Teaching predicates and invariants on shared data structures in concurrent programming

Stein Gjessing  
Department of informatics,  
University of Oslo  
steing@ifi.uio.no

Arne Maus  
Department of informatics,  
University of Oslo  
arne@ifi.uio.no

Abstract
In concurrent programing, threads may communicate via data structures that can be embedded in shared objects or monitors. In this paper we outline the basis for a short module that can be used to teach better programming of such monitors by emphasizing logical statements, called predicates, on the data structures inside these shared objects. We base the module on the general concepts of object oriented programming, where the (usually) private data structure is manipulated by public methods, and where an invariant predicate holds before an after the execution of each method. An important challenge when writing a monitor is to program correct synchronization, and the main contribution of this paper is to show how to teach the students to use predicates to reason about, and make code that performs correct waiting and signaling inside monitors.

Categories and Subject Descriptors D.3.3 [Programming Techniques]: Concurrent Programming, Object Oriented Programming; D.2.4 [Software Engineering]: Software/Program Verification

General Terms Algorithms, Design, Documentation, Languages, Theory, Verification,

Keywords Threads, Monitors, Predicates, Invariants, Synchronization, Didactics

1. Introduction
In a second semester object oriented programming course we teach students concurrent programming based on classes and objects, concepts that they already know. They learn to program threads, and to let threads communicate and collaborate using shared objects or monitors [1] (The term monitor is a loaded word, but in a parallel programing context the term monitor was coined by C.A.R. Hoare in 1974 as a shared object.).

Students learn to reason about the state of the data structure of the program they are writing, using logical statements or predicates. When they write classes, they have to reason about the internal (usually private) data structures in the objects, and how the public methods change the state of these data structures. The consistency of the data structures inside an object can be expressed by an invariant predicate, and all public methods have to preserve this invariant. It may temporarily be broken during the execution of a method, but it will be restored before the method terminates. We emphasize, however, that such a proof (that an invariant is maintained by all methods) is very short of a full proof of the class, e.g. it does not express that each method performs the task it is intended to. A full verification of a class is usually a too lengthy, complicated and also error prone activity to be taught in a second semester course. The use of invariants is a kind of ‘formalism light’ suitable for the kind of problems treated here, and analogous to tabular specification of methods [3]. When threads and monitors are introduced, we thus build on this knowledge about invariants in classes.

2. A restaurant table for a sequential program
The archetypical example of a monitor is a shared buffer where producers can put data and consumers can get data. To make the situation more memorable for the students, instead of a generic buffer we use an example in the form of a limited sized counter or a serving table in a restaurant, where dishes (single plates) are placed (under heating lights) after being produced by chefs, waiting for the waiters to be served (fig. 1). In order to make the programs in this paper as small as possible, we do not represent the individual dishes. It would be very easy to add this feature to the programs shown.

With this simplification we only need to have a data structure with one variable, called nosOnTable, in the object. Based on this specification of the data structure, the students easily write down a simple invariant:

\[ 0 \leq \text{nosOnTable} \leq \text{TABLE\_SIZE} \]

![Figure 1. Three producer threads (chefs) and three consumer threads (waiters) performing respectively put and get operations on a shared table implemented as a monitor.](image-url)
where \( \text{TABLE\_SIZE} \) is the maximum number of plates that can simultaneously be placed on the heating table, i.e. the buffer size.

However, we first ask the students to write a class for our restaurant heating table that is to be used in a sequential program. Then they produce something like the program shown in fig. 2. Since the individual dishes are not represented, both methods returns a boolean value telling if the operation was successful or not.

```java
class HeatingTable {
    final int \text{TABLE\_SIZE} = 10;
    int nosOnTable = 0;
    // Invariant: \( 0 \leq \text{nosOnTable} \leq \text{TABLE\_SIZE} \)

    boolean putPlate() {
        if (nosOnTable == \text{TABLE\_SIZE}) {
            return false;
        }
        // \( 0 \leq \text{nosOnTable} < \text{TABLE\_SIZE} \)
        nosOnTable++;
        // \( 0 < \text{nosOnTable} \leq \text{TABLE\_SIZE} \)
        return true;
    }

    boolean getPlate() {
        if (nosOnTable == 0) return false;
        // \( 0 < \text{nosOnTable} \leq \text{TABLE\_SIZE} \)
        nosOnTable--;
        // \( 0 \leq \text{nosOnTable} < \text{TABLE\_SIZE} \)
        return true;
    }
}
```

**Figure 2.** Program showing a heating table that can store 10 plates. To be used in a sequential program.

The invariant obviously holds initially (\( \text{nosOnTable} = 0 \)), and we have to make sure that all methods preserve this invariant.

