Solving the Apprentice Challenge with 3VMC

November 28, 2002

Problem Statement

The apprentice challenge was presented by Moore [1] as a challenge in verification of Java programs. The challenge is to show that the value of the `counter` variable of the `Container` class in Figure 1 increases monotonically (under all possible schedules).

Modelling in 3VMC

Our solution of the apprentice challenge is based on [2] and does not assume any *a priori* bound on the number of `Job` threads or on the value of the `counter` field. This should be contrasted with previous attempts to solve the apprentice challenge using model-checking (i.e., the "finite Apprentice").

In our solution, we use the predicates of Table 1. The set $Fields$ contains all fields used in the program. The set $Labels$ contains all program labels. The predicates $zero(v)$ and $successor(v_1, v_2)$ are used to represent integer values as a sequence starting with a node v_0 representing the value 0, for which $zero(v_0)$ is true, and proceeding with successive nodes related by the $successor(v_1, v_2)$ predicate.

The model used here could be easily extended to handle the overflow of integer variables (by introducing a special terminating node in the representation of the integers). For simplicity, we do not introduce such overflow and assume that execution does not go beyond some (unspecified) maximal integer.

Predicates	**Intended Meaning**
$isthread(t)$	t is a thread
$\{at[lab](t) : lab \in Labels\}$	thread t is at label lab
$\{rv[fld](o_1, o_2) : fld \in Fields\}$	field fld of the object o_1 points to the object o_2
$heldby(l, t)$	the lock l is held by the thread t
$blocked(t, l)$	the thread t is blocked on the lock l
$zero(v)$	v represents the integer 0
$successor(v_1, v_2)$	v_2 is the successor of v_1

Table 1: Core predicates used for verification of the Apprentice Challenge.

```
class Container {
   public int counter;
}

class Job extends Thread {
   Container objref;
   public Job incr () {
      synchronized(objref) {
         objref.counter = objref.counter + 1;
      }
      return this;
   }
   public void setref(Container o) {
      objref = o;
   }
   public void run() {
      for (;;) {
         incr();
      }
   }
}

class Apprentice {
   public static void main(String[] args) {
      Container container = new Container();
      for (;;) {
         Job job = new Job();
         job.setref(container);
         job.start();
      }
   }
}
```

Figure 1: Source of Apprentice Example

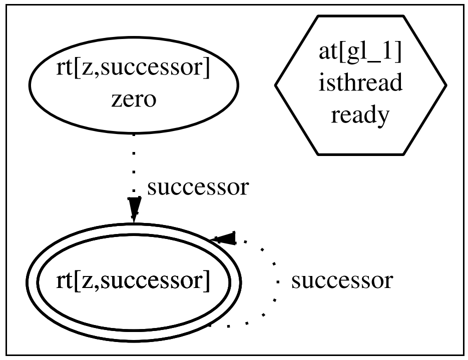

Figure 2: Initial configuration for the apprentice challenge.

```
public Job incr () {
   synchronized(objref) {
      //objref.prevcounter = objref.counter;
      temp = objref.counter + 1;
      objref.counter = temp;
   }
   return this;
}
```

Figure 3: Conceptual rewrite of `incr()` method.

The initial configuration for the 3VMC model is shown in Figure 2. In this configuration there is a single thread node, corresponding to the main program thread. This thread resides at the initial label $\texttt{gl}_1$, and is ready to be scheduled. The other nodes in this configuration represent integer values: one node represents the value zero, and the summary node summarizes the rest of the integer values.

Our system requires two technical modification of the `incr()` method (shown in Figure 3): (i) splitting the increment statement into two assignments, one assigning `counter + 1` to a temporary variable, and another copying the value of the temporary variable into `counter` (In principle, this could be performed by a trivial front-end); (ii) instrument the method to record the previous value of the counter, this again is a technical issue that could be avoided in principle. A conceptual view of the instrumented method is shown in Figure 3. It is important to note that `prevcounter` is introduced as an additional predicate in the model and not as an additional program variable, i.e., it cannot be modified by the program.

