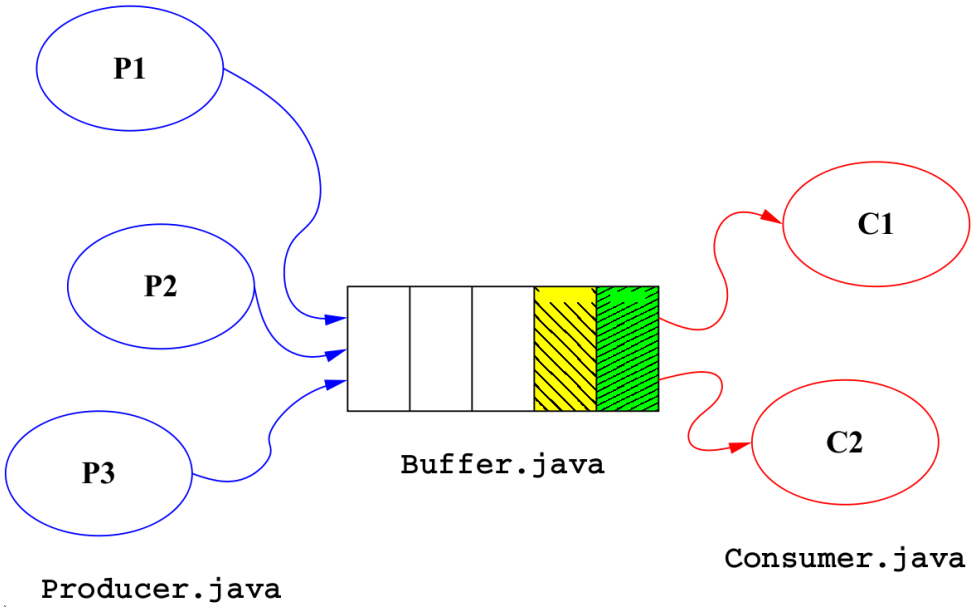
An Example of Debugging Java with a Model Checker

A Multi-Threaded Program (in Java)

To write concurrent programs is hard. To a large extend, this difficulty comes from a lack of good testing and debugging strategies. We also make mistakes when we write sequential programs, but we are better equipped to detect and correct them.

What makes dealing with concurrent systems more difficult is their inherent nondeterminism: the same program, on the same input, can behave differently from run to run. This results in bugs that occur extremely rarely and are not reproducible. Such bugs can be real nightmares: There is something wrong with your system, you know it, you have seen it, but until you can make it happen again, you don’t know how to approach the problem.



To illustrate this difficulty, consider the classic example of a shared buffer in a system of producers and consumers. Such a buffer holds the data created by producing threads until they are retrieved by consuming threads. More importantly, the buffer acts as a synchronizer, blocking and suspending threads when there is nothing for them to do. When the buffer is empty, any consuming thread needs to be blocked until there is data in the buffer. Conversely, if the buffer has finite capacity, producing threads must be suspended when the buffer is full.

```
1  public class Buffer<E> {
2
3    public final int capacity;
4    private final E[] store;
5    private int head, tail, size;
6
7    @SuppressWarnings("unchecked")
8    public Buffer (int capacity) {
9      this.capacity = capacity;
10     this.store = (E[])new Object[capacity];
11   }
12   private int next (int x) {
13     return (x + 1) % store.length;
14   }
15   public synchronized void put (E e) throws InterruptedException {
16     while (isFull())
17       wait();
18     notify();
19     store[tail] = e;
20     tail = next(tail);
21     size++;
22   }
23   public synchronized E get () throws InterruptedException {
24     while (isEmpty())
25       wait();
26     notify();
27     E e = store[head];
28     store[head] = null; // for GC
29     head = next(head);
30     size--;
31     return e;
32   }
33   public synchronized boolean isFull () {
34     return size == capacity;
35   }
36   public synchronized boolean isEmpty () {
37     return size == 0;
38   }
39 }
```

An unfortunate programmer once wrote the `Buffer` (Downloads/Buffer.java) class displayed above. The buffer is implemented as a circular list. Method `get` suspends the calling thread until the buffer is nonempty. The thread then removes one object from the buffer and does a single call to `notify` to (potentially) unlock a thread blocked on `wait`. Method `put` is symmetric. Both methods are `synchronized` in order that their execution appears atomic to all threads. In particular, a thread cannot see intermediate states while the buffer is being modified by another thread. (This is also why the call to `notify` can take place before the buffer is actually modified.)

