Extending IMP to Support Threads with Race Detection by Model Checking

Kristal Pollack
kristal@cs.ucsc.edu

CMPS203 Final Project

Abstract

This project explored non-determinism introduced by threads in the context of an extremely simple language called IMP. IMP was extended to support multi-threaded execution, as well as basic locking with mutexes. A model checker was implemented to explore all possible execution states of a program written in this extended IMP language. While the model checker executes a number of checks are run with it to test for race conditions that may exist for both variables and locks. The model checker writes all the possible state transitions, as well as annotations for race conditions between states, to a file in the input language for the graphing utility dot. This file can then be given as input to dot to generate a state diagram.
Extending IMP to Support Threads with Race Detection by Model Checking

Kristal Pollack

CMPS203 Final Project

1 Introduction

The motivation for this project was to add non-determinism to IMP \cite{3} by extending the language to support multi-threaded execution. Threads share resources, so when more than one thread uses a resource the order of access to this resource is non-deterministic. This is particularly problematic when the shared resource is modified during execution, because the value of the resource now depends on the non-deterministic order of execution. In order to protect operations where determinism is important, locks were implemented for IMP. Only one thread can hold a specific lock at any time, so a lock can be used to restrict access to a shared resource for an operation that requires its value to be deterministic. It is often difficult to tell whether a multi-threaded program is deterministic or not, so a model checker was implemented to explore all the possible paths of execution through a multi-threaded IMP program. The model checker was extended to detect possible causes of non-determinism and highlight where they occur. Its output can be visualized as a state diagram using the graphing utility dot \cite{2}.

2 Abstract Syntax

In order to extend IMP to support threads and locks, first the abstract syntax had to be extended by adding the syntatic entities $program$ and $mutex$:

$$\text{program} ::= [C_1...C_n]$$

$$\text{mutex} ::= \text{locations}$$

$$\text{Lock } m \text{ for } m \in \text{mutex add to Comm}$$

$$\text{Unlock } m \text{ for } m \in \text{mutex add to Comm}$$

In order to support threads the type $program$ was added. This defines a program as a list of commands $C_1$ through $C_n$ that will each be executed as a separate thread. A $mutex$ type was also added, to support mutual exclusion among threads by naming a location to store the mutex state in. The expressions Lock and Unlock were added to the existing $Comm$ type to perform operations on a mutex.
3 Operational Semantics

The operational semantics for IMP remain the same with the addition of the semantics for handling Lock and Unlock for mutexes and executing a threaded program, as shown below.

\[
\begin{align*}
tid, m \ni \sigma, H[\text{Lock } m] &\longrightarrow \sigma', H[\text{Skip}] \quad \text{where } \sigma' = [\sigma, m = [\text{tid}]] \\
tid, m = [] \in \sigma, H[\text{Lock } m] &\longrightarrow \sigma', H[\text{Skip}] \quad \text{where } \sigma' = [\sigma, m = [\text{tid}]] \\
tid, m = [\text{tid}, t_1...t_n] \in \sigma, H[\text{Lock } m] &\longrightarrow \sigma, H[\text{Skip}] \\
tid, m = [t_1...t_n] \in \sigma, H[\text{Lock } m] &\longrightarrow \sigma', H[\text{Lock } m] \quad \text{where } \sigma' = [\sigma, m = [t_1...t_n, \text{tid}]] \\
tid, m \ni \sigma, H[\text{Unlock } m] &\longrightarrow \sigma, H[\text{Skip}] \\
tid, m = [] \in \sigma, H[\text{Unlock } m] &\longrightarrow \sigma, H[\text{Skip}] \\
tid, m = [\text{tid}, t_1...t_n] \in \sigma, H[\text{Unlock } m] &\longrightarrow \sigma', H[\text{Skip}] \quad \text{where } \sigma' = [\sigma, m = [t_1...t_n]] \\
k = \text{rand 1 to } n, \sigma, H[S_1 \text{c}_k S_n] &\longrightarrow \sigma', H[S_1 \text{c}'_k S_n] \\
\text{where } S_i \text{ list of commands } 1 \text{ to } k-1 \text{ and } S_n \text{ list of commands } k+1 \text{ to } n
\end{align*}
\]

