Static Deadlock Detection in SHIM with an Automata Type Checking System

Dave Aaron Smith
das2169@columbia.edu

Nalini Vasudevan
naliniv@cs.columbia.edu

Stephen Edwards
sedwards@cs.columbia.edu

Abstract

With the advent of multicores, concurrent programming languages are become more prevelant. Data Races and Deadlocks are two major problems with concurrent programs. SHIM is a concurrent programming language that guarantees absence of data races through its semantics. However, a program written in SHIM can deadlock if not carefully written.

In this paper, we present a divide-and-merge technique to statically detect deadlocks in SHIM. SHIM is asynchronous, but we can greatly reduce its state space without loosing precision because of its semantics.

Keywords: SHIM, Concurrency, Deadlock, Static Analysis, Automata

1 Introduction

SHIM is a deterministic concurrent language. SHIM is a combination of Kahn's network and Hoare's CSP rendezvous style of communication. It is guaranteed to be deterministic: a program written in SHIM has the same input/output behavior regardless of nondeterministic scheduling choices of the run-time environment.

Although, SHIM is asynchronous, the semantics simplifies the verfication process. SHIM does not require a powerful model checker like SPIN. SPIN considers all interleavings in a concurrent model. We can use synchronous abstractions for SHIM because of SHIM's scheduling independence. If a program deadlocks with one particular schedule of the run-time environment, then it also deadlocks with any other schedule. This property greatly reduces the state space in our verification model.

In this paper, we propose a new technique that builds the automaton of every task in the program. We merge automata using SHIM's semantic rules. By merging, we build the combined behavior of two or more concurrent tasks. When there is a conflict while merging, we report a deadlock. We abstract the data values in the program. If our verifier reports the absence of a deadlock, then the program is guaranteed to be deadlock free, however the converse is not true because of data abstraction.

We describe the SHIM language and show how to build the abstracted automaton for individual tasks in the program. Then we describe our merge algorithm and prove it is correct. Finally, we conclude by saying that a design in the language can greatly simplify concurrent programming verification challenges.

2 Related Work

Many static techniques have been proposed to find deadlocks and dataraces in concurrent program. SPIN is one tool that can be used to model check interleaved concurrency models. It checks for all possible interleavings in a program. In SHIM, this is not required therefore making the state space of the problem small and hence verification easy.

3 The SHIM Programming Language

SHIM is an imperative language. It is C-like with additional constructs for concurrency. *p par q* runs statements *p* and *q* in parallel, and waits for both *p* and *q* to terminate before proceeding. *next c* is a blocking communication operator that sends or receives data overa communication channel depending on the context in which it occurs.

Tasks in SHIM run asynchronously, however they synchronize whenever data has to be shared and this happens explicitly through communication. The communication is through multiway rendezvous: there can be multiple receivers at a time.

In Figure 1, *a* and *b* are two channel shared by task 1 and task 2. Both the tasks run concurrently because of the *par* statement. The *next a* in task 1 is on the left hand

```
void main()
{
 chan int a, b;
 {
   // Task 1
   next a = 5; // Send 5 on a (wait for task 2)
   // a = 5 here
   next b; // Receive b (wait for task 2)
   // b = 10 here
 } par {
   // Task 2
   next a; // Receive a (wait for task 1)
   // a = 5 here
   next b = 10;// Send 10 on b (wait for task 1)
   // b = 10 here
  }
}
```

Figure 1. A SHIM program in which two tasks communicate on channels a and b

```
void main() {
  chan int a, b;
  {
    // Task 1
    next a = 5; // Deadlocks here
    next b = 10;

  } par {

    // Task 2
    next b; // Deadlocks here
    next a;
  }
}
```

Figure 2. A SHIM program that deadlocks

side of the assignment and therefore the SHIM compiler interprets it as send. The *next a* in task 2 is a receive. The two tasks rendezvous at the *next a* statements, exchange data and proceed to rendezvous at *next b*. On channel b, task 2 is the sender and task 1 is the receiver.