The basic mechanism used here to suspend a Java thread is the method `Object.wait`, which suspends a thread unconditionally. Every thread that calls `x.wait()` is suspended and placed into the *wait set* of object `x` (every Java object has a wait set). A call to `x.notify()` selects a thread in the wait set of `x`, if any, to resume execution. A call to `x.notifyAll()` allows all the threads in the wait set of `x` to resume execution. Calls to `notify` or `notifyAll` have no effect if the wait set is empty. (This is an overly simplified description of these methods; see the Java Language Specification (http://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.2) for details.)

The strategy used in this buffer implementation is to use the wait set of the buffer object (i.e., calling `wait()` on `this`) and to allow one blocked thread to resume execution after each buffer modification. The idea is that there is no reason, after placing *one* object in the buffer (or after removing one), to notify *more than one* thread.

A Bug!

One day, the author of the `Buffer` class gets a phone call from one of his customers to inform him that the system he wrote is completely frozen and nothing is happening and everything just hangs and money is being lost by the millions and this is just unacceptable.

After solving the immediate problem the way programmers solve every problem (that is, by pulling the plug and restarting the whole thing) and profusely apologizing for any inconvenience caused to his customer, our author decides that his job would be more secure if he could offer guarantees that this will not happen again in the future. So, he embarks upon a debugging mission.

```
37    public synchronized void put (E e) throws InterruptedException {
38      String name = Thread.currentThread().getName();
39      while (isFull()) {
40        logger.info(String.format("buffer full; %s waits", name));
41        waitSetSize++;
42        wait();
43        logger.info(String.format("%s is notified", name));
44      }
45      logger.info(String.format("%s successfully puts", name));
46      notify();
47      if (waitSetSize > 0)
48        waitSetSize--;
49      logger.fine(String.format("wait set size is %d", waitSetSize));
50      store[tail] = e;
51      tail = next(tail);
52      size++;
53      lastChange = System.currentTimeMillis();
54    }
```

Our fellow is a little old fashioned (he does his debugging using print statements) but not completely dumb. He guesses (correctly) that his buffer implementation can cause threads to *deadlock*. So, he decides to write an annotated version (Downloads/AnnotatedBuffer.java) of his buffer so he can study a trace of an execution after a deadlock.

```
94      class Producer extends Thread {
95        public Producer (int n) {
96          super("producer-"+n);
97        }
98        public void run () {
99          String name = getName();
100         try {
101           while (true) {
102             sleep(rand.nextInt(sleepProd));
103             buffer.put(name);
104           }
105         } catch (InterruptedException e) {
106           return;
107         }
108       }
109     }
```

He then writes a program to create producing and consuming threads in an attempt to reproduce the deadlock that almost ruined his most important customer. He simulates the processing tasks of the real system (the time it takes to create an object before it is added to the buffer and the time it takes to process an object after it is removed from the buffer) using `Thread.sleep`, which suspends a thread for a specified amount of time (as if the thread was busy doing the work that is being simulated).

Using `Thread.sleep` has two major benefits. First, suspended threads don't use CPU resources, so the customer's large system can be simulated on a cheap laptop (which is all our writer of buggy code can afford). Second, the `sleep` method can be fed random timing values to simulate a large class of short, medium and long processing tasks. This is essential

because the system is nondeterministic and only *some* combination of timing values can produce the desired deadlock. So, it is imperative to use as many different combinations of values as possible, for which pseudo-random numbers are suitable (though not great, but that is a discussion for another day).

```
136       long time = System.currentTimeMillis();
137       while (true) {
138         Thread.sleep(60000); // check for deadlock every minute
139         synchronized (buffer) {
140           if (buffer.waitSetSize == threadCount) {
141             logger.severe(String.format
142                           ("DEADLOCK after %d messages and %.1f seconds!",
143                            buffer.msgCount, (buffer.lastChange - time)/ 1e3));
144             for (Thread t : participants)
145               t.interrupt();
146             return;
147           }
148         }
149       }
```

Finally, our programmer writes a main program that monitors the producer-consumer application. Every minute, this program checks the size of the wait set of the buffer. If it is equal to the total number of threads, it means that *all* the threads are waiting and no thread is left running to notify them, which is the deadlock situation observed by the customer.