As one would expect, all the threads share a single store that maps variable names and mutex names to their current values. Notice that the values stored for mutexes are lists of thread identifiers. This list is the queue of threads waiting for the lock, and at the head of the list is the current lock owner. When a Lock is attempted for a mutex the thread ID of the current thread is checked against the head of the waiting queue. If it is the head, or the queue is empty, the thread is the owner of the mutex and the Lock is granted. If another thread is the owner, the current thread ID is queued for the mutex and the Lock does not succeed on this step. The thread must then wait until it is the owner of the mutex before it can proceed past the current Lock command. The Unlock command for a mutex will only succeed if the thread attempting the Unlock is the current owner of the mutex. A successful Unlock removes the head of the waiting queue and the next thread ID in the queue becomes the head/owner if the queue is not empty.

It is interesting to note that there are simpler semantics for mutual exclusion, however these have a trade-off in fairness. The values stored for mutexes could simply be a 0 or a 1 for unlocked and locked, but there is no ownership associated with the mutex so a thread can unlock a locked mutex even if it wasnt the thread that locked it. If a program is written carefully this isnt a problem, but it could cause trouble if used incorrectly. The value stored for a mutex could be the thread ID that locked it or 0 (assuming thread IDs are greater than 0). This gives ownership to mutexes, however, when running a program a thread could potentially be waiting for a lock for perhaps an unfair amount of time.

4 Model Checking

In order to explore the non-determinism introduced with the addition of threads to IMP a model checker was implemented to check all the possible states of an IMP program. The model checker starts from the initial state the program would begin to run in, and then explores every possible combination of thread ordering during execution using breadth first traversal.

Figure 1 shows a simple example of the output available from the model checker executing the simple program.
The items displayed in each state from left to right are: the program counter $p$, the thread path (order of thread operations) $t$, the variables and their values, the mutexes and their values. If there are no variables and/or mutexes “Empty” is displayed instead. By displaying the thread path it is easy to see the tree of possible states that are formed by a multi-threaded program. Figures 2 and 3 show other options for viewing the same data as shown in Figure 1. Figure 2 leaves out the thread path, so it is easier to see where states converge at the same point in a program. Figure 3 leaves out both the thread path and the program counter so just the values of the shared resources and their transitions can be examined.

This is a trivial example because the threads weren't accessing the same resources, therefore there were no race conditions. Model checking becomes interesting when threads access and modify shared resources.
because race conditions become possible. Since a model checker explores every possible state of a multi-threaded program it provides the opportunity for detecting if and where race conditions exist.

4.1 Race Detection

A race condition means that accesses to a shared resource are non-deterministic. A race condition can be generally described as a situation where one thread modifies a shared resource, which another thread is attempting to read or modify. This non-deterministic ordering of events may cause unexpected results. These race conditions may lead to incorrect behavior for a program, so it is useful to check for them. Unfortunately this is not a simple problem as static race detection has been proven to be an NP hard problem [4].

Race detection can easily be done while model checking since all the possible states are explored. Of course this is not an efficient way to detect race conditions, but it is the most accurate way since it is an NP hard problem. Detecting with a model checker is fairly straightforward when using a breadth first exploration of the possible states. From some state, all the commands that could possibly be executed in order to get to the next state can be compared to see if any of these commands access the same shared resource. If at least one of the accesses modifies the shared resource then there is a race condition from the current state. So in the tree of all possible states, at each node all children can be compared to determine if there is a race to get to the next state.

Here is an example program that has race conditions:

Initial state: x = 10
Figure 4:

[Assn ("x", Sum (Var "x", Int 10)); Assn ("x", Sum (Var "x", Int 15))]

Figure 4 shows the tree for the above program. The output here uses the same options as Figure 2, but now there are some additional bubbles between states for which a race condition occurs in the transition. A race condition bubble contains the same information as the state bubbles, but instead of the value of the shared resources it gives the resource that is being raced for and the thread IDs of the threads that are racing for it at this point. As you can see, there is a race for the variable "x and the result of the race is that there are three different values possible for x at the end of the program. If these were banking transactions to deposit money into an account it would be possible to lose 5 or 10 dollars in the transfer due to race conditions in the software.