In Figure 2, the two tasks attempt to communicate again on channels a and b. However, task 1 attempts to synchornize on a expecting task 2 to also synchronize on a, while task 2 tries to synchronize on b causing a deadlock.

```
void main() {
  int i;
  chan int a, b;
  {
    for (i = 0 ; i < 100 ; i++) {
      if (i % 10)
        next a = 1;
      else
        next a = 0;
      next b = 10;
    }
  } par {
    next a;
    next b;
  }
}
```

Figure 3. A deadlock-free SHIM program with a loop, conditionals, and a task that terminates

4 Abstracting SHIM programs

We follow the same abstraction technique used by Vasudevan and Edwards. Since tasks synchronize only when they communicate, we can abstract any computation in a task. We also assume that both branches of a conditional statement can be taken with equal probability. This gives us the flexibility to abstract away data values and hence reduce the state space. This abstraction may however lead to false positives that we discuss later.

5 The Algorithm

First we will construct automata for individual tasks in a SHIM program. Then we will select parallel tasks, and merge them into unified sub-automata that describes the combined behaviour of both automata. We will work our way up from leaf tasks, with the goal of unifying the entire program automaton. If we can accomplish such a task, we can conclude that our program is guaranteed to be deadlock free.

We run our algorithm through Figure 3. The example in Figure 3 starts two tasks that communicate through channels a and b. The first task communicates on channels a and b 100 times. The second task communicates on a and b once. Once task 2 terminates, task 1 does not wait for task 2 to terminate. During the first cycle of communication, task 1's communication pattern meshes with task 2. Therefore, the program does not deadlock.

5.1 Automata Construction

Our algorithm creates an automaton that models the control and communication behavior for every task in SHIM. We have shown the automata for Figure 3 in Figure 4. The first value, *Si* in each state is a unique label. The main program goes to the *par* state from its *start* state and then exits. Task 1 can take two transitions upon start. It can either take the *if* branch or the *else* branch and wait on *a*. Finally, it waits on *b*. We have represented wait on *a* as two distinct nodes in the graphs, because they occur at different places in the control flow of the program. There is a loop back from *b* to both the *a*'s representing the *for* loop in Figure 3.

5.2 Representation

Our automata have several different kinds of nodes, so our algorithm will describe how to combine every possible type of pairs of nodes. First, we will define every type of node.

For convinience, we use a triangle to represent a subgraph in an automata. For eg., the subgraph starting from *state 10* in Figure 4 is shortly represented as a triangle as shown in Figure 5. We label triangles with the unique label of the top node.

Rectangles represent machine decisions. Therefore, outgoing edges on rectangles represent branches in code. We take a pessimistic approach to machine decisions, and insist that all possible machine decisions remain valid. We assume that all outgoing edges occur with equal probability. A rectangle may contain a channel, representing communication along that channel. A rectangle may represent *start* state or *exit* (*null*) state.

In the context of two tasks, a rectangle that indicates a blocking action on a channel *c* is an *internal node* of one task if *c* does not belong to the other task; otherwise we call it a *external node*. Internal and external nodes have meaning with respect to two tasks while merging. The representations of these nodes are shown in Figure 6. In Figure 3, both *a* and *b* are external nodes with respect to tasks 1 and 2.

We also introduce a new environment node as ellipse while merging that we represent as shown in Figure 7. SHIM is a partial ordered system. The ordering between two nodes of two different task may be unclear when we try to merge them. In such cases, we create a new environment node with outgoing transitions as possible orderings. Ellispes appear when we merge automata. We take an optimistic approach to environment decisions, and assume

