View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Columbia University Academic Commons

Static Deadlock Detection for the SHIM Concurrent Language

Nalini Vasudevan
Department of Computer Science
Columbia University
New York, USA
naliniv@cs.columbia.edu

Stephen A. Edwards
Department of Computer Science
Columbia University
New York, USA
sedwards@cs.columbia.edu

Abstract

Concurrent programming languages are becoming mandatory with the advent of multi-core processors. Two major concerns in any concurrent program are data races and deadlocks. Each are potentially subtle bugs that can be caused by non-deterministic scheduling choices in most concurrent formalisms. As an alternative, the SHIM concurrent language guarantees the absence of data races by eschewing shared memory, but a SHIM program may still deadlock if a program violates a communication protocol.

We present a model-checking-based static deadlock detection technique for the SHIM language. Although SHIM is asynchronous, its semantics allow us to model it synchronously without losing precision, greatly reducing the state space that must be explored. This plus the obvious division between control and data in SHIM programs makes it easy to construct concise abstractions.

Experimentally, we find our procedure runs in only a few seconds for modest-sized programs, making it practical to use as part of a compilation chain.

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

1. Introduction

Concurrent programs can exhibit many difficult-to-diagnose problems that are absent in all sequential programs. One such problem is deadlock: a program state where two or more tasks wait for each other indefinitely. Deadlock is never wanted, so finding and removing it is a necessary part of the concurrent programming process.

In this paper, we describe a way to statically detect deadlocks in SHIM [6, 18, 19]. Being an asynchronous concurrent language, SHIM's main novelty is scheduling-independence: regardless of non-deterministic scheduling choices taken by its runtime environment (e.g., an operating system), it is guaranteed to have the same input/output behavior. A SHIM system consists of asynchronously running sequential tasks that synchronize only when they communicate. SHIM combines the functional determinism of Kahn networks [14] with the rendezvous of Hoare's CSP [12].

A central goal of our work was to confirm that a careful choice of concurrent semantics simplifies the verification problem. In particular, while SHIM's semantics are asynchronous, they are constructed so that checking a (much simpler) synchronous abstraction remains sound. In particular, we do not need the power of a model checker such as Holtzmann's SPIN [13], which is designed to analyze an interleaving concurrency model.

The synchronous abstraction we use to check for deadlock is sound because of SHIM's scheduling independence: the choice of scheduling policy cannot affect the function of the program. In particular, a program either always or never deadlocks for a particular input—a scheduling choice cannot affect this. This means we are free to choose a particular scheduling policy without fear of missing or introducing deadlocks. Here, we choose a scheduling policy that amounts to a synchronous execution of a SHIM program. This greatly reduces the number of distinct states our model checker must consider, simplifying its job.

In this paper, we propose a technique that builds a synchronous abstraction of a SHIM program and uses the NuSMV symbolic model checker to determine whether the model can deadlock. Because of SHIM's semantics, if the model does not deadlock, the program is guaranteed not to deadlock, but because we abstract the data values in the program, the converse is not true: a program may not deadlock when we report it does.

By design, SHIM is a finite-state language provided it has no unbounded recursion (Edwards and Zeng [10] show how to remove bounded recursion), which makes the deadlock problem decidable. Unfortunately, exact analysis of SHIM programs can be impossible in practice because of state space explosion; we build sound abstractions instead.

Our main contribution is an illustration of how efficient, synchronous model-checking techniques can be used to analyze an asynchronous system. The result is a practical

static deadlock detector for a particular asynchronous language. While the theoretical equivalence of synchronous and asynchronous models has long been known, we know of few other tools that exploit it.

This work confirms that having designed SHIM's semantics to be scheduling-independent makes the language much easier to analyze with automated tools (elsewhere, we have argued that scheduling independence helps designers understand programs [6]). Few other concurrent languages were designed with formal verification in mind.

After reviewing related work, we describe the SHIM language and show how we abstract SHIM programs to make their deadlock properties easy to analyze. After that, we detail the generation of a NuSMV model and present experimental results that show our method is practical enough to run as part of a normal compilation flow. Overall, this suggests that careful language design can simplify the challenge of concurrent programming by making it easy to automatically check for certain problems.

2. Related Work

Avoiding deadlocks and data races in concurrent programs has been studied intensively; both static and dynamic techniques have been proposed. Detecting deadlocks in a running system is easy if global information is available. Distributed algorithms, such as Lee and Kim's [15], are more complicated, but computationally inexpensive. In this paper, we focus on the harder, more interesting problem of detecting potential deadlocks before a system runs since this is when we want to correct them.

As we propose in this paper, model checking is often used for static deadlock. Corbett [5] reviews a variety of static deadlock detection methods for concurrent Ada programs. He observes the main challenge is dealing with the state explosion from Ada's interleaved concurrency model. SHIM's scheduling-independent semantics sidesteps this problem. Taking a very different approach, Boyapati and Rinard [2] propose a static type system for an extended Java language based on programmer-declared ownership of each object. Their system guarantees objects are only accessed in a synchronized manner. SHIM guarantees unique ownership by simply prohibiting shared objects.

