Modelling Java Concurrency with Object-Z

Roger Duke    Luke Wildman    Brad Long
School of Information Technology and Electrical Engineering,
The University of Queensland,
Brisbane, Qld 4072, Australia.
email: {rduke, luke, brad}@itee.uq.edu.au

Abstract

In this paper we present a formal model of Java concurrency using the Object-Z specification language. This model captures the Java thread synchronisation concepts of locking, blocking, waiting and notification. In the model we take a viewpoints approach, first capturing the role of the objects and threads, and then taking a system view where we capture the way the objects and threads cooperate and communicate. As a case study, we then use Object-Z inheritance to integrate the model with the classical producer-consumer system to create a specification directly incorporating the Java concurrency constructs.

1. Introduction

If we are to reason about concurrent systems, and ultimately formally verify, or even adequately test, the correctness of concurrent software, we need to have a precise specification of the model of concurrency adopted by the underlying programming language. Furthermore, we need to be able to integrate this model into the specification of our concurrent systems. With these goals in mind, in this paper we present a formal model of Java concurrency [5] using the Object-Z specification language [3]. Our model captures the core Java thread synchronisation concepts of threads locking objects, of objects blocking threads, of threads waiting on objects, and of objects notifying the waiting threads. There are two main advantages of using Object-Z. First, it enables us to take a viewpoints approach, where we can begin by capturing separately the role of the objects and threads, expressing these roles from their own point of view, and then taking a system view where we capture the way the objects and threads cooperate and communicate. Second, we can make direct use of Object-Z inheritance to integrate our concurrency model with any software system specification and hence extend the system specification to include the Java concurrency concepts. The specification that results creates a system view that can form the basis for the creation of test suites and formal concurrent system analysis. As an illustrative case study we use Object-Z inheritance to integrate the Java concurrency model with a specification of the classical producer-consumer system [1].

1.1. Related work

Several authors have written formal specifications of the Java thread synchronisation model. Long and Long [9] present a model of Java concurrency using Standard Z [6] which accomplishes similar goals to ours. We extend their work by showing how Object-Z [3] can be used to model suspension, and how application threads and objects may be specified by inheritance of our abstract Object and Thread classes.

For the purpose of justifying test-coverage it is important to understand the classes of failure that may be exhibited by a system and the ability of certain test-sequences to detect such failures. For the purpose of classifying failures in concurrent Java components, Long and Strooper [10] present a model of Java concurrency specified using Petri-nets [4]. By considering the failure of transitions in the model, they are able to systematically consider all possible failures of a Java monitor and provide explanations of how they may occur. Unfortunately the failure explanations are not based on the model but on their considerable experience with such Java programs. This is partly because the Petri-net model does not contain enough detail about the mechanics of each transition to explain how a transition may fail. This detail is present in our Object-Z model and in future work we will explore a more formal treatment of the possible failures.

Formal specifications may be used to generate test-cases for their corresponding implementations. However, few specifications contain the detail required to model thread suspension. An Do [2] extends Object-Z with special notation for preconditions for the suspension and notification of threads of multiple types. The state of the model includes the number of threads of each type and each operation involves
the conditions for suspension, the condition for successful entry to the critical section and the effect, and the condition for notification and its subsequent effect. The extended state is then used as a basis for test-sequences for input to the Java concurrency testing tool Conan [8] which uses a timing analysis [1] to test the thread suspension behavior of a monitor. Our model of thread behavior is specified in Object-Z without any extension and is a richer, more accurate model of Java thread behavior. However, our model contains more than is required for generating test-sequences for Conan. This is because the test-method being used detects thread suspension through delayed termination, and does not detect the intermediate blocking, lock request, lock acquisition and lock release events observable in our model.

Welch and Martin [13] model Java multi-threading in CSP [11] with the aim of demonstrating the correctness of their implementation of Occam-like channels for Java. Because they use model checking to verify the absence of deadlock and race-conditions, their model is deliberately restricted to reduce state explosion; they consider a limited number of objects and threads. Our aim is to use formal verification for the above proofs, so our model need not be restricted and our model allows any number of objects and threads.