In the putPlate method the invariant can be broken by the execution of the operation \( \text{nosOnTable}++ \). If we can establish that \( \text{nosOnTable} < \text{TABLE\_SIZE} \) before this operation, we know it can be executed without breaking the invariant. Hence we test whether \( \text{nosOnTable} \) already has reached \( \text{TABLE\_SIZE} \). If it has, we exit the method without doing anything, and we return “false” in order to show this to the calling program.

In fig. 2 we see the two comments (the predicates) that describes the state of the monitor respectively before and after this increment operation in the putPlate method. The argument why the method getPlate in fig. 2 also preserves the invariant, is symmetric. We have not written down the full invariant in front of (nor behind) all the methods.

3. A first concurrent solution

In a concurrent program, where more threads operate on shared objects, if the execution of a method seems to break the object’s invariant, the method does not have to return immediately with mission unaccomplished. Instead the thread’s execution of the method can wait a while, and the thread can hope that another thread will call a method in the object, and that the execution of this method will change the state of the data structure inside the object in such a way that the waiting thread now can finalized the execution of its method without breaking the invariant. Hence we teach the students that they can program the operations so that they may wait for the state to be changed (by one of the other threads), and when the wait is finished, the state is hopefully (ideally always) so that the intended method can be executed without breaking the invariant. Before a wait statement the students have to write down in the program a predicate that show that the rest of the method can not be executed without breaking the invariant.

When the wait is completed, they have to write a predicate that shows that the method may now be completed while preserving the invariant.

We then also teach them that methods must notify other waiting threads, when the state of the data structure has changed so that these other threads may not have to wait any more. The predicate that indicates why a notify operation should be executed must be inserted before the notify.

When teaching this, we first just ask the students to turn the program in fig. 2 into a monitor. A chef that is making a dish when the table is full, then have to wait inside the putPlate method, and a waiter that wants to serve a dish has to wait inside the getPlate method if the table is empty. In this paper, we do not show this first simple monitor.

It is then easy to write a program with chef threads and waiter threads that run a restaurant indefinitely. If, however, that task becomes to produce and serve a given number of dishes, and also terminate all threads in an orderly fashion, the monitor becomes a little more complex.

In addition to the \( \text{TABLE\_SIZE} \), we then have the number of plates to be produced, called \( \text{NOS\_TO\_BE\_MADE} \). A variable in the object, called \( \text{nosProduced} \), then counts how many plates have been made and put on the table so far. Hence the invariant now consists of two parts that must both hold:

\[
0 \leq \text{nosOnTable} \leq \text{TABLE\_SIZE}
\]

\[
0 \leq \text{nosProduced} \leq \text{NOS\_TO\_BE\_MADE}
\]

In this problem we hence have four ‘balls in the air’ - the three inequalities stated in the two invariants and the demand on normal terminations on all but the main thread when the job is done, all plates made and served.

We also specify that when \( \text{NOS\_TO\_BE\_MADE} \) dishes have been produced and placed on the table, the putPlate method shall return “false”, so that the chefs may terminate. Likewise the method getPlate shall return “false” if there is nothing more to serve. Otherwise the methods return true. As we have already pointed out, in order to simplify the program, the dishes are not represented in the programs in this article. An obvious way to do this is to let putPlate have a dish or a plate as a parameter, and let getPlate return a plate instead of a boolean. Instead of a “false” return one would then get a “null” return when there are no more plates to serve.