Figure 4 and Figure 5 show two intermediate configurations occurring in the analysis of the Apprentice program. The abstract configuration in Figure 4 represents concrete configurations in which an arbitrary number of `Job` threads reside at their initial label ($\texttt{jl}_1$), the main thread is at label $\texttt{gl}_2$ after starting the `Job` thread referenced by

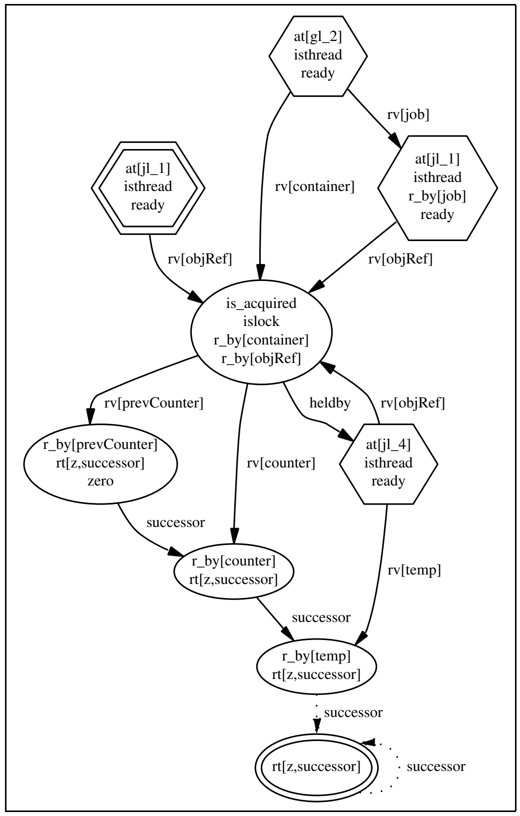

Figure 4: Intermediate configuration of the apprentice challenge.

its `job` field, and another `Job` thread is at label $\mathtt{jl}_4$ after setting the value for `temp` and before setting the new value to `counter`. Note that in this configuration the value of the `prevcounter` is zero.

In the abstract configuration of Figure 5 the value of `counter` is an arbitrary integer, and the value of `prevcounter` is its immediate predecessor.

We use the instrumentation predicates of Table 2 to refine the abstraction.

Results

We applied 3VMC to verify that the original Apprentice program satisfies the goal property. Verification produced 1757 configurations and took approximately 120 seconds and 2.46 MB of memory.

We have also applied 3VMC to find errors in an erroneous version of the Apprentice program in which no synchronization was used by `Job` threads while performing the `incr()` operation. In this analysis, an error was detected after approximately 720 seconds, processing 6066 configurations taking 13.8 MB of memory. Note that since no

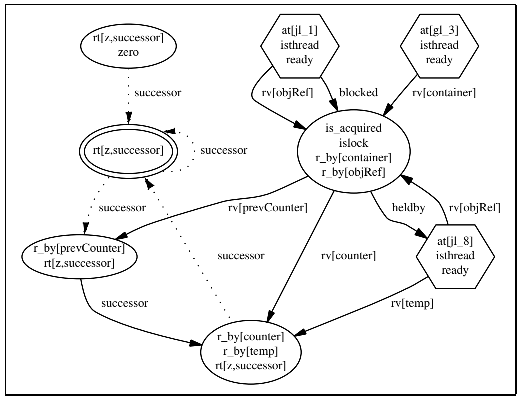

Figure 5: Another intermediate configuration of the apprentice challenge.

Predicates	**Intended Meaning**
$isacquired(l)$	l is acquired
$rt[z, successor](v)$	v is reachable from $zero$ via $successor$ edges
$\{r\_by[fld](v) : fld \in Fields\}$	v is referenced by a field fld

Table 2: Instrumentation predicates used in verification of the Apprentice Challenge.

synchronization was applied between `Job` threads, the number of possible interleaving considered in this exploration is huge.

Unlike the ACL2 solution for the apprentice challenge, our approach is based on a conservative abstraction of the concrete Java semantics. Generally, this means that we might produce false-alarms even when a property does hold for the verified program. However, for the Apprentice challenge, we are able to verify the goal property with no false alarms.

References

[1] J. S. Moore and G. Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(3):193–216, 2002.

[2] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. In *Proc. of 27th POPL*, pages 27–40, Mar. 2001.