A Framework for Model Checking Concurrent Java Components

Brad Long
School of Maths, Physics and Information Technology,
James Cook University (Brisbane Campus), Australia.
Email: brad.long@jcu.edu.au

Abstract— **The Java programming language supports concurrency. Concurrent programs are harder to verify than their sequential counterparts due to their inherent non-determinism and a number of specific concurrency problems, such as interference and deadlock. In this paper we illustrate how to construct a base model of Java concurrency primitives using the Promela language of SPIN. Subsequently, a readers-writers monitor, and eighteen mutants, are used as an example to show the power and simplicity of using SPIN for verifying concurrent Java components. This builds on previous work and contributes in three ways, 1) each Java concurrency primitive is modelled directly and added to a standard modelling library for inclusion into models for a range of concurrent components, 2) we assume a concurrent component may be used in potentially many contexts rather than simply the context or contexts it may have been used or found, 3) by providing a modelling library we illustrate how model checking can be implemented in a simple, powerful, and practical manner.**

Index Terms— **model checking, concurrency, Java, testing and verification**

I. Introduction

A concurrent program specifies two or more processes (or threads) that cooperate in performing a task [1]. Each process is a sequential program that executes a sequence of statements. The processes cooperate by communicating using shared variables or message passing. Programming and testing concurrent programs is difficult due to the inherent non-determinism in these programs. That is, if we run a concurrent program twice with the same input, it is not guaranteed to return the same output both times.

The Java programming language has included concurrency primitives as part of the core language ever since its first public release. Tools and techniques for testing concurrent Java programs are still under active research and include static analysis, dynamic analysis, model checking, and combinations of these techniques.

The SPIN model checker has successfully checked models of software written in various programming languages including Java. This paper illustrates, in detail, the construction of a simple yet powerful Promela library that models each individual Java concurrency language primitive, which is then used to build a model to verify that an example readers-writers monitor is free from interference, deadlock and other liveness errors. This facilitates the construction of models of monitors (and other concurrent components) using the Promela library, rather than reconstructing entirely new models for each monitor under verification. An interesting observation is the suitability of the SPIN language (Promela) functions to model Java components used in a concurrent context.

We review related work in Section II. In Section III, Java concurrency primitives are reviewed and the models of each are constructed and described in detail. In Section IV, the readers-writers example is introduced and a model is constructed using the framework detailed in the previous section. Section V introduces 18 mutant monitors and presents the results of verifying the monitors using SPIN.

II. Related Work

A model is a simplified representation of the real world. It includes only those aspects of the real-world system relevant to the problem at hand. Models of software are often based on finite state machines or call graphs with well-defined mathematical properties [2], [3].

There are many tools and ongoing research in model checking. Of particular note, with respect to the Java language, is Java PathFinder (or JPF). JPF [4], [5] combines model-checking techniques with techniques for dealing with large or infinite spaces. JPF uses state compression to handle complex states, and partial order reduction, abstraction, and runtime analysis techniques to reduce state space. JPF runtime analysis uses the Eraser [6] and GoodLock [7] algorithms as guides to the model checker for detecting potential deadlocks and race conditions. JPF allows predicate abstraction across classes by automatically translating a Java program annotated by user-specified predicates to another Java program that operates on the abstract predicates (using the Stanford Validity Checker [8]). It has been used with Bandera to take advantage of program slicing techniques. An earlier version of JPF converted Java source code to Promela (the language of the model checker SPIN [9], [10]).

Motivation for this paper was driven from previous research on model checking [11], [12] and a strong interest to assess the practicality of applying model checking techniques in industry. The eighteen mutant monitors that are examined in Section V are the same as those used in [13]. However, this paper does not compare model checking with the techniques used in [13] to detect the eighteen mutants. There is recent research [14], [15] on these complementary testing and verification techniques and how to select the best combination for a purpose [16]. It is widely accepted that each have their particular strengths and weaknesses. Whilst much of the work contained in this paper builds on previous research on applying model checking to verify Java programs [11], [12], it contributes in three ways:

1) A Promela procedure is created for each Java concurrency primitive. Hence, models of monitors (and other components) can be constructed using the Promela library, rather than reconstructing entirely new models for each monitor under verification.
2) By focusing on verifying an individual component we do not need to be concerned with the number of threads that are executing in the system as a whole, because we assume the component can be used in potentially many contexts and therefore may be accessed by any number of threads at a time. That is, we verify a component under the assumption of multiple thread access. Hence, a model is created to verify a component (potentially many uses) and not a particular use, which would require modelling the surrounding context (i.e. calling classes) of the component.
3) By implementing a Promela library, we illustrate how simple and practical it is to create SPIN models of Java monitors using the Promela language. The library of Promela procedures corresponding to Java concurrency primitives assists in keeping the model uncluttered and similar in structure to the Java program under verification.