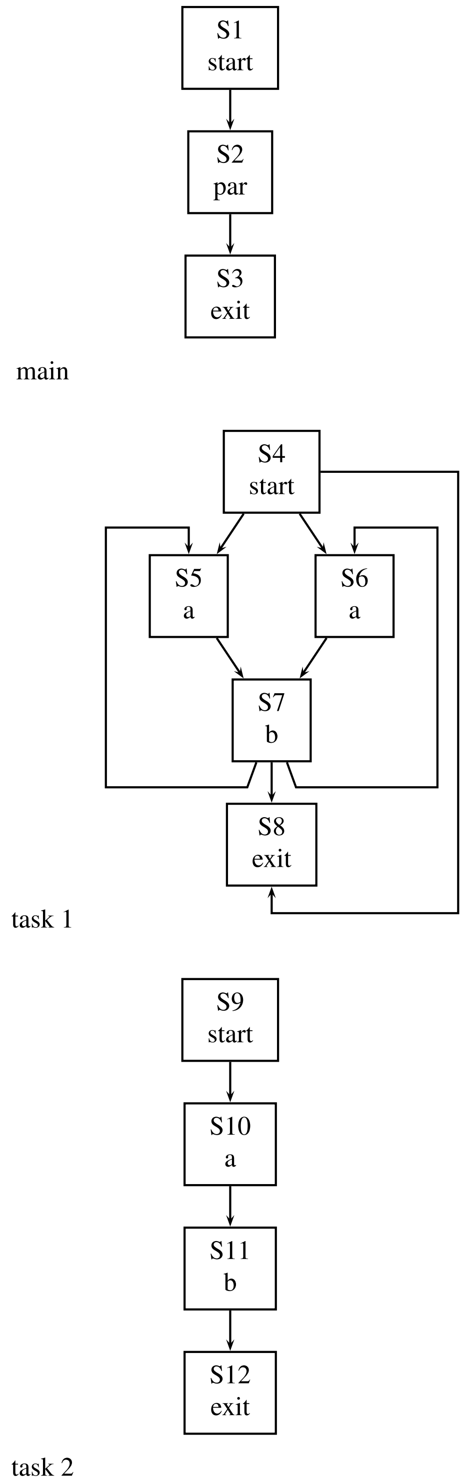

Figure 4. The automata for the example in Figure 3. The compiler broke the *main* function into three tasks.

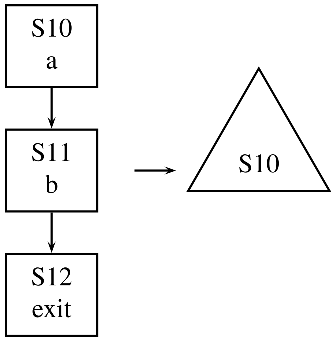

Figure 5. Representing a sub-automaton

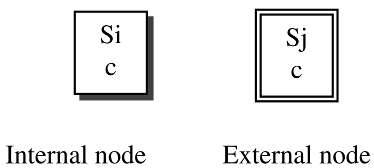

Figure 6. Internal and external nodes

that the environment will make a valid decision if one exists. Therefore, as it becomes necessary, we can remove up to all but one outgoing edges from circles.

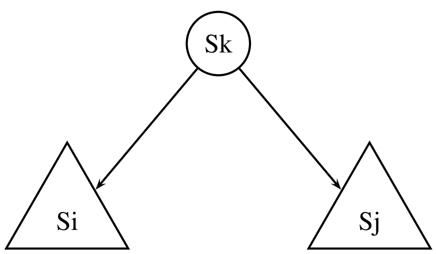

Figure 7. Environment node

5.3 Selecting tasks to merge

First we choose which automata to merge. We start by considering parallel tasks that are siblings. We merge children tasks before merging parent tasks. Now consider two automata whose intersection of channels is small compared to the number of channels. The product of such a pair would tend to expand according to the product of the number of channels. Accordingly, we should pick pairs of automata with a large overlap. The heuristic we use is as follows: choose the pair of automata that maximize the number of overlapping channels minus the smaller number of non-overlapping channels found in one of the automata. For eg., suppose we have two tasks, each containing 10 and 15 channels respectively, out of which 5 is common. Then our heuristic function returns $5-10=-5$. We calculate this heuristic function for all possible pairs of siblings and choose the pair that returns the highest value. The highest value it can take is 0 when the channel list of one task is a subset of the other. Once we form the product of two automata, we must recompute the heuristic.

5.4 The Product Machine

Consider the two subautomata you wish to productize. Each should begin at a blank start node and end with a node marked as the null node. We start at the top blank nodes and combine them according to the rules that follow. This will cause you to recurse through the automata to create a new automaton. If we are unable to produce a product automaton, then you have encountered a potential deadlock. If we can form the product, then the two original tasks represented by the original automata are guaranteed to be deadlock free. We can now represent them with our single product automaton.