Relying on Luck

All that is left is to run the program until the deadlock happens and then to analyze the resulting trace to understand what went wrong. Leaving aside the issue of analyzing the trace (which can be tricky because print statements from different threads tend to be interleaved in confusing ways) and of solving the actual problem after it is understood, the first and biggest challenge is to obtain a suitable trace, that is, to make the deadlock happen.

In his first attempt, our brave programmer starts his program as follows:

```
java AnnotatedBuffer 10 5 5 50 50
```

This creates a buffer of capacity 10, 5 producers, 5 consumers, and simulates the producing and consuming tasks with random timings between 0 and 50 milliseconds.

After a few hours—and because the fans in his laptop are starting to produce a foul smelling smoke—he gives up and decides to try a different combination of parameters. After many attempts, a combination of parameters finally produces a deadlock in less than two hours, along with a trace:

```
java AnnotatedBuffer 5 10 3 10 3
20:53:45.695: DEADLOCK after 13963562 messages and 6477.4 seconds!
```

The trace, however, is hard to follow, in part because there are delays (and extra messages) between the moment a thread is notify and the moment it prints *“I’m being notified”*, but also because there are 13 threads to follow.

Since, as Einstein would put it, CPU cycles are cheaper than grey matter, our programmer decides to keep looking for a deadlock that would involve fewer threads. He finally gets one which, as it happens, is the smallest non trivial (buffer capacity, number of producers and number of consumers all larger than one) deadlock case for this system:

```
java AnnotatedBuffer 2 3 2 3 2
01:35:39.047: DEADLOCK after 3970423422 messages and 1576335.9 seconds!
```

Note that this is almost 4 billion messages and more than 18 days running! The output comes from an actual run on an 8-core machine (our programmer having given up on using his laptop).

The analysis of the trace reveals the flaw in the buffer implementation: All the threads (producers and consumers) are waiting in the *same* wait set. When method `get` calls `notify`, the intent is to notify a *producer* that a slot has been made available in the buffer. But this call can sometimes notify a *consumer* instead and, if this happens a few times, it leads to a

deadlock.

Applying Formal Methods

After a quick fix to his code—replacing `notify` with `notifyAll`, thus bringing back correctness at the cost of inefficiency—our relieved programmer thinks he can start to relax. But he can't. A question keeps haunting him: What if this happens to a more complex system? What if he is not so lucky next time? What if the next bug only happens once a month? Once a year? What if the particular timing that produces the bug in deployed code cannot be reproduced in his testing settings?

So, our programmers decides to enroll in CS–745/845 (teaching-cs745_845.html), an excellent class in formal methods offered at the University of New Hampshire, where he is exposed to alternative techniques to track tricky bugs in concurrent software. He learns about TLA$^+$, the *Temporal Logic of Actions* (http://research.microsoft.com/people/lamport/tla/tla.html), and `TLC`, a model checking tool for TLA$^+$. He decides that notations and tools like that could help him in his job and starts to practice using them on his bounded buffer problem.

The central algorithm of the bounded buffer can be modeled as a TLA$^+$ module:

```
EXTENDS Naturals, Sequences

CONSTANTS Producers,   (* the (nonempty) set of producers                    *)
          Consumers,   (* the (nonempty) set of consumers                    *)
          BufCapacity, (* the maximum number of messages in the bounded buffer *)
          Data         (* the set of values that can be produced and/or consumed *)

ASSUME /\ Producers # {}                    (* at least one producer *)
       /\ Consumers # {}                    (* at least one consumer *)
       /\ Producers \intersect Consumers = {} (* no thread is both consumer and producer *)
       /\ BufCapacity > 0                   (* buffer capacity is at least 1 *)
       /\ Data # {}                         (* the type of data is nonenpty *)
...
```

TLA$^+$ is a mathematical notation based on set theory and uses mathematical symbols rather heavily (and very few "keywords"). For this reason, it is much more readable in its pretty-printed form than in its raw ASCII source. So, we will continue the description of the `Buffer` module in pretty-printed form.

MODULE *Buffer*

This module simulates a producer-consumer example as it could be written using *Java* threads. In particular, we want to demonstrate the risk of deadlock when producers and consumers wait on the same object.

EXTENDS *Naturals*, *Sequences*

CONSTANTS
- *Producers*, the (nonempty) set of producers
- *Consumers*, the (nonempty) set of consumers
- *BufCapacity*, the maximum number of messages in the bounded buffer
- *Data* the set of values that can be produced and/or consumed

ASSUME
- $\land\ Producers \neq \{\}$ at least one producer
- $\land\ Consumers \neq \{\}$ at least one consumer
- $\land\ Producers \cap Consumers = \{\}$ no thread is both consumer and producer
- $\land\ BufCapacity > 0$ buffer capacity is at least 1
- $\land\ Data \neq \{\}$ the type of data is nonenpty

The TLA$^+$ module begins with a few constants, which act as parameters: a set of producing threads, a set of consuming threads, a buffer capacity and a data type (think of *Data* as the type parameter `E` in the java class). For the module to make any sense, we need to assume that the sets of threads are disjoint and nonempty, the buffer capacity is at least one and there is at least one piece of data that can be sent through the buffer.

$$
\begin{array}{l}
\textsc{variables}\ \mathit{buffer}, \quad \text{the buffer, as a sequence of objects} \\
\qquad\qquad \mathit{waitSet} \quad \text{the wait set, as a set of threads} \\
\\
\mathit{Participants} \triangleq \mathit{Producers} \cup \mathit{Consumers} \\
\mathit{RunningThreads} \triangleq \mathit{Participants} \setminus \mathit{waitSet} \\
\\
\mathit{TypeInv} \triangleq \land\ \mathit{buffer} \in \mathit{Seq}(\mathit{Data}) \\
\qquad\qquad\quad \land\ \mathit{Len}(\mathit{buffer}) \in 0 \mathinner{..} \mathit{BufCapacity} \\
\qquad\qquad\quad \land\ \mathit{waitSet} \subseteq \mathit{Participants} \\
\\
\mathit{Notify} \triangleq \textsc{if}\ \mathit{waitSet} \neq \{\} \qquad \text{corresponds to method } \mathit{notify}() \text{ in } \mathit{Java} \\
\qquad\qquad\quad \textsc{then}\ \exists\, x \in \mathit{waitSet} : \mathit{waitSet}' = \mathit{waitSet} \setminus \{x\} \\
\qquad\qquad\quad \textsc{else}\ \textsc{unchanged}\ \mathit{waitSet} \\
\\
\mathit{NotifyAll} \triangleq \mathit{waitSet}' = \{\} \qquad \text{corresponds to method } \mathit{notifyAll}() \text{ in } \mathit{Java} \\
\\
\mathit{Wait}(t) \triangleq \mathit{waitSet}' = \mathit{waitSet} \cup \{t\} \qquad \text{corresponds to method } \mathit{wait}() \text{ is } \mathit{Java}
\end{array}
$$

The module then introduces variables for the buffer (as a sequence of data elements) and its wait set (as a set of threads). *RunningThreads* is defined as a set difference between all the threads (*Participants*) and the threads currently in the wait set. Thus, it is the set of threads currently running.

Next, the module defines three “macros” (*operators*, in TLA$^+$) to model the Java methods `wait`, `notify` and `notifyAll`. The definitions may look weird (and somewhat intimidating), but one gets used to it. To understand them, however, we need to dive a little more into TLA$^+$ semantics.

Basically, a TLA$^+$ module defines a *state transition system*. A state (a mapping of variables names to values) *transitions* into a new state through actions that represent the behavior of the system. Actions are defined logically as a relation between values in the state *before* the action and values in the state *after* the action, which are primed. For instance, the Java assignment statement `x = 3*x + 1;` corresponds to the relation `x' = 3*x + 1`. Note that this is a Boolean formula and `=` is good old mathematical equality, *not* an assignment. So, `x' ≥ x` or `(x' - x) % 7 = 1` are possible TLA actions even though they have no equivalent in programming languages like Java. In particular, one can write *nondeterministic* actions (like the last two above for which, given `x`, there are several possible values of `x'`), which are often very useful in modeling concurrent systems: `x' = 0 ∨ x' = x + 1 ∨ x' = x` specifies that `x` could become 0 or increase by 1 or stay unchanged; `x' ∈ {x, x + 1, 0}` is exactly the same thing.

With this in mind, it is now possible to decipher the TLA$^+$ formulations of `wait` (add the thread `t` to the wait set unconditionally), `notifyAll` (remove every thread from the wait set, which becomes empty) and `notify` (if the wait set is not empty, remove one thread, otherwise do nothing). The case of `notify` is the most interesting because it is modeled nondeterministically: *some* thread is removed from the wait set, but we don’t specify which one (and neither does the Java Language Specification).

$$Init \triangleq buffer = \langle\rangle \land waitSet = \{\}$$

$$\begin{aligned} Put(t,\, m) \triangleq\ & \text{IF } Len(buffer) < BufCapacity \\ & \text{THEN } \land buffer' = Append(buffer,\, m) \\ & \qquad\ \land Notify \\ & \text{ELSE } \land Wait(t) \\ & \qquad\ \land \text{UNCHANGED } buffer \end{aligned}$$

$$\begin{aligned} Get(t) \triangleq\ & \text{IF } Len(buffer) > 0 \\ & \text{THEN } \land buffer' = Tail(buffer) \\ & \qquad\ \land Notify \\ & \text{ELSE } \land Wait(t) \\ & \qquad\ \land \text{UNCHANGED } buffer \end{aligned}$$

$$\begin{aligned} Next \triangleq\ & \exists\, t \in RunningThreads : \lor\ t \in Producers \ \land \exists\, m \in Data : Put(t,\, m) \\ & \qquad\qquad\qquad\qquad\qquad\ \lor\ t \in Consumers \land Get(t) \end{aligned}$$

$$Prog \triangleq Init \land \Box[Next]_{\langle buffer,\, waitSet\rangle}$$

Using these Java-like operators, the module continues with a formalization of the `put` and `get` methods. The TLA+ formulation mimics the Java code and is straightforward: `Put(t, m)` : if the buffer is not full, add `m` at the end and notify; otherwise, thread `t` waits and the buffer is left unchanged.

Finally, the module specifies the acceptable initial states (in this example, there is only one possible initial state: all the threads are running and the buffer is empty) and all the possible transitions of the system (`Next`). The definition of `Next` reads as follows: for the system to transition to its next state, *some* thread `t` , currently running, performs an operation; either `t` is a producer and it attempts to put *some* piece of data `m` in the buffer; or `t` is a consumer and it attempts to retrieve some piece of data from the buffer.

Deterministic systems only have one possible behavior. Nondeterministic systems can have many, which is why they are so difficult to test and debug. The deadlock that is being investigated here happens in *some* of these behaviors, but not all (and, as we have seen before, there are *many* behaviors in which it doesn't happen). The TLA+ model has three sources of nondeterminism: which thread wakes up when `notify` is called; which thread acquires the lock on the buffer and can attempt a `put` or `get` operation; and when the thread is a producer, which piece of data is put into the buffer. Note how all this nondeterminism is introduced using logical disjunction or existential quantifiers (which are basically the same thing). Note also that the third source of nondeterminism may not be in the Java program (producers tend to produce specific elements) and is irrelevant anyway. The other two sources of nondeterminism are the ones that complicate the analysis of the system and lead to potential deadlocks. (Actually, further study would show that the deadlock remains possible even if `notify` is made FIFO, so it is really the second source of nondeterminism, the scheduling of threads and the contention on locks, that is the reason for all our trouble here.)

$$NoDeadlock \triangleq \Box(RunningThreads \neq \{\})$$

$$\text{THEOREM } Prog \Rightarrow \Box TypeInv \land NoDeadlock$$

The TLA+ module ends with a formalization of the expected property of the system, namely that it is free of deadlock. The `NoDeadlock` formula says that there is always at least one thread running (the □ makes it apply to *all* the states of the system) and the theorem expresses that the system is type-correct and satisfies the `NoDeadlock` property.