The goal of this paper is to describe a practical Promela library and technique to model check Java programs for verifying concurrent components without the need to install other more complex tools, and show how this technique can be applied in industrial and commercial settings today.

III. Concurrency in Java

Each Java concurrency construct and its model representation is reviewed in this section. Interrupts are not included in the Promela library. Interrupts are discussed in previous work [11] and may be incorporated in the future. Also, it is assumed that the Java memory model has been fixed [17].

A. *Mutual Exclusion and Object Locking*

In the Java programming language [18], [19] mutual exclusion is achieved by a thread locking an object. Two threads cannot lock the same object at the same time, thus providing mutual exclusion. A thread that cannot access a synchronised block because the object is locked by another thread is *blocked*. In Java there are two ways of locking an object.

1) Explicitly call a synchronised block.

```
synchronized (anObject) {
    ...
}
```

The Java code above, locks the object `anObject`. The lock is released when the executing code leaves the synchronised block. If another thread is already executing code within the synchronised block, the requesting thread will be blocked until the thread holding the lock leaves the synchronised block.

2) Synchronize a method.

```
public synchronized void aMethod() {
    ...
}
```

The Java code above, which synchronises a method, is the same as locking the `this` object in a synchronised block. The following code provides identical behaviour:

```
public void aMethod() {
    synchronized (this) {
        ...
    }
}
```

A thread can lock more than one object. For example, the thread executing the following Java code locks the two objects `object1` and `object2`. Both locks are held whilst in the inner-most synchronised block. As each block is exited, the associated lock is released. The Java scheduler is not required to be fair, that is, locks are not necessarily served to blocked threads in a first-in first-out (FIFO) manner.

```
synchronized (object1) {
    ...
    synchronized (object2) {
        ...
    }
}
```

B. *Modelling Mutual Exclusion and Object Locking with Promela*

1) Modelling Lock Acquisition:

```
inline getLock(obj) {
  atomic {
    if
    :: (locked[obj] == _pid)
        -> skip;
    :: (locked[obj] == -1)
        -> locked[obj] = _pid;
    fi;
  }
}
```

The above Promela code models a request for an object lock. This occurs at the beginning of a synchronised method or block and also when re-acquiring the lock after a call to the Java `wait` method. The `atomic` construct is used to group statements together. As expected, statements within the `atomic` block are executed in sequence and are treated as an atomic operation (i.e. no interleaving within statements in the block).

The `getLock` code accepts an object to be locked as its argument. If the current owner of the lock is this thread (identified by `_pid`), then do nothing (`skip`). This models the behaviour of threads requesting locks that they already own. Otherwise, if the object is not locked, then it is locked, and the owner is set to this thread (where this thread has been selected by the JVM to receive the lock). If neither of the guards are satisfied, the model checker will treat this thread as being blocked, awaiting a lock, until one of the guards becomes true. This is exactly what is needed to model a blocking thread and is a powerful feature of the model checker.

The model checker evaluates all possible interleavings of statements, so a blocked thread is checked for progress after every statement. Thread progress is then checked from every point of possible progression. That is, progress is checked immediately after a lock is available and similarly after every statement past that point. Verification traces all paths.

2) Modelling Synchronisation:

```
inline synchronized(obj) {
  getLock(obj);
  lockDepth[obj] = lockDepth[obj] + 1;
}
```

The above code models the **synchronized** keyword in Java. Consideration must be given to re-entrant monitor calls, hence the `lockDepth` variable is incremented. When the synchronised block is exited, the lock will only be released if the code exits the last enclosing synchronised block.