When students run programs with threads and monitors, it is very important to visualize the execution sequence inside the monitor. Hence we tell them to write output that shows how threads are sequenced inside the monitor, and when they wait and when they wake up. In order to do this it is advantageous to bring along a thread identification when a monitor method is called. In fig. 3, the two methods putPlate and getPlate both have a parameter that is a reference to the calling thread, and statements are given that will produce informative output when the program is run. The students can also play with different numbers of chefs, waiters and dishes to be produced, and they can change the time the chefs and the waiters are sleeping before they access the monitor again. The main program file that includes the chef threads and the waiter threads are given in the Appendix.
class HeatingTable {  // A monitor
    int nosOnTable = 0;  // Invariants holds initially
    int nosProduced = 0;  // Invariants holds initially
    final int TABLE_SIZE;
    final int NOS_TO_BE_MADE;

    // INVARIANTS:
    // 1) 0 <= nosOnTable <= TABLE_SIZE
    // 2) nosProduced <= NOS_TO_BE_MADE
    HeatingTable(int maxOn, int max) {
        TABLE_SIZE = maxOn;
        NOS_TO_BE_MADE = max;
    }
    // Invariants holds initially as long as
    // parameters are none-negative
    public synchronized boolean getPlate(Waiter w) {
        while (nosOnTable == 0 && nosProduced < NOS_TO_BE_MADE) {
            try {  // The while test holds here meaning
                // that a chef should but can not
                // make a dish, because the table is full
                System.out.println("Waiter "+w.ind+" WAITING to serve a dish");
                w.wait();
            } catch (InterruptedException e) {
                notifyAll();
            }
        } catch (InterruptedException e) {
            return false;
        } else { return true; }
    }

    // Hence OK to decrease nosOnTable:
    public synchronized boolean putPlate(Chef c) {
        while (nosOnTable == TABLE_SIZE && nosProduced < NOS_TO_BE_MADE) {
            try {  // The while test holds here meaning
                // that a chef should but can not
                // make a dish, because the table is full
                System.out.println("Chef "+c.ind+" WAITING to make a dish");
                c.wait();
            } catch (InterruptedException e) {
                notifyAll();
            }
        } catch (InterruptedException e) {
            return false;
        } else { return true; }
    }

    public synchronized void notifyAll() {
        // Must wake up a sleeping chef:
        notifyAll();
    }

    // Must wake up a sleeping waiter:
    public synchronized void notifyWaiters() {
        notifyAll();
    }

    // Must wake up a sleeping chef:
    public synchronized void notifyChefs() {
        notifyAll();
    }

    Figure 3. A bounded buffer table with a given number of dishes to be produced. This example uses Java built in
    waiting and signaling.

    Notice that we have decorated the program in fig. 3 with predicates telling the state of the data structure at this point in the pro-
    gram. These predicates are the essence of our message to the students, and what we want to teach them to use in order for them
    to be able to write programs that they can convince themselves and others are correct.

    Initially we ask the students to write down predicates as they have learned for objects with private data structures and methods
    that manipulate this structure in sequential programming. I.e. they find and write the invariant down at the start of all methods, and
    then they must show that it holds for all possible terminations of the methods. We also want them to write down the usual facts
    that the while test holds immediately inside the while block, and that the negation holds immediately after the while statement (and
    similar for if-tests). The program in fig. 3 could have been decorated with even more of these predicates.

    Then we concentrate on the particular aspects of predicates inside monitor methods. The most important predicates in fig. 3 are
    the ones in front of the wait()-statements. These predicates state the fact that the threads has to wait because the state of the moni-
    tor is such that it is currently impossible to complete the execution of the methods without violating the invariant.

    When it is safe to complete the execution of the methods, the predicates before the increment and decrement operations of the
two object variables ensure this. These predicates tell that it is safe to execute these operations without violating the invariant. The
predicates after the while loops, and the if-tests ensure these predicates.

