from the FullPolicy concern (FB), and state ONE from the FIFO concern (FIFO), since state EMPTY also appears in the FIFO concern and therefore must be present in the set if EMPTY from EmptyPolicy is present. One of the states in the composed system will be the initial state. This state must correspond to the initial states of all processes, otherwise the composition is impossible.

Given these rules, there are only three sets of compatible states in the composed bounded buffer. We labeled the new states B0, B1, and B2. State B0 contains the initial states of all three processes, so it is the initial state of the composed system.

```
B0: (EB.EMPTY, FB.NOT-FULL, FIFO.EMPTY)
B1: (EB.NOT-EMPTY, FB.NOT-FULL, FIFO.ONE)
B2: (EB.NOT-EMPTY, FB.FULL, FIFO.FULL)
```

Having the set of states of the composed buffer, we need to determine the set of transition sequences from each state. In order to determine this set, we simulate the concurrent execution of the concerns following LTSA rules.

For example, to find the transitions from B0, we simulate the parallel execution of EmptyPolicy starting at EMPTY, FullPolicy starting at NOT-FULL, and FIFOPolicy starting at EMPTY.

We now determine all possible sequences of transitions from these states. If, at some instant during execution, there is a choice between two or more transitions, we must choose the one from the concern that comes earlier in the composition expression  $\{EmptyPolicy\ followed-by\ FullPolicy\ followed-by\ FIFOPolicy\}$ . If the transition sequence leads the processes to states that are not compatible (that is, that together do not form one of the states in the composed system), the entire transition sequence is discarded.

As an example, to find the transition sequences from state B0, we'll simulate the parallel execution of the concerns starting from its three corresponding states. One of the possible actions from these states is *write*. All three processes execute the action synchronously at first, and the action is added to the composed transition sequence. Note that in the FullPolicy concern we have two possibilities for the transition, as there is a non-deterministic choice. We must create one transition sequence for each of the choices. Next, we have a choice of action *read.notify* from the EmptyPolicy concern, or action *dowrite* from the FIFO concern (the FullPolicy concern has finished its sequence, no matter what choice we made). Since the EmptyPolicy concern comes before the FIFO concern in the composition sequence, the rule dictates that we must choose the *read.notify* action. Now, only the *dowrite* action is left, and we are left with two possible transition sequences:

```
write → read.notify → dowrite → (EB.NOT-EMPTY, FB.FULL, FIFO.ONE)
```

write → read.notify → dowrite → (EB.NOT-EMPTY, FB.NOT-FULL, FIFO.ONE)

Since the first combination of states (EB.NOT-EMPTY, FB.FULL, FIFO.ONE) does not correspond to a state in the composed process, it must be discarded, leaving us with the second transition sequence from state B0 to state B1.

While the correspondence and behavioral semantics follow the above description in the general case, there are some exceptions, transitions that must be handled differently so that composition succeeds. These are *suspend* and *resume* transitions. The reason for this special treatment is that the “suspend-notify” synchronization protocol used in the bounded buffer introduces function entry and exit points. That is, a process may enter the *read* and *write* functions not only through a function call, but also by receiving a notification from another thread after being suspended, in which case execution starts at the function location that immediately follows the suspend command.

We need to take multiple entry points into consideration because of the time-ordered nature of the composition. If a *suspend* transition is found in a concern, the execution should stop at that concern. Likewise, after a *resume* transition, the execution should proceed starting from the concern where the equivalent *suspend* occurred.

The behavior of *resume* transitions can be handled at the correspondence level. We must specify that resume transitions match their equivalent function-entry transitions. The syntax for this is the following:

```
read.resume  forward-match read
write.resume forward-match write
```

The *forward-match* correspondence operator means that the left-hand-side operand corresponds to the right-hand-side operand, but only when looking forward in the composition expression. For example, given the composition expression  $\{EmptyPolicy\ followed-by\ FullPolicy\ followed-by\ FIFOPolicy\}$ , any transition *write.resume* found in *FullPolicy* will correspond to any *write* found in *FIFOPolicy*, but not to *write* transitions found in *EmptyPolicy*.