3) Modelling Lock Release:

```
inline releaseLock(obj) {
  // Assertion JM.1
  assert(locked[obj] == _pid);
  locked[obj] = -1;
}
```

The above Promela code models the releasing of a lock. The assertion is not strictly necessary since a lock can only be released by exiting an enclosing synchronised block (i.e. the program structure enforces it). However, the assertion can help during modelling. If a model of a synchronised block has been incorrectly constructed the assertion will fail. The `releaseLock` subroutine erases the owner of the lock. The value `-1` is used to represent an unassigned lock.

4) Modelling Synchronisation Exit:

```
inline exit_synchronized(obj) {
  lockDepth[obj] = lockDepth[obj] - 1;
  if
  :: (lockDepth[obj] == 0)
      -> releaseLock(obj);
  :: else
      -> skip;
  fi;
}
```

The above Promela code models leaving a synchronised block. Note that `lockDepth` is decremented for each call (i.e. each exit of a synchronised block). The lock is only released once the outer-most synchronised block (for a particular object) has been exited.

C. Waiting and Notification

Threads are suspended by calling the Java `wait` method. This causes the lock on the object to be released, allowing other threads to obtain a lock on the object. Suspended threads remain dormant until woken. As an example, a particular implementation of the producer-consumer monitor provides two methods. The `put` method places an item into the buffer and the `get` method (refer below) retrieves an item from the buffer. A thread will be suspended via the `wait` statement if it calls `get` whilst there are no items in the buffer.

```
public synchronized Item get() {
  while (buffer.size() == 0)
      wait();
  ...
}
```

A thread calling `notify` will cause the run-time scheduler, managed by the Java Virtual Machine (JVM), to arbitrarily select a waiting thread to be woken. The selected thread will then attempt to regain the object lock for re-entry to the synchronised block. The `put` call (refer below) places an item into the buffer and then notifies a waiting thread. Only one arbitrarily selected thread is notified. A notified thread attempts to regain the object lock and re-enter the synchronised block at the statement immediately after the call to `wait`. There is also a method `notifyAll` that wakes all waiting threads on the object.

```
public synchronized void put(Item item) {
  ...
  buffer.add(item);
  notify();
}
```

D. Modelling Waiting and Notification with Promela

1) Modelling Waiting Threads:

```
inline wait(obj) {
  // Assertion JM.2
  assert(locked[obj] == _pid);
  atomic {
    releaseLock(obj);
    waiting[obj] = waiting[obj] + 1;
    WAIT[obj]?0;
  };
  getLock(obj);
}
```

The above Promela code for the `wait` operation initially checks the assertion, which ensures that this thread owns the object lock. An assertion failure indicates that the Java code is trying to call process synchronisation primitives without holding the object lock. In practical terms, this would raise an `IllegalMonitorStateException` if the program is executed.

After the assertion check, the object lock is released (via `releaseLock`), the number of threads waiting on the object is incremented (this is used by the `notifyAll` code), and the thread is suspended on the `WAIT` channel. Finally, when the thread is awoken by a call to `notify` or `notifyAll`, the lock is reacquired via the `getLock` subroutine.

2) Modelling Notification:

```
inline notify(obj) {
  // Assertion JM.3
  assert(locked[obj] == _pid);
  if
  :: (waiting[obj] > 0)
      -> WAIT[obj]!0;
         waiting[obj] = waiting[obj] - 1;
  :: else
      -> skip;
  fi;
}
```

The above Promela code for `notify` begins by asserting that the thread owns the object lock. Then, if there are objects waiting, one is non-deterministically selected. In fact, during verification every combination of waiting thread is notified, thus testing every possible execution path.

```
inline notifyAll(obj) {
  // Assertion JM.4
  assert(locked[obj] == _pid);
  do
  :: (waiting[obj] > 0)
      -> WAIT[obj]!0;
         waiting[obj] = waiting[obj] - 1;
  :: else
      -> break;
  od;
}
```

The above Promela code for `notifyAll` is similar to the `notify` implementation. However, in this case, all waiting threads are notified.