The race conditions for x can be eliminated by requiring that a lock be held by a thread before it can access and/or modify x. Here is the same program implemented with a mutex m:
Initial state: \( x = 10 \)

\[
\begin{align*}
&\text{Seq (Lock (Mutex “c”), Seq (Assn (“x”, Sum (Var “x”, Int 10)), Unlock (Mutex “c”)))}; \\
&\text{Seq (Lock (Mutex “c”), Seq (Assn (“x”, Sum (Var “x”, Int 15)), Unlock (Mutex “c”)))}
\end{align*}
\]

Figure 5 shows the tree for this program. Notice that there is only one final value for \( x \). The race for \( x \) is gone, however there is now a race for the mutex \( m \). Race conditions exist when locking a mutex, however these cases are not usually considered in race detection. This is because mutexes are used to prevent race conditions on shared resources by incurring the non-determinism at the time of locking, so that what follows can be deterministic (accessing shared resources). Therefore the order in which a lock is acquired between the threads is generally accepted as being non-deterministic. However, there is a case where races for locks become interesting, which is discussed in the next section.

4.2 Deadlock Detection

Deadlock for threads occur when threads cannot make progress because they are waiting on locks they will never be able to acquire. This is caused by a special case of race conditions for acquiring locks. If locks are acquired in an order such that the threads are all waiting for a lock that another waiting thread already has then deadlock is possible. The simplest case is where 2 threads both try to acquire two locks in the opposite order. If each thread acquires a lock and then waits for the other to be released, a deadlock is reached. For more complex cases deadlock can be detected by searching for a cycle in a dependency graph of waiting threads and the lock owners. To build a dependency graph for deadlock detection each thread becomes a node and edges are added from a waiting thread to the thread that is the owner of the lock it is waiting for. Such a graph construction is shown in Figure 6. Deadlock detection is implemented in the model checker by testing each Lock operation requested to see if it will cause the requesting thread to wait, and if so, it traverses the dependency graph of locks starting with the edge that would be created by this thread waiting. If along the traversal path the thread we started from is revisited then we have a cycle and we know deadlock has been reached. When deadlock is found the state of it is recorded and progress further down that path is discontinued. Figure 7 shows the large state diagram (simplified) for the following program involving only 3 locks.

\[
\begin{align*}
&\text{Seq (Lock (Mutex “c”), Seq (Lock (Mutex “d”), Seq (Unlock (Mutex “c”), Unlock (Mutex “d”)))}; \\
&\text{Seq (Lock (Mutex “d”), Seq (Lock (Mutex “e”), Seq (Unlock (Mutex “d”), Unlock (Mutex “e”)))}; \\
&\text{Seq (Lock (Mutex “e”), Seq (Lock (Mutex “c”), Seq (Unlock (Mutex “e”), Unlock (Mutex “c”)))}}
\end{align*}
\]

Figure 8 shows an enlarged portion of Figure 7 showing that the finished state is either reached due to deadlock or the successful Unlock of all the locks. Deadlock can be avoided by imposing a partial order on the locks such that in order to acquire a lock of a higher order the thread must own all the locks lower than it [1].

5 Conclusion

Threads and locking done using strict methodologies for avoiding race conditions and deadlocks can be beneficial for many applications, however it is clear there are many opportunities for unintended errors. Examining even simple multi-threaded programs with a model checker makes it clear that when using threads the number of possible states grows enormously with the complexity of the program, leaving room for
errors that could take many runs to surface. It is clear that model checking is not an efficient methodology for detecting race conditions for multi-threaded programs of realistic complexity, so the need for good approximate static checkers is obvious.

References

Figure 5:
<table>
<thead>
<tr>
<th>Locks</th>
<th>Owner</th>
<th>Waiting</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>t1</td>
<td>t3</td>
</tr>
<tr>
<td>B</td>
<td>t2</td>
<td>t1</td>
</tr>
<tr>
<td>C</td>
<td>t3</td>
<td>t2</td>
</tr>
</tbody>
</table>

Figure 6:

Figure 7:
Figure 8: