

# A Structured Approach for Developing Concurrent Programs in Java

Masaaki Mizuno  
Department of Computing and Information Sciences  
Kansas State University, Manhattan, KS 66506  
masaaki@cis.ksu.edu

**Keywords:** Concurrency, Global invariant, Java multi-threaded programming

## 1 Introduction

In recent years, concurrent programming has become the norm rather than the exception in many applications. In particular, popularity of the Java programming language has accelerated this trend. Most textbooks on Operating Systems and concurrent programming teach concurrent programming by demonstrating solutions for some well-known problems, such as the producer/-consumer, readers/writers, and dining philosophers problems.

A more systematic and formal approach to develop concurrent programs is presented in [1, 2]. In this approach, for a given problem, we first specify a *global invariant* that implies the safety property<sup>1</sup>. Then, we develop a so-called *coarse-grained solution* using the two synchronization constructs:  $\langle \mathbf{await} B \rightarrow S \rangle$  and  $\langle S \rangle$ . Using Programming Logic, the global invariant is formally verified in the coarse-grained solution. Finally, the coarse-grained solution is mechanically translated to a *fine-grained* semaphore or monitor program that maintains the global invariant.

This approach has many advantages. First, this is a formal approach that enables verification of programs being developed. Second, the most important activity in the programming process lies at a high level; namely, specifying global invariants. Once an appropriate global invariant is specified, much of the rest of the process is mechanical. Furthermore, global invariants and coarse-grained solutions are platform (synchronization primitive) independent. Thus, if the platform is switched from a semaphore-based to a monitor-based system, we only need to translate the existing coarse-grained solution to a monitor-based fine-grained program.

The Java programming language encourages the use of multiple threads. Therefore, as Java's popularity grows, concurrent programming using Java synchronization primitives will become more important. Java provides monitor-like synchronization primitives. However, these primitives have limitations. Each Java monitor object can only have one condition variable, which is associated with the object itself; all waits and signals (called *notify* in Java) refer to it. The translation

---

<sup>1</sup>The safety property asserts that the program never enters a bad state.

to monitor programs presented in [1, 2] assumes multiple condition variables. Even though it is possible to map all condition variables in a translated fine-grained solution into the unique condition variable of a Java monitor, it yields an inefficient program.

Hartley gives an exercise problem to develop a `ConditionVariable` class and provides enough hints for it [4]. With this class, we can still use the formal method. However, it yields inefficiency. A more preferable approach is to develop a direct translation from coarse-grained solutions to efficient fine-grained programs in Java synchronization primitives. This paper presents such a translation that preserves global invariants. The translation uses *specific notification locks* [3] (also called *notification objects* [4]) and is classified in a design pattern called the *specific notification pattern* [3, 5].

The rest of the paper is organized as follows: Section 2 overviews the formal method to develop concurrent programs presented in [1, 2]. Section 3 reviews Java synchronization primitives. Section 4 presents our translation from coarse-grained solutions to fine-grained Java programs. The section also gives correctness argument, an example of the translation, and a remark about performance of translated programs.

## 2 Formal Approach for Developing Concurrent Programs

This section reviews a formal method to develop concurrent programs presented in [1, 2].

### 2.1 Global invariant

In a concurrent program, since there are many possible execution sequences of program statements, the key to developing a proof of such a program is to specify a global invariant that implies the safety property and holds at every *critical assertion* [1, 2] in the program.

For example, consider the readers/writers problem as follows: *Reader processes and writer processes access a shared buffer. At any point in time, the buffer can be accessed by either multiple readers or at most one writer.*

A simple way to specify this synchronization problem is to count the number of each kind of processes trying to access the buffer, and then to constrain the values of the counters. Let  $nr$  and  $nw$  be non-negative integers that respectively count the numbers of readers and writers accessing the buffer. Then, a global invariant  $RW$  that implies the safety property is:

$$RW : (nr = 0 \vee nw = 0) \wedge nw \leq 1$$

### 2.2 Coarse-grained solution

Let  $B$  be a boolean expression (called a guard) and  $S$  be a sequence of statements. The following two types of synchronization constructs are used in a coarse-grained solution<sup>2</sup>:

---

<sup>2</sup>Some synchronization problems assume a more general form of coarse grained solutions;  $\langle \mathbf{await} B_1; S_1; \mathbf{await} B_2; S_2; \dots; S_{n-1}; \mathbf{await} B_n; \rangle$ . However, many problems can be solved in the form of coarse-grained solutions presented here.