2. An Object-Z model of Java concurrency

The Java thread synchronisation model we wish to specify is based on synchronized methods and blocks of code, together with the Java methods wait, notify and notifyAll inherited from the Java Object superclass. We shall assume basic understanding of the Java synchronisation model, although in fact the issues will be briefly discussed here as part of the informal description of the Object-Z specification. A basic understanding of Object-Z is also assumed (see [3] for Object-Z background).

The Object-Z specification presented in this section constructs a model of Java concurrency consistent with that described within the Java Language Specification [5] and the Java Virtual Machine Specification [7]. Instances (i.e. objects) of the class Object denote the objects in a Java system; this class captures the concurrency of the system from the point of view of these objects. In a similar way, instances of the class Thread denote the program threads in a Java system; this class captures the concurrency of the system from the point of view of these threads. The class System captures the overall view of the concurrent system in terms of the way the objects and threads in a Java system interact.

As is standard in Object-Z, when an identifier denoting a class is used within a type expression it denotes the set of instances (objects) of that class [3]. Hence the sets OneObject and OneThread defined by

OneObject == \{so : \text{Object} \mid \#so \leq 1\}
OneThread == \{st : \text{Thread} \mid \#st \leq 1\}

denote subsets containing at most one element of instances of the classes Object and Thread respectively.

2.1. The class Object

Consider now the class Object in detail.

```
Object

isLockedBy : OneThread
isBlocking : \text{P} Thread
hasAskedToWait : \text{P} Thread

disjoint\{isLockedBy, isBlocking, hasAskedToWait\}

Init

isLockedBy = \emptyset
isBlocking = \emptyset
hasAskedToWait = \emptyset

_lockRequestedByThread __
\Delta(isBlocking)

\If t : Thread
\Then \text{isLockedBy} = \emptyset
\text{isBlocking} = \emptyset
\text{hasAskedToWait} = \emptyset
\text{isBlocking} = \emptyset \cup \{t\}

_giveLockToThread __
\Delta(isLockedBy, isBlocking)

\If t : Thread
\Then \text{isLockedBy} = \emptyset
\text{isBlocking} = \emptyset \cup \{t\}
\text{isLockedBy} = \emptyset
\text{isBlocking} = \emptyset \cup \{t\}

_lockReleasedByThread __
\Delta(isLockedBy)

\If t : Thread
\Then \text{isLockedBy} = \emptyset
\text{isLockedBy} = \emptyset
\text{isLockedBy} = \emptyset

_askThreadToWait __
\Delta(isLockedBy, hasAskedToWait)

\If t : Thread
\Then \text{isLockedBy} = \emptyset
\text{isLockedBy} = \emptyset
\text{isLockedBy} = \emptyset
```

2
The three state attributes each denote subsets of Thread instances. The attribute isLockedBy denotes the thread (if any) that holds the lock on the object, isBlocking denotes the set of threads blocked on the object, and hasAskedToWait denotes the set of threads that are waiting to be notified by the object. These three subsets are mutually disjoint, capturing the requirement that any thread can play at most one of these roles for any given object. Initially all three subsets are empty.

The operation lockRequestedByThread specifies what happens when a thread requests access to the object. This captures the object’s view of the situation when a Java thread seeks to enter a synchronized block of the object. The thread in question cannot already be locking the object, cannot be blocked on the object, and cannot be waiting to be notified by the object. The outcome of the operation is that the thread joins the set of threads blocked by the object.

The operation giveLockToThread specifies what happens when the object selects a thread from among those currently waiting on the object and allows that thread to lock it. This captures the object’s view of the situation when a Java thread is given access to a synchronized block of the object. The object cannot already be locked, and the outcome of the operation is that the thread is given the lock on the object and is removed from the set of blocked threads.

The operation lockReleasedByThread specifies what happens when the thread that currently holds the lock on the object releases that lock. This captures the thread’s view of the situation when a Java thread completes the execution of a synchronized block of the object. The outcome is that the object is no longer locked.

The operation askThreadToWait specifies what happens when the object requests that the thread currently holding the lock on the object wait for notification from the object. This captures the object’s view of the situation when a Java thread executes a ‘wait’ command while accessing a synchronized block of the object. The outcome is that the thread is added to the set of threads waiting to be notified by the object, and the object is no longer locked.