E. Putting It All Together

To simplify model creation, each of the Promela subroutines are contained within file called `java.model` (see Appendix A). For convenience, two further files have been created: `single_lock.model` (see Appendix B) and `multi_lock.model` (see Appendix C). Both of these files use the core `java.model` and, in the single lock case, eliminate unnecessary lock handling. As described in the next section, the appropriate file is included in the Promela model of the component under verification.

```
class ReaderWriter {
  private int readers = 0;
  private boolean writing = false;
  private int writersWaiting = 0;

  // Start Read routine
  public synchronized void startRead()
  throws InterruptedException {
    while (writing||(writersWaiting>0)){
      wait();
    }
    ++readers;
  }

  // End Read routine
  public synchronized void endRead() {
    --readers;
    if (readers == 0) {
      notifyAll();
    }
  }

  // Start Write routine
  public synchronized void startWrite()
  throws InterruptedException {
    ++writersWaiting;
    while (writing || (readers != 0)) {
      wait();
    }
    writing = true;
    --writersWaiting;
  }

  // End Write routine
  public synchronized void endWrite() {
    writing = false;
    notifyAll();
  }
}
```

Figure 1. Java code for the readers-writers monitor

IV. READERS-WRITERS EXAMPLE

This section introduces a version of the readers-writers monitor that gives writer threads priority over waiting reader threads. It is described in detail here, and will be used as the example throughout this paper. The readers-writers problem involves a shared resource that is read by reader threads (the *readers*) and written to by writer threads (the *writers*) [20]. To prevent readers and writers interfering with each other, individual writers must be given exclusive access to the resource, locking out other writers and readers. However, as reading does not result in interference, multiple readers may access the resource concurrently. A monitor is used to control this access to the resource. Typical Java code for such a monitor is given in Figure 1.

The monitor state is maintained through three variables: `writing` is a boolean that is true if and only if a writer has access to the resource, the integer `readers` represents the number of readers currently accessing the resource, and the integer `writersWaiting` represents the number of writers waiting for access to the resource.

The `startWrite` method is required to be executed prior to a writer gaining mutually exclusive access (to write) to a shared resource. The `endWrite` method is required to be executed when the writer no longer requires access to the shared resource. Only one writer can have access to the resource at a time; if a second writer seeks access by executing the `startWrite` method it will satisfy the condition of the while-loop (as `writing` will be true) and hence execute a `wait`. Similarly, if a writer seeks access to the resource by executing the `startWrite` method when one or more readers currently have access to the resource, a `wait` will also be executed as `readers` will be non-zero. As a consequence of waiting, `writersWaiting` will become positive.

The `startRead` method is required to be executed prior to a reader gaining access (to read) a shared resource. The `endRead` method is required to be executed when the reader no longer requires access to the shared resource. Many readers can access the shared resource at the same time. If another reader seeks access by executing the `startRead` method, provided there are no writers waiting (i.e. `writersWaiting` is 0) it will fail the condition of the while-loop and hence complete execution of the method and gain access to the resource. However, if a reader executes the `startRead` method when a writer currently has access to the resource (in which case `writing` is true) or when there are writers waiting (in which case `writersWaiting` is positive), the condition of the while-loop will be satisfied and the reader will execute a `wait`. In effect, this implementation of the monitor gives writers priority; if both readers and writers are waiting for access to the resource, preference is given to writers.

A. Modelling Readers-Writers

Constructing a model of the readers-writers monitor is straightforward (see Figure 2). Two processes are defined, a reader and a writer. Since the component being modelled uses only one lock, the file `single_lock.model` is included. The reader process has code for modelling `startRead` and `endRead`. Similarly, the writer process includes code for modelling `startWrite` and `endWrite`. The translation to Promela is simple. Java `while` loops are replaced with Promela equivalents (i.e. `do ... od`). Occurrences of `wait`, `notify`, and `notifyAll` are easily replaced with the subroutines previously detailed in Section III, which are now part of our Promela library of Java primitives. Similarly, synchronised blocks are modelled with `synchronized` and `exit_synchronized`. Little abstraction is required since every variable in the Java monitor is involved in, or affects, one or more concurrent statements. Hence, each instance variable is appropriately modelled. The `init` procedure details the number and type of processes to be verified by the model checker.