The interleaved concurrency model in most concurrent software environments is a challenge for model checkers. Many, such as SPIN [13], Improvsio [17], and Java PathFinder [20] use partial order reduction to reduce the number of states they much consider. While SHIM is asynchronous, its communication semantics do not require all possible interleavings to be considered, making the model checking problem much easier because we can check a synchronous model with far fewer states.

SHIM does not provide locks (although some of its implementations employ them) so it presents no danger of

```
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 exchange data on channels *a* and *b*

deadlock from bad locking policies. Hence lock-focused analysis, such as Bensalem et al. [1], which examines a single (non-deadlocking) trace for potential deadlock situations, is not applicable to SHIM.

3. The SHIM programming language

SHIM [19] is a C-like language with additional constructs for communication and concurrency. Specifically, *p par q* runs statements *p* and *q* in parallel, waiting for both to terminate before proceeding; *next c* is a blocking communication operator that synchronizes on channel *c* and sends data if it appears on the left side of an assignment and receives data when on the right. SHIM tasks communicate exclusively through this multi-way rendezvous; there are no global variables or other shared data.

In Figure 1, two peer tasks communicate on channels *a* and *b*. Tasks 1 and 2 are executed in parallel. SHIM interprets *next a* in task 1 as a send because it is on the left-hand side of the assignment. The *next a* of task 2 is a receive. The *next a* in task 1 waits for task 2 to receive the value. The tasks therefore rendezvous at their *next*s then continue to run after the communication takes place. Next, the two tasks rendezvous at *next b*. This time, task 2 sends and task 1 receives.

In Figure 2, the two tasks also attempt to communicate, but task 1 attempts to synchronize on *a* first, then *b*, while task 2 expects to synchronize on *b* first. This is a deadlock—each task will wait indefinitely for the other.

SHIM was designed for hardware/software codesign applications. It excels at describing stream-like algorithms, such as signal processing, but is poor for algorithms that re-

```
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

quire recursive data structures. By design, it is a finite-state language, so there is no way to allocate memory at runtime. In short, it was designed for embedded applications, not general-purpose programs designed to run on desktops or servers.

SHIM can be compiled efficiently. By design, most task code is C-like and imperative, so it can be compiled onto standard processors using familiar techniques. SHIM's communication and concurrency semantics are more subtle. SHIM has been compiled to shared-memory multiprocessors running the *pthreads* library [9], single-threaded processors without operating system support for threads [7], and even hardware translation has been proposed [8].

By design, SHIM is scheduling-independent in the same sense and for the same reasons as Kahn networks [14]. Specifically, the sequence of data values communicated across each channel is independent of the particular order in which tasks are executed, due to its block-on-a-single-channel semantics. Provided the scheduler blocks tasks that are waiting to rendezvous, any scheduling policy will produce the same result.

SHIM's scheduling independence means that the deadlock in Figure 2 will always occur, regardless of decisions made by the scheduler. This greatly simplifies the problem of checking for deadlocks since it suffices to check only one schedule. To minimize the number of states the model checker must explore, we check simple synchronous schedules in which each task rendezvous at most once per cycle.

4. Abstracting SHIM Programs

A sound abstraction is the central idea behind our deadlock detector for SHIM. A SHIM task alternates between computation and communication. Because tasks only interact when they communicate and never deadlock when they are computing, we abstract away the computation and assume a task always either communicates again or terminates, i.e., will never enter an infinite loop that never communicates. This is tantamount to assuming a schedule that perfectly balances the computation time of each process.

```
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

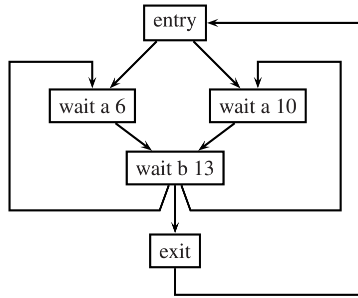
This assumption is optimistic in the sense that our tool may report a program is deadlock-free even if one of its tasks enters an infinite loop where it computes forever. However, checking for process termination can be done independently and can likely consider tasks in isolation. Answering the task termination question is outside the scope of this paper.

We also abstract away the details of data manipulation and assume all branches of any conditional statement can always be taken at any time. This is a conservative assumption that may increase the number of states we must consider. As usual, by ignoring data, we leave open the possibility that two tasks may appear to deadlock but in fact stay synchronized because of the data they exchange, but we believe this abstraction is a reasonable one and furthermore believe that system that depend on such behavior are probably needlessly difficult for the programmer to understand. In Section 7, we show an example of working code for which our tool reports a deadlock and how to restructure it to avoid the deadlock report.

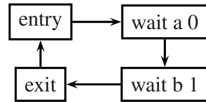
4.1. An Example

Consider the SHIM program in Figure 3. The *main* function starts two tasks that communicate through channels *a* and *b*. The first task communicates on channels *a* and *b* 100 times; the second task synchronizes on channels *a* and *b*, then terminates. This program does not deadlock because the communication patterns of the two tasks mesh properly.