1.  $\langle S \rangle$ :  $S$  is executed atomically.
2.  $\langle \mathbf{await} B \rightarrow S \rangle$ : If  $B$  is true,  $S$  is executed atomically. If  $B$  is false, the executing process is delayed until a later time when  $B$  is true; at which point,  $S$  is executed atomically. No interleaving occurs between the final evaluation of  $B$  and the execution of  $S$ .

The following code presents a coarse-grained solution for the readers/writers problem.

Reader Processes	Writer Processes
a1 $\langle \mathbf{await} \ nw = 0 \rightarrow nr := nr + 1 \rangle$	b1 $\langle \mathbf{await} \ nw = 0 \ \mathbf{and} \ nr = 0 \rightarrow nw := nw + 1 \rangle$
a2            Read from the buffer	b2            Write to the buffer
a3 $\langle nr := nr - 1 \rangle$	b3 $\langle nw := nw - 1 \rangle$

Once the above coarse grained solution is obtained, we can construct a fine-grained semaphore or monitor program by applying the translation presented in [1, 2].

### 3 Java Synchronization Primitives

This section reviews the Java synchronization primitives.

#### 3.1 Preliminaries

Each Java object has a lock to form a critical section. If  $obj$  is a reference to some object,  $obj$  can be used to protect a critical section as follows:

```
e1    synchronized (obj) { critical section }
```

If all critical sections are in methods in the same object, we can use “*this*” for a synchronization object.

If the entire body of a method is a synchronized block on “*this*,” we have the following rule:

```
f1    type method_name ( $\dots$ ) {
f2        synchronized (this) {
f3            body of the method
f4        }
f5    }
```

may be written as:

```
g1    synchronized type method_name ( $\dots$ ) {
g2        body of method_name
g3    }
```

Thus, the monitor concept may be implemented in Java by adding **synchronized** for all methods in which only one thread should be executing at a time.

Unfortunately, each Java monitor has just one condition variable, which is associated with the object itself; all **wait**, **notify**, and **notifyAll** operations refer to it. The translation from coarse-grained solutions to fine-grained monitor programs presented in [1, 2] assumes multiple condition variables. It is still possible to associate all the  $\langle \mathbf{await} B_i \rightarrow S_i \rangle$  statements with the unique condition variable of “*this*” object. However, a signal on any condition variable must be translated to **notifyAll()**, and the resulting code will be inefficient.

### 3.2 Specific notification locks

Cargill devised a design pattern called a *specific notification* which uses objects somewhat like condition variables [3, 5]. Such objects are called *specific notification locks* [3] or *notification objects* [4].

Using a specific notification lock *obj*, two methods within the same object, say *method1* and *method2*, can synchronize with each other as follows, where *condition()* is assumed to be a **synchronized** method [4]:

<pre> in <i>method1</i> h1   <b>synchronized</b> (<i>obj</i>) { h2     <b>if</b> (! <i>condition</i>()) h3       <b>try</b> { <i>obj.wait</i>(); h4         } <b>catch</b>(InterruptedException e) { } h5     some statement h6   }</pre>	<pre> in <i>method2</i> h7   <b>synchronized</b> (<i>obj</i>) { h8     <b>if</b> (<i>condition</i>()) <i>obj.notify</i>() h9     ... h10  }</pre>
---	---

Do specific notification locks work exactly the same way as condition variables? In the above example, the **if** statement on line h2 and the following statement on line h5 are executed atomically only with respect to statements in synchronized blocks on *obj*, but not with respect to statements outside synchronized blocks on *obj*. Therefore, even though *condition()* is checked on line h8 before a thread waiting on *obj* is woken up by *obj.notify()*, the condition may no longer hold by the time the woken thread executes statement h5. This is true even if the **if** construct on line h2 is replaced by a **while** construct. If we make *method1* and *method2* **synchronized** methods, deadlock will result. This is because *obj.wait()* releases only a lock on *obj* but does not release a lock on *this*.