While merging nodes, we maintain a hash table and add a pointer to each of the newly merged nodes into the hash table. While merging, if we encounter a node that is already in the hash table, we do not repeat the merge operation again. Therefore if task 1 has M nodes and task 2 has N nodes, the product automata will have atmost O(M*N) nodes.

In the following sections, we discuss the merge techniques based on the type of nodes merged.

5.4.1 Merging two external nodes

Consider Figure 8. If $c_1 != c_2$, then we have a deadlock, so we label the node which represents the product of S_i and S_j as X, a dead node. If $c_1 = c_2$, then we label the product node as $S_1 \cdot S_2$. We create new child nodes for every pair of subtrees under S_i and S_j.

5.4.2 Merging two internal nodes

Consider figure 2. c_1 is present in the first task but absent in the other. c_2 is in the second task but absent in first. After combining them, either c_1 can come first or c_2.

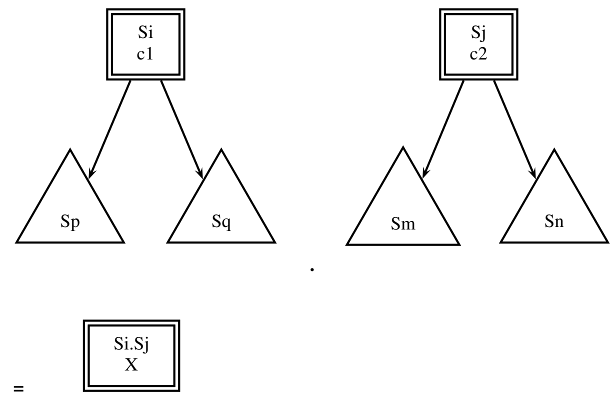

Figure 8. Merging two external nodes, $c1! = c2$

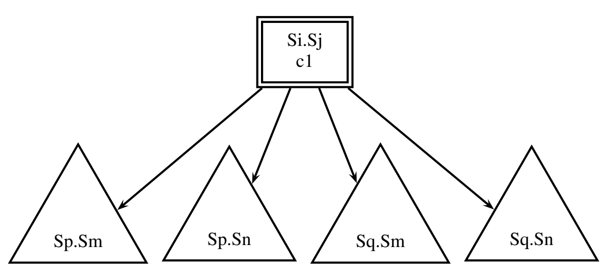

Figure 9. Merging two external nodes, $c1 = c2$

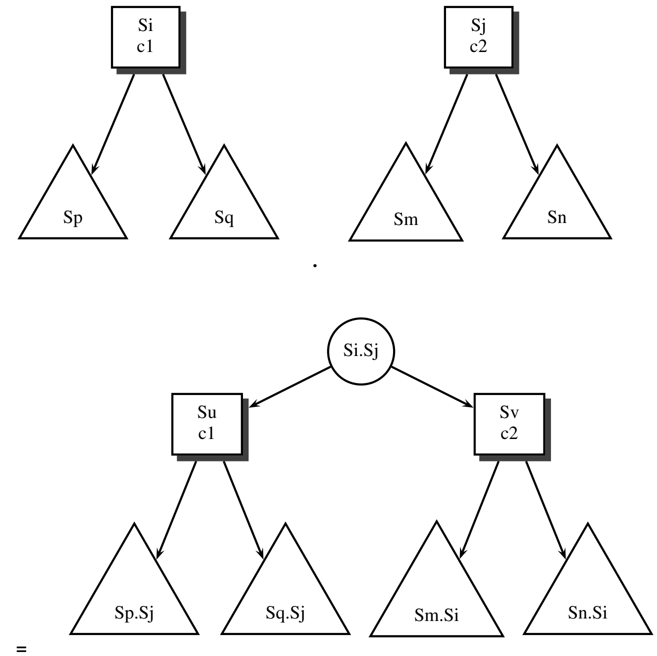

Figure 10. Merging two internal nodes

5.4.3 Merging an internal node with an external node

Let c_1 be an internal node and c_2 be an external. Accordingly, it doesn't make any sense to rendezvous on c_2 first and c_1 second as communication on c_2 is blocking, since both automata must communicate on c_2 simultaneously. The result of the merge turns out to be Figure 11

5.4.4 Merging two environment nodes

Consider Figure 12. The output of two environment nodes is simply a new environment with one outgoing edge for every pair of outgoing sub-automata.