Four assertions have been added to the readers-writers model. Violating any of these assertions means that the monitor has not met its behavioural specification. That is, the monitor contains a fault that may lead to a failure. The assertions, `RW.1 - 4`, are trivial to create and are derived from the requirements of the monitor. After a call to `startRead`, `RW.1` asserts that there must be at least one reader in the critical section of the monitor, and `RW.2` asserts that there must be no thread writing. After a call to `startWrite`, `RW.3` asserts that there must be a writing thread, and `RW.4` asserts that there are no readers in the critical section of the monitor.

```
#include "single_lock.model"
bool writing = false;
short readers = 0;
short writersWaiting = 0;

proctype reader() {

  /* startRead */
  synchronized();
  do
  :: (writing || writersWaiting>0) -> wait();
  :: else -> break;
  od;
  readers = readers + 1;
  exit_synchronized();

  assert(readers > 0);  // Assertion RW.1
  assert(!writing);     // Assertion RW.2

  /* endRead */
  synchronized();
  readers = readers - 1;
  if
  :: (readers == 0) -> notifyAll();
  :: else -> skip;
  fi;
  exit_synchronized();
}

proctype writer() {

  /* startWrite */
  synchronized();
  writersWaiting = writersWaiting + 1;
  do
  :: (writing || readers != 0) -> wait();
  :: else -> break;
  od;
  writing = true;
  writersWaiting = writersWaiting - 1;
  exit_synchronized();

  assert(writing);      // Assertion RW.3
  assert(readers == 0); // Assertion RW.4

  /* endWrite */
  synchronized();
  writing = false;
  notifyAll();
  exit_synchronized();
}

// system
init {
  run reader(); run reader();
  run writer(); run writer();
}
```

Figure 2. Readers-writers model

Mutant	Method	Original Code	Mutant Code	Killed By
1	startRead	`while (writing || waitingW>0) wait()`	`while (waitingW>0) wait()`	Assertion (RW.2)
2	endRead	`if (readers==0) notifyAll()`	`notifyAll()`	–
3	startWrite	`while (readers>0 || writing) wait()`	`while (writing) wait()`	Assertion (RW.4)
4	endWrite	`notifyAll()`	`notify()`	Invalid End State
5	startRead	`while (writing || (writersWaiting > 0))`	`if (writing || (writersWaiting > 0))`	Assertion (RW.2)
6	startRead	`while (writing || (writersWaiting > 0))`	`while (writing && (writersWaiting > 0))`	Assertion (RW.2)
7	startWrite	`while (writing ||(readers != 0)`	`if (writing || (readers != 0))`	Assertion (RW.3)
8	startWrite	`while (writing || (readers != 0))`	`while (writing && (readers != 0))`	Assertion (RW.4)
9	startRead	`public synchronized void startRead()`	`public void startRead()`	Assertion (JM.2)
10	startWrite	`public synchronized void startWrite()`	`public void startWrite()`	Assertion (RW.2)
11	startRead	`while (writing || (writersWaiting > 0))`	`while (writing || (writersWaiting > 0) || readers != 0)`	–
12	endRead	`--readers` is synchronised	`--readers` is not synchronised	Assertion (JM.4)
13	startWrite	`++writersWaiting` is synchronised	`++writersWaiting` is not synchronised	Assertion (RW.2)
14	endWrite	`writing = false` is synchronised	`writing = false` is not synchronised	–
15	endRead	`public synchronized void endRead()`	`public void endRead()`	Assertion (JM.4)
16	endWrite	`public synchronized void endWrite()`	`public void endWrite()`	Assertion (JM.4)
17	startRead	`++readers`	`while (true) ++readers`	Search Error
18	endRead	`notifyAll`	`notify`	Invalid End State

TABLE I.
READERS-WRITERS MUTANTS

V. DETECTING FAILURES IN MUTANT MONITORS

This section describes the results of using the Promela models to kill a number of mutant implementations of the readers-writers monitor. This demonstrates the effectiveness and applicability of SPIN, however, it is recognised that this is only a start to more work on applying and evaluating SPIN on a range of components including more realistic industrial components. The mutants were created by modifying the correct implementation with programming faults. They originated from a number of different sources: some were created to model typical source-code faults that programmers make, some were created to exercise specific concurrency runtime failures, and others were obtained from exam questions from a course on concurrency.

Model checking using SPIN was applied to each mutant implementation in turn. Table I presents the mutant components verified by SPIN. The components are mutations of the readers-writers monitor detailed in Section IV. The table provides enough information to be able to construct each mutant from the modified code detailed in the table and the original example. Table I lists the following information:

- Mutant: an identifier for the mutant.
- Method: the name of the component method that was changed.
- Original Code: the original component code.
- Mutant Code: the modified code.
- Killed By: how the mutant was detected.

Table I details the results of verifying the Promela models of the mutant monitors with SPIN. The assertions that detected each mutant are referenced in the 'Killed By' column.

Mutant 2 has been changed, but it is not in error. It is a valid implementation of the monitor. Verifying mutant 4 results in an 'Invalid End State'. This means execution of the component may result in permanently suspended (or dormant) threads. In this case, the SPIN verification trace shows that a certain execution sequence may result in two readers and one writer suspended in the wait set.

No problem is detected with mutant 11 although it is an erroneous implementation. Mutant 11 only allows one reader to enter the critical section at a time. The original requirement of the readers-writers monitor is to allow many readers to access the critical section at the same time.

Mutant 14 is not detected. On close inspection it is noted that it will not produce a failure, hence, this is a correct, albeit confusing, implementation of the readers-writers monitor.

Verifying mutant 17 causes the model checker to run out of space with a ‘max search depth too small’ error. Gradually increasing the search depth space does not eliminate the error. On inspection it is obvious that the endless loop in the monitor is causing this problem.

Like mutant 4, the verification of mutant 18 results in an ‘Invalid End State’. The SPIN verification trace shows that some execution sequence could result in the permanent suspension of one reader and two writer processes. Details of assertion failures of all other components are detailed in Table I.

VI. Conclusion

Model checking has been used in research and industry to successfully verify Java programs including concurrent programs. In particular, SPIN has a good track record for detecting concurrent programming bugs, and has been incorporated into sophisticated automated verification tools, such as Bandera and early versions of JPF. Clearly, this paper does not extend such tools. This paper illustrates the simplicity of building a simple, yet powerful framework for model checking concurrent Java components that almost any programmer could implement with little training.

Initially, the framework or base model of some Java concurrency primitives is created. The framework is used when building the model of a component under verification. SPIN then verifies the model for absence of interference, deadlock and other liveness failures. For simplicity, two frameworks were created: 1) for single lock programs, and 2) for programs that use multiple locks. The framework was used to construct a model of the readers-writers monitor and 18 mutants. Verification of the models detected 15 failures. On closer inspection it was discovered that there were in fact two benign mutants, meaning that all but one failure was detected.

It is hoped that this paper goes some way to encouraging further research in the area, and in particular the use of model checking techniques on commercial applications that use Java concurrency. The technique scales well since individual components are tested, not an entire system. For ease of demonstration this paper has focused on a simple Java monitor, but there is no reason to be so restrictive, since our model handles multiple locks it can be applied to a variety of concurrent Java components.

Future work consists of: 1) identifying additional correctness properties including the use of invariants and temporal assertions, 2) using the technique on a range of concurrent Java components, 3) building base models for additional Java primitives and concurrency packages (such as `java.util.concurrent`), and 4) conducting case studies on verifying a range of components from commercial applications.

Appendix A. Promela Model of Java Concurrency Primitives

```
short locked[NUM_LOCKS] = -1;
chan WAIT[NUM_LOCKS] = [0] of {bit};
short waiting[NUM_LOCKS] = 0;
short lockDepth[NUM_LOCKS] = 0;


inline _synchronized(obj) {
    _getLock(obj);
    lockDepth[obj] = lockDepth[obj] + 1;
}


inline _exit_synchronized(obj) {
    lockDepth[obj] = lockDepth[obj] - 1;
    if
    :: (lockDepth[obj] == 0)
        -> _releaseLock(obj);
    :: else
        -> skip;
    fi;
}


inline _getLock(obj) {
    atomic {
        if
        :: (locked[obj] == _pid)
            -> skip;
        :: (locked[obj] == -1)
            -> locked[obj] = _pid;
        fi;
    }
}


inline _releaseLock(obj) {
    assert(locked[obj] == _pid);
    locked[obj] = -1;
}


inline _wait(obj) {
    assert(locked[obj] == _pid);
    atomic {
        _releaseLock(obj);
        waiting[obj] = waiting[obj] + 1;
        WAIT[obj]?0;
    };
    _getLock(obj);
}


inline _notify(obj) {
    assert(locked[obj] == _pid);
    if
    :: (waiting[obj] > 0)
      -> WAIT[obj]!0;
         waiting[obj] = waiting[obj] - 1;
    :: else -> skip;
    fi;
}


inline _notifyAll(obj) {
    assert(locked[obj] == _pid);
    do
    :: (waiting[obj] > 0)
        -> WAIT[obj]!0;
           waiting[obj] = waiting[obj] - 1;
    :: else -> break;
    od;
}
```