In a general monitor environment, if *method1* and *method2* are both monitor functions and *obj* is a condition variable, then statements h2 and h5 are executed atomically. Therefore, if the **if** construct on line h2 is replaced by a **while** construct, the condition checked at h2 holds at the beginning of h5<sup>3</sup>.

---

<sup>3</sup>If the monitor employs the Signal and Wait (SW) or the Signal and Urgent Wait (SU) discipline, because of the check on line h8, the condition holds at h5 even with the **if** construct on line h2 [1].

## 4 Translation from Coarse-Grained Solution to Java Program

Hartley gives an exercise problem to develop a `ConditionVariable` class, which may be used in **synchronized** methods [4]. Using the class, the existing translation can be used to obtain Java programs from coarse-grained solutions. However, the implementation of `ConditionVariable` is inefficient because a signal operation is implemented by `notifyAll()`. Our approach is to develop a direct translation from coarse-grained solutions to efficient Java monitor programs. The translation is an application of specific notification locks.

### 4.1 Translation

In a coarse-grained solution, there are two types of synchronization constructs:  $\langle \mathbf{await} B_i \rightarrow S_i \rangle$  and  $\langle S_j \rangle$ . For each appearance of such constructs in a coarse-grained solution, we define an independent method in a Java class.

1. For each  $\langle \mathbf{await} B_i \rightarrow S_i \rangle$ , define one public (non-**synchronized**) method and one private **synchronized** method, and declare one private specific notification lock (instance variable) of class *Object*. Let  $method_i$ ,  $checkBS_i$ , and  $o_i$  be the public method, the private **synchronized** method, and the specific notification lock, respectively.

We have the following declaration for  $o_i$ :

```
j1 private Object o_i = new Object();
```

Public method  $method_i$  is defined as:

```
k1 public void method_i() {
k2     synchronized (o_i) {
k3         while (! checkBS_i())
k4             try {
k5                 o_i.wait();
k6             } catch (InterruptedException e){}
k7     }
k8 }
```

Private **synchronized** method  $checkBS_i$  is defined as:

```
m1 private synchronized boolean checkBS_i() {
m2     if (B_i) {
m3         S_i; return true;
m4     } else return false;
m5 }
```

2. For each  $\langle S_j \rangle$ , define a public (non-**synchronized**) method, say  $method_j$ , as follows:

```

n1 public void  $method_j()$  {
n2   synchronized ( $this$ ) {
n3      $S_j$ ;
n4   }
n5 }
```

3. In each public method  $method_i$  and  $method_j$ , if execution of any statement may potentially change some guard  $B_k$  from *false* to *true*, add either of the following two statements at the end of the method (outside any **synchronized** block).

```

p1 synchronized ( $o_k$ ) { $o_k$ .notifyAll();}
p2 synchronized ( $o_k$ ) { $o_k$ .notify();},
```

where  $o_k$  is a specific notification lock associated with  $\langle \mathbf{await} B_k \rightarrow S_k \rangle$ . If more than one thread may leave the construct  $\langle \mathbf{await} B_k \rightarrow S_k \rangle$  when  $B_k$  becomes *true*, **notifyAll** should be issued; otherwise, **notify** should be issued.

Note that in the case of  $method_i$ , **synchronized** method  $checkBS_i$ , not  $method_i$ , may change  $B_k$ . However, a **notify** statement should be placed in  $method_i$ , not in  $checkBS_i$ . This is because  $checkBS_i$  is a **synchronized** method and executed within a **synchronized** block on  $o_i$ . Therefore, if another **synchronized** block on  $o_k$  is placed in  $checkBS_i$ , it may result in deadlock.