The *suspend* transitions affect the behavior of the composition. We must specify that transitions that would ordinarily follow a *suspend* should not be executed. We use the following syntax:

```
last-transition read.suspend
last-transition write.suspend
```

The transition-level unary operator “*last-transition T*” overrides the concern-level operator “*followed-by*” for specific transitions. It signals that when a transition *T* is reached, the system should end the transition sequence and move to a principal state.

Figure 5 shows the result of composing the three concerns of the bounded buffer following the composition expression  $\{EmptyPolicy\ followed-by\ FullPolicy\ followed-by\ FIFOPolicy\}$  and the composition procedure outlined above. It is interesting to note how the transitions to ERROR from the FIFOPolicy concern don’t appear in the composed specification. This is because the special semantics we have assigned to the *suspend* transitions have made reaching the error states impossible.

### 3.3 Verifying the Composed Model

Having the composed model, we used the LTSA tool to try to detect possible deadlocks. We also generated different composed specifications from the concern specifications and compared their effects.

In addition to the specification shown in figure 5, we tested the specification  $\{Full-Policy\ followed-by\ EmptyPolicy\ followed-by\ FIFOPolicy\}$ . The specifications shown so far assume an infinite number of read and write functions. In order to test more realistic boundary cases, we created two modified specifications

```

BUFFER = B0,
B0 =
  ( read → read.suspend → B0
  | write → read.notify → dwrite → B1
  | write.resume → dwrite → B1 ),
B1 =
  ( read → write.notify → doread → B1
  | write → read.notify → dwrite → B2
  | read.resume → write.notify → doread → B0
  | write.resume → dwrite → B2 ),
B2 =
  ( read → write.notify → doread → B1
  | write → read.notify → write.suspend → B2
  | read.resume → write.notify → doread → B1 ).

READER = ( read → READAUX | read.notify → READER),
READAUX = ( read.suspend → read.notify → READRESUME
  | doread → READER
  | read.notify → READAUX ),
READRESUME = (read.resume → READAUX | read.notify → READRESUME).

WRITER = (write → WRITEAUX | write.notify → WRITER ),
WRITEAUX = ( write.suspend → write.notify → WRITERESUME
  | write.notify → WRITEAUX
  | dwrite → WRITER ),
WRITERESUME = (write.resume → WRITEAUX | write.notify → WRITERESUME).

```

Figure 5: Result of composing the three bounded buffer concerns

Table 1: Result of checking composed buffer for deadlocks using the LTSA tool

Composition	Infinite accesses	Finite accesses
{ <i>FullPolicy followed-by EmptyPolicy followed-by FIFOPolicy</i> }	No deadlock	No deadlock
{ <i>EmptyPolicy followed-by FullPolicy followed-by FIFOPolicy</i> }	No deadlock	DEADLOCK

that read and write exactly three elements, one more than the buffer size. Table 1 shows the results of testing the four specifications for deadlock using the LTSA tool.

The results are interesting as only one of the compositions resulted in deadlock, and only when the reader and writer threads halt after a finite number of transactions. Deadlock happens in the following situation: the writer thread writes two elements to the buffer; in the third write attempt, the buffer is full; it notifies the reader and suspends; the reader thread reads two elements, notifying the writer; in the third read attempt, it suspends; the writer thread writes the last element and quits, leaving the reader suspended with an element available for reading in the buffer.

The deadlock doesn't happen with the other composition because the buffer begins in the empty state. If the initial state of the buffer had been full, we would find the opposite situation: the first composition would deadlock, and the second would not.

## References

[1] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 1999.

[2] Torsten Nelson, Donald Cowan, and Paulo Alencar. A model for describing object-oriented systems from multiple perspectives. In *Fundamental Approaches to Software Engineering, FASE 2000*, volume 1783 of *Lecture Notes in Computer Science*, Berlin, Germany, 2000. Springer.