The TLC Model Checker

A TLA+ module can be used for several things. It can be the basis for randomized simulation, similar to what was done with the `AnnotatedBuffer` Java program. More interestingly, one can formally *prove* the correctness of the module, i.e., that the model satisfies the desired properties. TLA+ has proof rules, based on classic concepts like *inductive invariants* and *well founded sets*, which can be used to formally derive the logical implication stated at the end of a module.

As an alternative, *Model checking* (http://cacm.acm.org/magazines/2008/7/5378-qa-talking-model-checking-technology/fulltext) is an approach that has been applied quite successfully in industry. It is much more powerful than randomized simulations and much easier to carry out than formal proofs. Model checking works by basically enumerating all the possible behaviors of a system and checking that they all have the desired properties.

TLC (http://research.microsoft.com/people/lamport/tla/tlc.html) is a *model checker* for TLA$^+$. Although it is very useful, conceptually, TLC is a fairly dumb tool. When confronted with several possibilities (as in the behaviors of nondeterministic systems), it refuses to choose. Unlike Buridan's ass (and more like nondeterministic Turing machines), TLC does not get stuck between possibilities; it explores all of them. More specifically, TLC tries to calculate *all* the reachable states of a system in a breadth-first manner. This is in contrast to the testing our programmer did before, which only tried *some* randomly chosen paths.

From this, the strengths and limitations of a tool like TLC are obvious: On the one hand, it doesn't rely on luck to find bugs; on the other hand, even small nondeterministic systems can have too many states to be model-checked. This problem, sometimes known as *state explosion*, has been the *bête noire* of researchers in the model checking community for years.

Still, model checking is a very useful technique, as can be demonstrated by applying it to our bounded buffer problem. To start TLC, one needs to instantiate a module by assigning values to all its constants. For the buffer example, a TLC configuration file could look something like this:

```
SPECIFICATION Prog
CONSTANTS     Producers = {p1,p2,p3,p4,p5}
              Consumers = {c1,c2,c3,c4,c5}
              BufCapacity = 10
              Data = {m1}
INVARIANT     TypeInv
PROPERTY      NoDeadlock
```

(Since what is put into the buffer is irrelevant, using a very small data type greatly reduces the number of states to explore.)

Given this configuration, TLC produces the following output:

```
Starting... (2011-06-29 15:51:54)
Computing initial states...
Finished computing initial states: 1 distinct state generated.
Model checking completed. <span class="red">No error has been found</span>.
  Estimates of the probability that TLC did not check all reachable states
  because two distinct states had the same fingerprint:
  calculated (optimistic):  val = 3.9E-14
  based on the actual fingerprints:  val = 1.3E-15
3461 states generated, 223 distinct states found, 0 states left on queue.
The depth of the complete state graph search is 20.
Finished. (2011-06-29 15:51:56)
```

In other words, if the buffer has capacity 10 and is shared among 5 producers and 5 consumers, the system is *deadlock free*. This is the configuration our programmer gave up on earlier when his laptop started to smoke, and he was right to give up: This system will never deadlock.

TLC can also be started on a buffer of capacity 2, with 3 producers and 2 consumers. It then produces the following output:

```
Starting... (2011-06-29 16:18:32)
Computing initial states...
Finished computing initial states: 1 distinct state generated.
Error: Invariant NoDeadlock is violated.
```

In itself, this is not very useful. It's very much like the phone call from the customer to complain that everything hangs. The best thing about TLC is what comes next, *an explicit trace of the system that leads to the deadlock state:*

```
Computing initial states...
Finished computing initial states: 1 distinct state generated.
Error: Invariant NoDeadlock is violated.
Error: The behavior up to this point is:
State 1: <Initial predicate>
/\ buffer = <<>>
/\ waitSet = {}

State 2: <Action line 52, col 9 to line 53, col 62 of module Buffer>
/\ buffer = <<m1>>
/\ waitSet = {}
```

It takes 23 transitions to reach the deadlock. Since TLC works in a breadth-first manner, there is no shorter behavior that results in a deadlock. For these 23 actions to actually happen in the right order with the Java program, one needs luck (*good* luck when testing, *bad* luck once deployed). Hence the difficulty of making it happen on purpose.