## 4.2 Correctness

We argue the correctness of a translated program with respect to preservation of a global invariant and deadlock freedom.

### Preservation of Global Invariant:

Let  $GI$  be a global invariant in the coarse-grained solution. First, note that in a translated program, assignment statements may be found only in  $S_i$  and  $S_j$  and they are executed inside synchronized blocks on  $this$  (lines m3 and n3).

[1] Translation of  $\langle \mathbf{await} B_i \rightarrow S_i \rangle$ : When a thread enters  $method_i$ , it immediately executes  $checkBS_i$ . Since  $checkBS_i$  is a **synchronized** method on  $this$ , it is executed atomically with respect to any assignment statement. If  $B_i$  does not hold when the thread enters  $checkBS_i$ ,  $checkBS_i$  returns *false* so that the thread waits on  $o_i$ .**wait**() and does not leave  $method_i$ . On the other hand, if  $B_i$  holds, the thread executes  $S_i$  and  $checkBS_i$  returns *true* to leave the **while** statement (line k3) and  $method_i$ . Since  $\{GI \wedge B_i\}S_i\{GI\}$  holds in the coarse-grained solution, it is clear that if  $GI$  holds before  $method_i$ , it holds after  $method_i$ .

[2] Translation of  $\langle S_j \rangle$ :  $\{GI\}S_j\{GI\}$  holds in the coarse-grained solution, and  $S_j$  is executed atomically with respect to any assignment statement in the translated program; therefore, if  $GI$  holds before  $method_j$ , it holds after  $method_j$ .

From [1] and [2],  $GI$  is a monitor invariant of the translated program.

### Deadlock Freedom:

[1] In  $method_i$ ,  $checkBS_i$  is called inside a **synchronized** block on  $o_i$ . This guarantees atomic execution of the statements  $checkBS_i$  (line k3) and  $o_i.\mathbf{wait}()$  (line k5) with respect to execution of  $o_i.\mathbf{notifyAll}()$  (line p1) (or  $o_i.\mathbf{notify}()$  (line p2)). Therefore, it is not possible to have a situation in which after one thread executes  $checkBS_i$  (which returns false) but before it executes  $o_i.\mathbf{wait}()$ , another thread executes  $o_i.\mathbf{notify}()$  (or  $o_i.\mathbf{notifyAll}()$ ). Note that if such a situation occurs, the notification signal ( $o_i.\mathbf{notify}()$ ) would be lost and a deadlock might result.

[2] In  $method_i$  and  $method_j$ , (a) execution of a **synchronized** block (on  $o_i$  (lines k2-k7) and on  $this$  (lines n2-n4), respectively) and (b) possible execution of  $o_k.\mathbf{notifyAll}()$  (line p1) or  $o_k.\mathbf{notify}()$  (line p2) are not atomic. Therefore, after a thread, say  $T_i$ , executes lines k2-k7 (or lines n2-n4) but before it executes line p1 (or p2), another thread, say  $T_j$ , may execute  $o_k.\mathbf{wait}()$  or  $o_k.\mathbf{notify}()$ . However, such an interleaving execution will not cause any problem.

[3] The nesting level of **synchronized** blocks is at most two (this occurs when the body of  $checkBS_i$  is executed). The order of nesting is always a **synchronized** block on a specific notification lock being outside and a **synchronized** block on  $this$  being inside. Furthermore,  $o_i.\mathbf{wait}()$  is executed within a sole **synchronized** block on  $o_i$ . Therefore, deadlock will never occur due to nested **synchronized** blocks.