The operation notifyThread specifies what happens when the locked object selects a thread from among those currently waiting on the object and notifies it. This captures the object’s view of the situation when a ‘notify’ command is executed by some other thread currently accessing a synchronized block of the object. The outcome is that the selected thread is removed from the set of waiting threads and added to the set of threads blocked on the object.

The operation notifyAllThreads is like notifyThread except that all threads waiting on the object are notified. This captures the object’s view of the situation when a ‘notifyAll’ command is executed by some other thread currently accessing a synchronized block of the object. The outcome is that all the threads currently waiting on the object are added to the set of threads blocked on the object, while the set of threads waiting on the object becomes empty.

2.2. The class Thread

Consider now the class Thread in detail. The three state attributes each denote subsets of Object instances. The attribute isLocking denotes the set of objects that are locked by the thread, isBlockedOn denotes the object (if any) blocking the thread, and isWaitingOn denotes the object (if any) on which the thread is waiting for notification. If the thread is locking an object, the thread cannot be blocked or waiting on that object. Furthermore, if a thread is blocked on an object it cannot be waiting on any object, and likewise if it is waiting on an object it cannot be blocked on any object. Initially all three subsets of objects described by the attributes are empty.

\[
\begin{array}{ll}
\text{Thread} & \\
\text{isLocking} : \mathbb{P} \text{Object} \\
\text{isBlockedOn} : \text{OneObject} \\
\text{isWaitingOn} : \text{OneObject} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{isLocking} \cap \text{isBlockedOn} = \emptyset \\
\text{isLocking} \cap \text{isWaitingOn} = \emptyset \\
\#(\text{isBlockedOn} \cup \text{isWaitingOn}) \leq 1 \\
\end{array}
\]

\[
\begin{array}{ll}
\text{INIT} & \\
\text{isLocking} = \emptyset \\
\text{isBlockedOn} = \emptyset \\
\text{isWaitingOn} = \emptyset \\
\end{array}
\]
The outcome of the operation is that the object in question
is blocked by the thread. This captures the Java thread’s view of
the situation when the thread is given a lock on the object
currently blocked by the thread.

The operation getLockOnObject specifies what happens
when the thread, currently not blocked or waiting on any
object, is asked to obtain a lock on an object. This captures
the Java thread’s view of the situation when a ‘wait’ command
is executed by some other thread currently accessing a
synchronized block of the object. The outcome is that the
object is removed from the set of objects locked by the thread.

The operation releaseLockOnObject specifies what hap-
pens when the thread, currently not blocked or waiting on
any object, is asked to release a lock on an object. This captures
the Java thread’s view of the situation when it completes the
execution of a synchronized block of the object. The outcome is that
the object is removed from the set of objects locked by the thread.

The operation notifiedByObject specifies what happens
when a thread, currently not blocked or waiting on an
object, is notified by a ‘notify’ or ‘notifyAll’ command. This captures
the Java thread’s view of the situation when it executes a
‘synchronized’ block of the object. The outcome is that the
object is removed from the set of objects locked by the thread.

Finally, consider the class System in detail.

```
System

objects : P Object
threads : P Thread

∀ o : objects • (o.isLockedBy ∪ o.isBlocked) ⊆ threads
∀ t : threads • (t.isLocking ∪ t.isBlocked ∪ t.isWaitingOn) ⊆ objects
∀ o : objects; t : threads •
  o ∈ t.isLockedBy ⇔ t ∈ o.isLockedBy
  o ∈ t.isBlocked ◁ t ∈ o.isBlocked
  o ∈ t.isWaitingOn ⇐ t ∈ o.hasAskedToWait

INIT
∀ o.objects • o.INIT
∀ t.threads • t.INIT

requestLock ≡
[ t! : threads ] • t!.requestLockOnObject
|| [ o? : objects ] • o?.lockRequestedByThread
```

The operation requestLockOnObject specifies what happens
when the thread, currently not blocked or waiting on any
object, requests access to an object. This captures
the Java thread’s view of the situation when it seeks to execute
a synchronized block of the object. The object in question
cannot already be locked by the thread, and the outcome of
the operation is that the object blocks the thread.

The operation getLockOnObject specifies what happens
when the thread is given a lock on the object currently blocking
it. This captures the Java thread’s view of the situation when
it is given access to a synchronized block of the object.

The outcome of the operation is that the object in question
is added to the set of objects locked by the thread, and the
thread is no longer blocked by any object.

The operation releaseLockOnObject specifies what happens
when the thread, currently not blocked or waiting on any
object, selects one of the objects that it is currently locking
and releases that lock. This captures the Java thread’s view of
the situation when it completes the execution of a
synchronized block of the object. The outcome is that the
object is removed from the set of objects locked by the thread.

The operation askedToWaitByObject specifies what happens
when the thread, currently not blocked or waiting on any
object, is asked to wait for notification by one of the
objects in the set of objects locked by the thread. This captures
the Java thread’s view of the situation when it executes a
‘wait’ command while accessing a synchronized block of
the object. The outcome is that the object is removed from
the set of objects locked by the thread and becomes the object
on which the thread is waiting for notification.

The operation notifiedByObject specifies what happens
when a thread, currently not blocked or waiting on an
object, is notified by a ‘notify’ or ‘notifyAll’ command. This captures
the Java thread’s view of the situation when a ‘notify’ or ‘notifyAll’ command is executed
by some other thread currently accessing a synchronized
block of the object. The outcome is that the selected thread is
no longer waiting on the object but instead becomes blocked
on the object.
3. Deriving properties from the specification

By analysing the Object-Z specification we can derive various properties of the Java model of concurrency. For example, we can derive the unique blocking and waiting property: a thread can be blocked on, or waiting on, at most one object, i.e.

\[ \forall o_1, o_2 : \text{objects} \implies o_1 \neq o_2 \implies (o_1.\text{isBlocking} \cap o_2.\text{isBlocking}) = \emptyset \]

\[ (o_1.\text{hasAskedToWait} \cap o_2.\text{hasAskedToWait}) = \emptyset \]

Proof:
Suppose \( t \in (o_1.\text{isBlocking} \cap o_2.\text{isBlocking}) \). Then by the state invariant of the System class, \( o_1 \in t.\text{isBlockedOn} \) and \( o_2 \in t.\text{isBlockedOn} \). But as \( o_1 \neq o_2 \) this is a contradiction of the Thread class invariant that isBlockedOn is a set of objects of size at most 1. The proof of the unique waiting property is similar.

As another example, we can derive the unique locking property: an object can be locked by at most one thread, i.e.

\[ \forall t_1, t_2 : \text{threads} \implies t_1 \neq t_2 \implies (t_1.\text{isLocking} \cap t_2.\text{isLocking}) = \emptyset \]

Proof:
Suppose \( o \in (t_1.\text{isLocking} \cap t_2.\text{isLocking}) \). Then by the state invariant of the System class, \( t_1 \in o.\text{isLockedBy} \) and \( t_2 \in o.\text{isLockedBy} \). But as \( t_1 \neq t_2 \) this is a contradiction of the Object class invariant that isLockedBy is a set of threads of size at most 1.

4. Applying the model to Producer-Consumer

Now that we have a formal specification of Java concurrency we can apply this model to software systems such as the classical producer-consumer. It is straightforward to specify this problem in Object-Z in a way that ignores concurrency issues. We begin by specifying the classes Producer, Consumer and Buffer:

```
[Item]
```

```
Producer

put

[Item] : Item
```

```
Consumer

get

[Item] : Item
```

```
Buffer

[size : \mathbb{N}]
```

```
buffer : \text{seq Item}
```

```
Init

buffer = \emptyset
```

```
#buffer \leq size
```
In the ProducerConsumer system class we allow an indeterminate number of producers, consumers and buffers:

ProducerConsumer

<table>
<thead>
<tr>
<th>put</th>
<th>get</th>
</tr>
</thead>
<tbody>
<tr>
<td>∆(buffer)</td>
<td>∆(buffer)</td>
</tr>
<tr>
<td>item? : Item</td>
<td>item! : Item</td>
</tr>
<tr>
<td>#buffer &lt; size</td>
<td>#buffer &gt; 0</td>
</tr>
<tr>
<td>buffer' = buffer ∩ (item?)</td>
<td>buffer = ⟨item!⟩ ∪ buffer'</td>
</tr>
</tbody>
</table>

This specification is essentially sequential. Under the blocking semantic model of Object-Z [3, 12], at each stage any enabled operation, and only an enabled operation, can occur. Consequently the operations put and get will be interleaved sequentially and the issue of concurrent access does not arise. To introduce concurrency into the producer-consumer problem, we can merge this specification with the earlier specification of Java concurrency by making use of Object-Z inheritance.