5.4.5 Merging an environment node with an internal or external node

In Figure 13, *Sj* can either be an external or internal node. Each child of the environment node *Si* is merged with *Sj*.

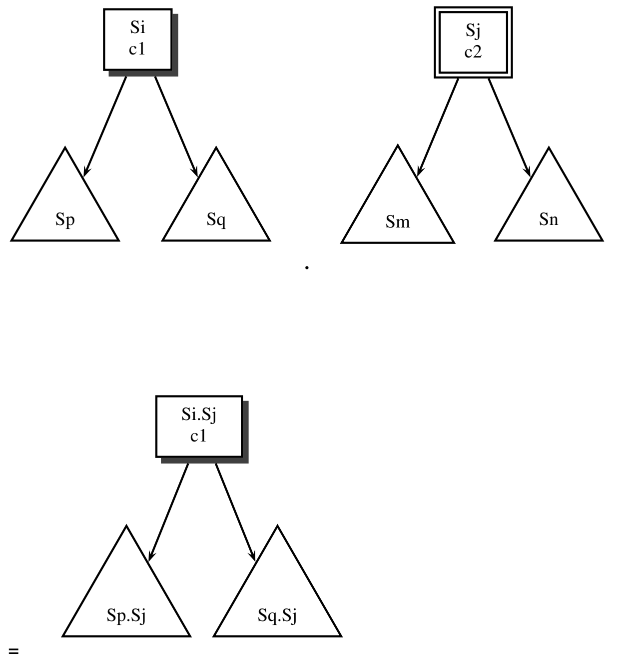

Figure 11. Merging an internal node with an external node

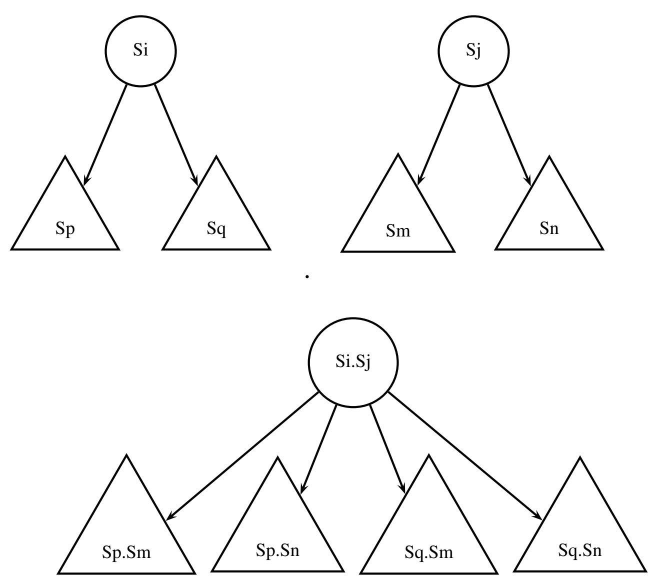

Figure 12. Merging two environment nodes

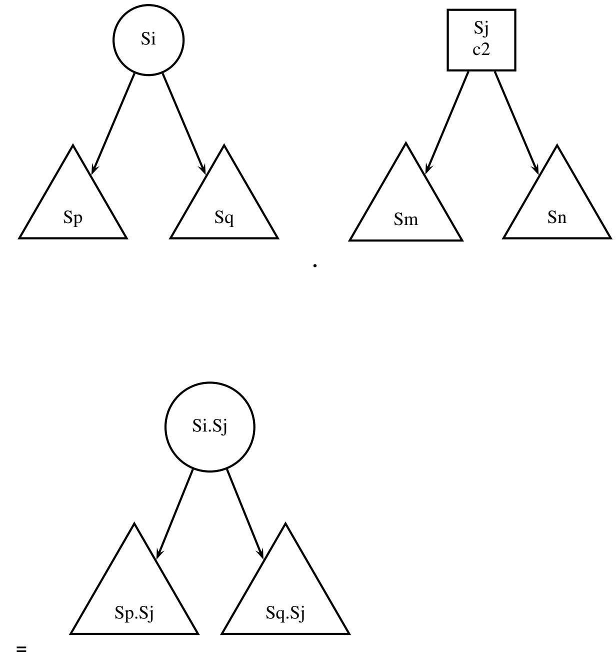

Figure 13. Merging an environment node with an internal or an external node

5.4.6 Merging two start nodes

When we merge, two start nodes, we end up with Figure 14.

5.4.7 Merging with a null node

Null Nodes are identity nodes. That is to say, $S_i \cdot S_2 = S_j \cdot S_i = S_j$ when $S_i = Null$.

5.4.8 Merging with a dead node

$S_i \cdot S_j = S_j \cdot S_i = Dead$ when either S_i or S_j is a dead node.

5.5 Simplification

As we are interested in deadlocks caused by blocking comunication along channels, we can simplify away much of the automaton which is a result of control structure.

- Once we merge two automata, we simply remove all start nodes, except for one, at the beginning of the automata. Since start nodes are just empty transitions,

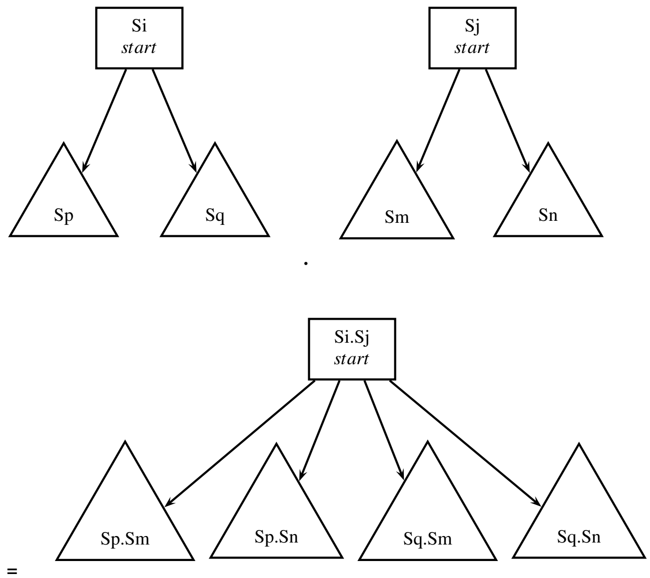

Figure 14. Merging two start nodes

we connect every incoming edge to an start node, directly to the start node's successors.

- Once we have merged all child nodes at a particular par statement, then we replace the par statement with this new merged automata. We also get rid of the start node that occurs at the beginning of the merged automaton, since the latter now is a part of the parent automata and does not stand alone.

- We can also remove all internal/external nodes whose channels are only local to the newly formed automata. The transitons affected by the removal should be updated though.

- If two states in an automata, have the same value and same output transitions, then we can merge the two states together, representing one state.

5.6 Finding deadlocks

Dead nodes occur as we form the product of automata and encounter deadlocks. Once we encounter a dead node, we propogate the death up the graph. We mark the predecessor of a dead node as also dead, provided the predecessor is not an environment node. Since environment nodes force any one of the outgoing transitions to be valid, an environment node dies only if all its children die. At the end of our merging procedure, if the automaton vanishes to a dead node, then we have encoutered a deadlock, else if an automaton can be formed representing all the tasks in the system, then the program is deadlock safe.

```
void main() {
  int i;
  chan int a, b;
  {
    for (i = 0 ; i < 100 ; i++) {
      if (i % 10)
        next b = 1;
      else
        next a = 0;
      next b = 10;
    }
  } par {
    next a;
    next b;
  }
}
```