## 4.3 Example

The following Java code is obtained by applying the above translation to the readers/writers coarse-grained solution described in Section 2:

```
class RWMonitor {
q1  private int nr = 0;
q2  private int nw = 0;
q3  private Object or = new Object();
q4  private Object ow = new Object();
q5  public void startRead() {
q6      synchronized (or) {
a7          while (!checkRead())
q8              try {or.wait();
q9                  } catch(InterruptedException e){}
q10     }
q11 }
q12 private synchronized boolean checkRead() {
q13     if (nw == 0) {
q14         nr := nr + 1; return true;
q15     } else return false;
q16 }
```

```

q17 public void finishRead() {
q18     synchronized (this) {
q18         nr := nr - 1;
q20     }
q21     synchronized (ow) {ow.notify(); }
q22 }
q23 public void startWrite() {
q24     synchronized (ow) {
q25         while (!checkWrite())
q26             try {ow.wait();
q27                 } catch(InterruptedException e){}
q28     }
q29 }
q30 private synchronized boolean checkWrite() {
q31     if ((nw == 0) && (nr == 0)) {
q32         nw := nw + 1; return true;
q33     } else return false;
q34 }
q35 public void finishWrite() {
q36     synchronized (this) {
q37         nw := nw - 1;
q38     }
q39     synchronized (or) {or.notifyAll(); }
q40     synchronized (ow) {ow.notify(); }
q41 }
}

```

Since `finishRead()` decrements variable `nr`, the writers' condition to proceed `((nw == 0) && (nr == 0))` may become true. Therefore, `ow.notify()` is added on line q21 in `finishRead()`. Similarly, the decrement of `nw` in function `checkWrite()` may change both the readers' condition `((nw == 0))` and the writers' condition to true. Therefore, `or.notifyAll()` (on line q39) and `ow.notify()` (on line q40) are added in `finishWrite()`. Note that since multiple readers may become active, `notifyAll()` is called on `or`. On the other hand, at most one writer can become active; therefore, `notify()` is called on `ow`.

#### 4.4 Performance

Given a coarse-grained solution, at least three approaches exist to obtain Java programs:

[ 1 ] apply the translation to a monitor program presented in [1, 2],

[ 1-a ] using only one condition variable (all the threads sleep on the condition variable



associated with “*this*”),

[1-b] using Hartley’s condition variable class, and

[2] apply the translation presented in this section.

We have the following comparison:

1. When only one process needs to be awoken, programs obtained by [1-a] and [1-b] always issue **notifyAll**, whereas a program obtained by [2] uses **notify**.
2. When one condition may become true, a program obtained by [1-a] wakes up all waiting threads, whereas programs obtained by [1-b] and [2] wake up only threads waiting on the condition.

From the above comparison, it is expected that programs obtained by [2] will outperform programs obtained by [1-a] and [1-b] in many applications.

## 5 Conclusion

A formal and systematic method to develop concurrent programs is presented in [1, 2]. In the method, we first specify a global invariant. Then, we develop a coarse-grained solution in which the global invariant holds at every critical assertion. Finally, we translate the coarse-grained solution to a fine-grained program.

Since Java synchronization primitives do not allow multiple condition variables within a monitor, the above translation cannot be used to produce efficient Java programs. This paper presented a translation from a coarse-grained solution to a fine-grained Java program. The translation uses specific notification locks. With the translation, we can use the formal method to develop efficient Java concurrent programs.

## Acknowledgment

I would like to thank the anonymous referees and the communicating editor, Professor G.R. Andrews, for their valuable and constructive comments.

## References

- [1] G.R. Andrews. *Concurrent Programming, Principles and Practice*. Benjamin/Cummings Publishing Co., 1991.
- [2] A.J. Bernstein and P.M. Lewis. *Concurrency in Programming and Database Systems*. Jones and Bartlett, 1993.
- [3] T. Cargill. Specific notification for java thread synchronization. In *International Conference on Pattern Languages of Programming*, 1996. <http://www.sni.net/~cargill/jgf/9809-/SpecificNotification.html>.

- [4] S.J. Hartley. *Concurrent Programming - The Java Programming Language*. Oxford University Press, 1998.
- [5] D. Lea. *Concurrent Programming in Java, Design Principles and Patterns*. Addison Wesley Publishing Co., 1997.