Consider first the class ProducerThread that inherits both of the classes Producer and Thread.

ProducerThread

\[\text{put} \equiv p : \text{producers} \cdot p.put \parallel b : \text{buffers} \cdot b.put\]
\[\text{get} \equiv c : \text{consumers} \cdot c.get \parallel b : \text{buffers} \cdot b.get\]

The operation getLockAndPut captures the situation, from the viewpoint of the producer, where the producer gets the lock on an object (in fact, a buffer object), puts an item into the buffer and then releases the lock. (In this paper, matching inputs and outputs (e.g. o? and ol) of the sequential composition operator "\(\circ\)" are identified but not hidden.)

The operation getLockAndWait captures the situation, again from the viewpoint of the producer, where the producer gets the lock (on a buffer) but is asked to wait (by the buffer). As a side-effect of the askedToWaitByObject operation inherited from the Thread class, the lock (on the buffer) is released.

The class ConsumerThread is similar in structure to the ProducerThread class just specified; in this case the classes Consumer and Thread are inherited.

ConsumerThread

\[\text{askThreadToWait} \equiv \text{askedToWaitByObject}\]

The operation getLockAndGet captures the situation, from the viewpoint of the consumer, where the consumer gets the lock on a buffer, gets an item from the buffer and releases the lock.

Consider now the class BufferObject that inherits the classes Buffer and Object.

BufferObject

\[\text{askThreadToWait} \equiv \text{askThreadToWaitByObject}\]

The schema askThreadToWait in this class is conjoined with the operation of the same name inherited from the class Object.
The operation \( \textit{giveLockAndPut} \) captures the situation, from the viewpoint of the buffer, where the buffer gives its lock to a thread (in fact, a producer thread), accepts an item (from the producer) and notifies all threads waiting on the buffer. The lock on the buffer is then released (by the producer). The operation \( \textit{giveLockAndGet} \) is defined similarly, except it involves a consumer rather than a producer thread.

The operation \( \textit{giveLockAndWait} \) captures the situation, from the viewpoint of the buffer, where the buffer gives its lock to a thread (in fact, a producer thread), accepts an item from the producer (consumer) to wait. In either case the lock on the buffer is released.

5. Detecting deadlock

We can apply the Object-Z model of Java concurrency to define deadlock. A thread \( t_1 \) \textit{directly depends upon} a thread \( t_2 \) if \( t_1 \) is blocked or waiting on an object locked by \( t_2 \). We can make this precise by defining the partial function \( \textit{directlyDependsUpon} \):

\[
\textit{directlyDependsUpon} : \text{Thread} \rightarrow \text{Thread}
\]

\[
\forall t_1, t_2 : \text{Thread} \quad t_1 \textit{directlyDependsUpon} t_2 \iff \\
\exists o : \text{Object} \quad o \in (t_1, \text{isBlockedOn} \cup t_1, \text{isWaitingOn}) \quad t_2 \in o, \text{isLockedBy}
\]

The relationship \( \textit{directlyDependsUpon} \) is functional because any thread is blocked or waiting on at most one object, and any object is locked by at most one thread. If \( t_1 \) \textit{directlyDependsUpon} \( t_2 \) then \( t_1 \) can do nothing until \( t_2 \) releases the lock on its object.

We can extend this notion to chains of dependencies. A thread \( t_A \) \textit{depends upon} a thread \( t_B \) if there is a sequence \( (t_1, t_2, \ldots, t_n) \) of threads such that

\[
t_A \textit{directlyDependsUpon} t_1 \quad t_1 \textit{directlyDependsUpon} t_2 \quad \ldots \quad t_n \textit{directlyDependsUpon} t_B
\]

This can be made precise by using transitive closure to define the relation \( \textit{dependsUpon} \):

\[
\textit{dependsUpon} \equiv \textit{directlyDependsUpon}^+
\]

A thread \( t \) is \textit{self dead-locked} if it depends upon itself, i.e.