Figure 15. Modification of Figure 15

6 The Example

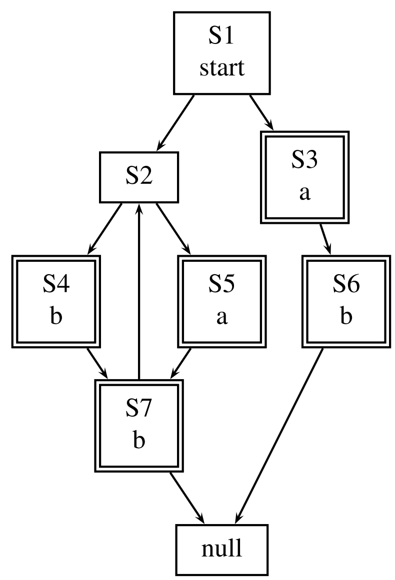

Figure 16. The automata for the example in Figure 15.

Figure 15 is a modification of Figure 3. We have replaced the first *next a* of Figure 3 with next b. Since we are abstracting the data values, there is equal probability of which branch is taken in the *in* statement. This program should report a deadlock, because there is a path in the

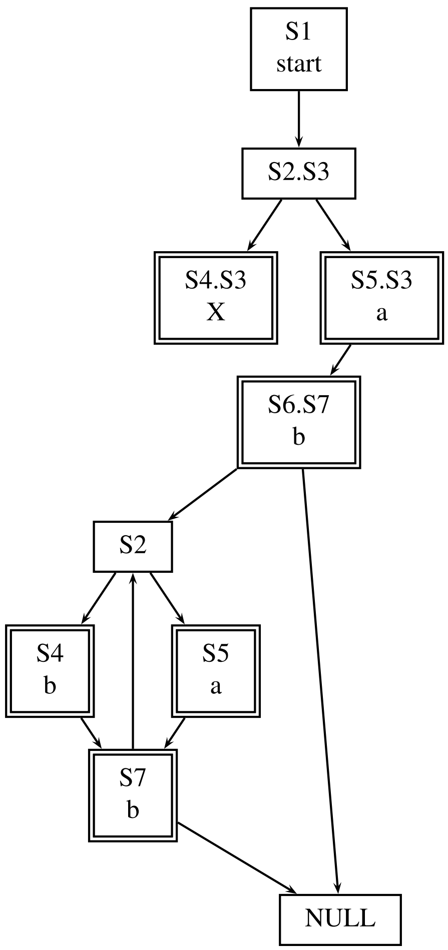

Figure 17. The automata from Figure 16 after merging.

program that deadlocks, although in reality the *else* branch is executed and the program does not deadlock during runtime.

7 Conclusions

We present a divide-and-conquer static deadlock detection technique for the SHIM concurrent language. We have expanded each SHIM program into a tree of tasks. We have abstracted each task as an automaton. We abstract data values. Then we combine tasks using our merge algorithm. Whenever merge fails, it signifies a deadlock path.

Since we abstract data-dependent control statements to reduce the state space, our algorithm can lead to false positives. We believe this is not a big limitation because programs can be rewritten with slight modifications to make it insensitive to data. Also, we have presented a technique to report inevitable deadlocks i.e, when a program can never escape from a deadlock.

We would like to build a tool for our algorithm and make it a standard part of the compilation process. We would also want to experimently compare our algorithm with the existing algorithm and report the pros and cons.

Tardieu and Edwards recently added exceptions to SHIM. We do not take them into account. This is a safe decision, because we may report a program as erroneous that throws exception to avoid a deadlcock but we never generate a false negative.

References

[1] S.A. Edwards, SHIM: A Language for Hardware/-Software Integration. In Proceedings of Synchronous Languages, Applications, and Programming (SLAP).

[2] S. A. Edwards and O. Tardieu, SHIM: A Deterministic Model for Heterogeneous Embedded Systems, In Proceedings of the ACM Conference on Embedded Software (Emsoft), Jersey City, NJ, September 2005.

[3] O. Tardieu and S.A. Edwards. R-SHIM: Deterministic Concurrency with Recursion and Shared Variables. In Proceedings of the 4rd International Conference on Formal Methods and Models for Codesign (MEMOCODE)