APPENDIX B. SINGLE LOCK MODEL

```
#define NUM_LOCKS 1

#include "java.model"

inline synchronized() {
    _synchronized(0);
}

inline exit_synchronized() {
    _exit_synchronized(0);
}

inline wait() {
    _wait(0);
}

inline notify() {
    _notify(0);
}

inline notifyAll() {
    _notifyAll(0);
}
```

APPENDIX C. MULTI-LOCK MODEL

```
/**********************************
 * define NUM_LOCKS in             *
 * component-under-test model file */

#include "java.model"

inline synchronized(obj) {
    _synchronized(obj);
}

inline exit_synchronized(obj) {
    _exit_synchronized(obj);
}

inline wait(obj) {
    _wait(obj);
}

inline notify(obj) {
    _notify(obj);
}

inline notifyAll(obj) {
    _notifyAll(obj);
}
```

REFERENCES

[1] G. Andrews, *Concurrent Programming: Principles and Practice*. Addison Wesley, 1991.

[2] E. Clarke, E. Emerson, and A. Sistla, “Automatic verification of finite-state concurrent systems using temporal logic specifications,” *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[3] M. McMillan, “Symbolic model checking,” Ph.D. dissertation, Carnegie Mellon University, 1992.

[4] K. Havelund, “Java PathFinder, a translator from Java to Promela,” in *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. Springer-Verlag, 1999.

[5] W. Visser, K. Havelund, G. Brat, and S. Park, “Model checking programs,” in *Proceedings of the 15th International Conference on Automated Software Engineering*. IEEE Computer Society, 2000, pp. 3–12.

[6] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, “Eraser: A dynamic data race detector for multithreaded programs,” *ACM Transactions on Computer Systems*, vol. 15, no. 4, pp. 391–411, Nov. 1997.

[7] K. Havelund, “Using runtime analysis to guide model checking of Java programs,” in *Proceedings of the 7th SPIN Workshop*. Springer-Verlag, 2000, pp. 245–264.

[8] Stanford University, “Stanford validity checker,” 2004, http://verify.stanford.edu/SVC/.

[9] G. Holzmann, “The model checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[10] ——, *The SPIN Model Checker*. Addison Wesley, 2004.

[11] K. Havelund and T. Pressburger, “Model checking Java programs using Java PathFinder,” *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[12] H. K. and J. Skakkebæk, “Applying model checking in Java verification,” in *Proceedings of the 7th SPIN Workshop*. Springer-Verlag, 1999, pp. 216–231.

[13] B. Long, R. Duke, D. Goldson, P. Strooper, and L. Wildman, “Mutation-based exploration of a method for verifying concurrent Java components,” in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004) – 2nd International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*. IEEE Computer Society, 2004.

[14] B. Long, “Testing concurrent Java components,” Ph.D. dissertation, The University of Queensland, 2005.

[15] B. Long, P. Strooper, and L. Wildman, “A method for verifying concurrent Java components based on an analysis of concurrency failures,” *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 281–294, 2007.

[16] P. Strooper and M. Wojcicki, “Selecting V&V technology combinations: How to pick a winner?” in *Proceedings of the 12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007)*. IEEE Computer Society, 2007, pp. 87–96.

[17] W. Pugh, “The Java memory model is fatally flawed,” *Concurrency: Practice and Experience*, vol. 12, no. 6, pp. 445–455, 2000.

[18] J. Gosling and K. Arnold, *The Java Programming Language*, 2nd ed. Addison Wesley, 1998.

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 2nd ed. Addison Wesley, 2000, http://java.sun.com/docs/books/jls/index.html.

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

Brad Long received the B.Sc. and Ph.D. degrees in computer science from the University of Queensland in 1988 and 2005 respectively. In 2000, he received the M.B.A. degree from the University of Southern Queensland.

From 1988, he has worked as a software engineer and manager for a number of national and international firms including Mincom Pty Ltd, Fujitsu Ltd, and Oracle Corporation.

He is currently a lecturer at James Cook University (Brisbane Campus) and is a director of Touchstone Consulting Pty Ltd. His research interests include software engineering, especially software verification and testing, and concurrent and distributed systems. He is a member of the IEEE and the IEEE Computer Society.