\[
\textit{selfDeadlocked} : \text{Thread}
\]

\[
\forall t : \text{Thread} \quad t \in \textit{selfDeadlocked} \iff t \textit{dependsUpon} t
\]

If a thread is self dead-locked then it will never be able to proceed; it is effectively permanently stuck! However, it is not just the self dead-locked threads that are stuck in this way. A thread \( t \) is \textit{dead-locked} if it depends upon a thread that is self dead-locked. To be precise,

\[
\textit{deadlocked} : \text{Thread}
\]

\[
\forall t : \text{Thread} \quad t \in \textit{deadlocked} \iff \exists t_1 : \text{Thread} \quad t \textit{dependsUpon} t_1 \quad t_1 \in \textit{selfDeadlocked}
\]

In essence, the deadlock specified here arises because in general it is possible for a thread that is locking one object to
become blocked or waiting on some other object. Such deadlock, however, is not possible for the concurrent producer-consumer system specified earlier because any producer or consumer that is blocked or waiting on a buffer cannot at the same time hold the lock on any other buffer, i.e.

\[
\forall t: (\text{producers} \cup \text{consumers}) ; \quad b_1, b_2 : \text{buffers} \bullet \\
\quad b_1 \neq b_2 \land b_1 \in \{ t.\text{isBlockedOn} \cup t.\text{isWaitingOn} \} \\
\Rightarrow b_2 \notin t.\text{isLocking}
\]

**Proof:**
The proof is by structural induction. Suppose \( t \in \text{producers} \) (the case for \( t \in \text{consumers} \) is essentially identical). Initially, the three sets \( t.\text{isBlockedOn}, t.\text{isWaitingOn} \) and \( t.\text{isLocking} \) are each empty, so the condition holds. In this state only the operation \( \text{requestLock} \) is enabled with respect to \( t \). If it occurs, \( t.\text{isBlockedOn} \) becomes \( \{ b \} \) for some buffer \( b \) but the other two sets remain unchanged and the condition remains true.

If \( t.\text{isBlockedOn} \) is \( \{ b \} \) and the other two sets are empty, operations \( \text{lockAndPut} \) and \( \text{lockAndWait} \) are enabled with respect to \( t \). Suppose \( \text{lockAndPut} \) occurs. Then \( t \) undergoes the thread operations \( \text{getLockOnObject} \), \( \text{put} \) and \( \text{releaseLockOnObject} \); no other operation involving \( t \) is permitted. Hence \( t.\text{isBlockedOn} \) becomes empty and \( t.\text{isWaitingOn} \) remains unchanged. During the execution of the operation, \( t.\text{isLocking} \) becomes \( \{ b \} \) but afterwards becomes empty and we revert to the initial state. Hence the condition remains true both during and after the operation.

If \( \text{lockAndWait} \) occurs, \( t \) undergoes the thread operations \( \text{getLockOnObject} \) and \( \text{askedToWaitByObject} \); no other operation involving \( t \) is permitted. Hence \( t.\text{isBlockedOn} \) becomes empty and \( t.\text{isWaitingOn} \) becomes \( \{ b \} \). During the execution of the operation \( t.\text{isLocking} \) also becomes \( \{ b \} \), but afterwards becomes empty. Hence the condition remains true both during and after the operation.

If \( t.\text{isWaitingOn} \) is \( \{ b \} \) and the other two sets are empty, the only way that \( t \) can be affected is if \( \text{lockAndPut} \) or \( \text{lockAndGet} \) are performed on \( b \) by some other thread. In this case \( t \) undergoes the thread operation \( \text{notifiedByObject} \) and \( t.\text{isBlockedOn} \) becomes \( \{ b \} \), while the other two sets are empty. Hence we revert to a state already dealt with above, and the condition remains true.

### 6 Conclusions

In this paper we have constructed an Object-Z specification of Java’s thread synchronisation model which takes a viewpoint approach by capturing separately the role of the objects and the threads in the system before combining these views into a system view that captures the way objects and threads cooperate and communicate. This model enables us to use Object-Z inheritance to integrate Java concurrency within an existing system specification in order to get a fully concurrent specification of the system. We illustrated this approach by exploring the classical producer-consumer system. We can also use our model to reason about system properties and explore issues such as deadlock detection. In future work we plan to use the model to explore the generation of suites for testing concurrent systems implemented in Java.

### References