    In native Java there is, for each shared object, one queue of threads waiting, after having executed any wait()-operation in any
method. It is possible to reason about which (textually) wait-operations the threads in this queue have performed. However, such
predicates that include reasoning about the queue of waiting threads, is complex, and only much later in the curriculum we tell
the students about the types of invariants that are needed in this case.
Because there is only one queue of waiting threads when using native Java synchronization, we can not know whether a notify()-operation will restart a thread waiting in the putPlate method or in the getPlate method. A chef that has just put a plate on the table must be sure that it is waking up a waiter. If it performs a simple notify() operation, it may wake up another chef instead. Hence in order to be sure that the putPlate method notifies a waiter waiting in the getPlate method it has to use the notifyAll() operation. This is of course also the case in the getPlate method where a waiter notifies a chef waiting in the putPlate method by a notifyAll() operation. Hence, by using notifyAll(), and reasoning about the invariant and the other predicates included in the program of fig. 3, it is possible for the students to convince themselves and others that the program is correct.

Obviously it is not an ideal solution to wake up all waiting threads when only one plate has been put on (or taken from) the table. What we really want is one waiting queue for each different state in which the execution has to wait before a certain operation can be executed without breaking the invariant. A high level view of what we want is shown in fig. 4.

![Figure 4. A monitor with one queue for each condition that a thread can wait on. In this case there is one queue waiting for the table to be not full, and one queue waiting for the table to be not empty.](image)

4. A second parallel solution using Conditions

Fortunately Doug Lea has made an addition to the Java library [2] that makes it possible to use two (or more) queues when methods are waiting for different predicates or conditions to hold. This addition, java.Concurrent, offers the ReentrantLock class. Then we can specify different predicates (or conditions) that threads have to wait for to hold. The ReentrantLock class provides a fabric for Condition variables. Hence, for each different predicate the program need to wait for to hold, we define a Condition variable, and all threads waiting for the same predicate to hold may now be placed in the same queue (by an await() operation), and this is exactly what we want. When one thread has changed the state so that another thread may now seize waiting, we can signal() this other thread to be restarted.

```java
import java.util.concurrent.locks.*;

class HeatingTable { // Monitor
  int nosOnTable = 0; // Invariant holds initially
  int int = 0; // Invariant holds initially
  final Condition notEmpty = lock.newCondition();
  final Condition notFull  = lock.newCondition();
  final Lock lock = new ReentrantLock();

  // Invariants
  // 1) 0 <= nosOnTable <= TABLE_SIZE
  // 2) nosProduced < NOS_TO_BE_MADE

  void putPlate(Chef c) throws InterruptedException {
    try {
      lock.lock();
      while (nosOnTable == MAX_ON_TABLE)
        notFull.await();
      // The while test holds here
      System.out.println("Chef "+c.ind+
        " WAITING to make a dish");
      notFull.await();
    }
    // one or both of the loop conditions are now false
    if (nosProduced < NOS_TO_BE_MADE) {
      // nosOnTable < TABLE_SIZE
      nosProduced++; c.dishNum = nosProduced;
      nosOnTable++;
      System.out.println("Chef number "+c.ind+
        " makes plate num:" + c.dishNum );
      // nosOnTable > 0
      notEmpty.signal();
      if (nosProduced == NOS_TO_BE_MADE) {
        // Last plate is produced
        notFull.signalAll(); // tell Chefs to stop
        // waiting and terminate
        notFull.signalAll(); // Not more to wait for,
        // Waiters serve last plate(s)
        return false;
      }
      return true;
    } else { return false; }
  }
  // end put
}

class HeatingTable(int maxOn, int max) {
  // n/ O_3
  TABLE_SIZE = maxOn;
  NOS_TO_BE_MADE = max;
  }
  // Invariants holds initially as long as
  // the parameters are non-negative

  public boolean putPlate(Chef c) throws InterruptedException {
    lock.lock();
    try {
      while (nosOnTable == MAX_ON_TABLE)
        notFull.await();
      // The while test holds here
      System.out.println("Chef "+c.ind+
        " WAITING to make a dish");
      notFull.await();
    }
    // one or both of the loop conditions are now false
    if (nosProduced < NOS_TO_BE_MADE) {
      // nosOnTable < TABLE_SIZE
      nosProduced++; c.dishNum = nosProduced;
      nosOnTable++;
      System.out.println("Chef number "+c.ind+
        " makes plate num:" + c.dishNum );
      // nosOnTable > 0
      notEmpty.signal();
      if (nosProduced == NOS_TO_BE_MADE) {
        // Last plate is produced
        notFull.signalAll(); // tell Chefs to stop
        // waiting and terminate
        notFull.signalAll(); // Not more to wait for,
        // Waiters serve last plate(s)
        return false;
      }
      return true;
    } else { return false; }
  }
  // end put
}
```

```java
public boolean getPlate(Waiter w)
  throws InterruptedException {
    lock.lock();
    try {
      while (nosOnTable == 0 &&
        nosProduced < NOS_TO_BE_MADE ) {
        // The while test holds here
        System.out.println("Waiter "+ w.ind +
          " WAITING to serve a dish");
        notEmpty.await();
      }
      // Get a plate
      if (nosOnTable > 0) {
        try {
          System.out.println("Waiter "+ w.ind +
            " takes plate num:" + nosOnTable);
        } catch (InterruptedException e) {
          System.out.println("Disrupted while executing");
        }
        notFull.signalAll();
      }
    }
    finally {
      lock.unlock();
    }
  }
  // end getPlate
```
5. Conclusion