Several of these transitions are noteworthy, like this one:

```
State 10: <Action line 54, col 9 to line 55, col 62 of module Buffer>
/\ buffer = <<>>
/\ waitSet = {p3, c1, c2}

State 11: <Action line 54, col 9 to line 55, col 62 of module Buffer>
/\ buffer = <<m1>>
/\ waitSet = {c1, c2}
```

Some producer (`p1` or `p2`) successfully puts `m1` into the buffer and calls `notify` . But instead of a consumer, *producer* `p3` is notified and removed from the wait set. As discussed earlier, the problem with this buffer implementation is that producers and consumers wait into the same set. In the same way, the transition from state 19 to state 20 has `p1` call `notify` and `p2` is notified; from state 20 to 21, another `put` operation notifies `p1` .

```
State 21: <Action line 54, col 9 to line 55, col 62 of module Buffer>
/\ buffer = <<m1, m1>>
/\ waitSet = {c1, c2}
```

At this point, the buffer is full and no consumer thread is running. The situation is hopeless and leads to state 24:

```
State 24: <Action line 54, col 9 to line 55, col 62 of module Buffer>
/\ buffer = <<m1, m1>>
/\ waitSet = {p1, p2, p3, c1, c2}
```

All five threads are in the wait set: deadlock!

Limitations of Model-Checking

Of course, if things always worked nicely like this, model checking would be taught in elementary school. As a technique, model checking suffers from a severe liability known as *state explosion*. (Technically, *state space explosion* is a better term. It's not the states that explode. In fact, nothing really *explodes* and model checking is not as dangerous (or exciting) as it may sound.) With all the possible interleavings of atomic actions, the number of states reachable by a system can quickly become huge, or even infinite if state variables are unbounded. Indeed, most of the research related to model checking focuses on the state explosion problem one way or another (symbolic model checking, abstraction, …).

As an illustration, consider the buffer example again with a capacity of 10 but 21 threads (11 producer and 10 consumers). This system can deadlock. The shortest trace is 431 steps and TLC had to generate 2,219,959,047 states (23,011,357 of which are distinct) to find it. (How often is it going to happen to the Java program with random timings?) So when our programmer's customer explains that he uses a buffer of capacity 1024 with "hundreds of threads", clearly TLC won't be the answer.

It turns out that there is an answer (the system is deadlock free exactly when the number of threads is at most twice the buffer capacity), but model checking is not the way to get it. As mentioned earlier, it is possible to carry out mathematical proofs on TLA models, but it is not quite as painless as running a model checker. However, proofs (possibly combined with some model checking) remain the only way to deal with infinite (or large) state spaces.

The lesson here is that a formal model can be used for many purposes and simple model checking can be a way to discover bugs that result from nondeterminism, for which testing can be very tricky (and requires patience, computing resources and, what is much more problematic, luck).

Formal models and model checking may not be suitable for elementary schools but should probably be taught as part of every computer science program. However, CS curriculum designers (ACM, NSF, ABET, …) are rather old-fashioned and only the best schools tend to cover this topic. Did I mention UNH's CS–745/845 (teaching-cs745_845.html)?

Downloads

- *Buffer.java* (Downloads/Buffer.java), the faulty Java implementation
- *AnnotatedBuffer.java* (Downloads/AnnotatedBuffer.java), the annotated version used for debugging
- *buffer.tla* (Downloads/buffer.tla), the TLA$^+$ model
- *buffer.cfg* (Downloads/buffer.cfg), a configuration file to run TLC on *buffer.tla*
- *buffer.pdf* (Downloads/buffer.pdf), the TLA$^+$ module, pretty-printed