We have outlined the content of a short module that can be used to teach students concurrent programming by Java threads that communicate via shared data structures encapsulated in monitors, as first advocated by C.A.R. Hoare [1]. One sequential and two slightly different parallel solutions to a bounded buffer with a maximum number to be produced are used as examples.

The main contribution of this paper is to show how reasoning about the state of an object is important, especially in concurrent programming. Our students have to reason about (private) data structures in objects from when they first learn to program such objects. They learn that in any object with a private data structure and public methods, invariants must be formulated and documented. When they learn about threads and shared data structures, we extend this knowledge to the use of invariants and predicates on the monitor data structure. This helps the students to formulate predicates and reason about when to wait and when to signal.

The two examples of a monitor in this article solve the same problem, and have the same invariant, but one example performs the task much more elegantly than the other. Based on the predicates that hold when threads wait and threads signal, we show that the built in wait() and notify() operations embedded in the Java programming language are not good enough tools. We also show that the library java.Concurrent written by Dough Lea [2], comes much closer to include tools that fits the way we believe it is natural and correct to express the problem using invariants and predicates.

In concurrent programming correct reasoning about the state of (shared) objects are even more important than in sequential programs, because incorrect concurrent programs are even harder to debug than sequential programs, and an erroneous concurrent program may compute correctly for a long time before it fails.

Appendix

class Restaurant {
    HeatingTable table;
    Restaurant(String[] args) {
        table = new HeatingTable(3,Integer.parseInt(args[0]));
        int numChefs = Integer.parseInt(args[1]);
        int numWaiters = Integer.parseInt(args[2]);
        if (numWaiters > 0) {
            System.out.println("<Waiter number " + w.ind + ", serves plate num:" + (nosProduced-nosOnTable));
        }
        // nosOnTable == 0 || nosProduced == NOS_TO_BE_MADE
        return false;
    }
    finally {
        lock.unlock();
    }
} // end Restaurant

class Chef extends Thread {
    HeatingTable sharedTable;
    int ind;
    int dishNum;
}
Chef(HeatingTable shared, int ind) {
    sharedTable = shared;
    this.ind = ind;
}

public void run() {
    try {
        while (sharedTable.putPlate(this)) {
            sleep((long) (1000 * Math.random()));
        }
    } catch (InterruptedException e) {} // Chef has finished
    System.out.println("Chef number finished: " + ind);
} // end Chef

class Waiter extends Thread {
    HeatingTable sharedTable;
    int ind;

    Waiter(HeatingTable shared, int ind) {
        sharedTable = shared;
        this.ind = ind;
    }

    public void run() {
        try {
            while ((sharedTable.getPlate(this))) {
                // serve the plate
                sleep((long) (1000 * Math.random()));
            }
        } catch (InterruptedException e) {} // This Waiter has finished
        System.out.println("Waiter number finished: " + ind);
    } // end Waiter
} // end Waiter

Figure 6. The Restaurant class, the Chef class and the Waiter class that all can use the HeatingTable monitor.

References