```
main_1(chan int32 &a, chan int32 &b)
local int32 i
local int32 tmp0
local int32 tmp1
  i = 0
  goto continue
while:
  tmp1 = i % 10
  ifnot tmp1 goto else
  a = 1
  send a // 6
  goto endif
else:
  a = 0
  send a // 10
endif:
  b = 10
  send b // 13
  i = i + 1
continue:
  tmp0 = i < 100
  if tmp0 goto while
```




```
main_2(chan int32 a,
       chan int32 b)
  recv a // 0
  recv b // 1
```




```
main()
channel int32 a
channel int32 b
  main_1(a, b) : main_2(a, b);
```

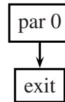


Figure 4. The IR and automata for the example in Figure 3. The compiler broke the *main* function into three tasks.

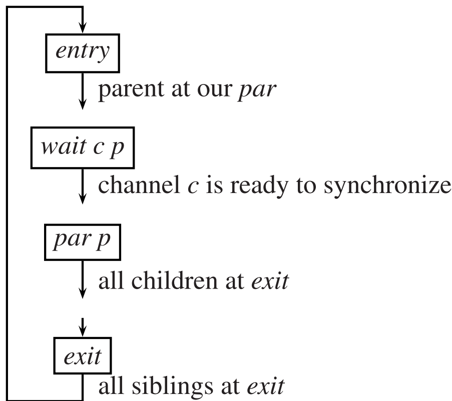


Figure 5. The four types of automaton states. Each has one *entry* and *exit*, but may have many *wait* and *par* states.

Note that SHIM semantics say that once a task terminates, it is no longer compelled to synchronize on the channels to which it is connected. So here, after the second task synchronizes on b and terminates, the first task talks to itself.

To abstract this program, our compiler begins by dismantling the SHIM program into a traditional, three-address-code-style IR (Figure 4). The main difference is that *par* constructs are dismantled into separate functions, here *main_1* and *main_2*, to ensure each function is sequential.

We assume the overall SHIM program is not recursive and remove statically bounded recursion using Edwards and Zeng [10]. We do not attempt to analyze recursive programs where the recursion cannot be bounded.

Next, we duplicate code to force each function to have a unique call site. While this has the potential for an exponential increase in code size, we did not observe it.

We remove trivial functions—those that do not attempt to synchronize. A function is trivial if it does not contain a *next* and all its children are trivial. Provided they terminate (an assumption we make), the behavior of such functions does not affect whether a SHIM program deadlocks. Fortunately, it appears that functions called in many places rarely contain communication (I/O functions are an exception), so the potential expansion from copying functions to ensure each has a unique call site rarely occurs in practice.

This preprocessing turns the call structure of the program into a tree, allowing us to statically enumerate all the tasks, the channels and their connections, and identify a unique parent and call site for each task (aside from the root).

Next, our tool creates an automaton that models the control and communication behavior for each dismantled function. Figure 4 shows automata and the code they model.

Each automaton consists of four kinds of states, shown in Figure 5. An automaton in its *entry* state waits for its parent to reach the *par* state at the automaton's call state. An automaton in its *exit* state waits for all its siblings to also be in their *exit* states. Each automaton (except the topmost one) starts in its *entry* state.

When an automaton enters a *par* state, it starts its children and waits for each of them to reach *exit* states. This is not a race because each child will advance from its *entry* state a cycle after the parent reaches the *par*. An automaton has one *par* state for each of its call sites. We label each with an integer that encodes the program counter.

Finally, *wait* states handle blocking communication. For an automaton to leave a *wait* state, all the running tasks that are connected to the channel (each *wait* corresponds to a unique channel) must also be at a *wait* for the channel. Note that an automaton may have more than one *wait* for the same channel; we label each with both the name of the channel and the program counter value at the corresponding *next*. The numbers 0, 1, 6, 10, and 13 in Figure 4 correspond to program counter values.

When we abstract a SHIM program, we ignore sequences of arithmetic operations; only conditionals, communication, and parallel function calls are preserved. Conditional branches, such as the test of *tmp1* in *main_1*, are modeled as non-deterministic choices.

We treat our automata as running synchronously, which amounts to imposing a particular scheduling policy on the program. SHIM's scheduling independence guarantees that we do not affect the functional behavior of the program by doing this. And in particular, the program can deadlock under any schedule if and only if it can deadlock under this schedule. This is what makes our abstraction of the program sound.

We do not explicitly model the environment in which the program is running; instead, we assume it is part of the program being tested. A sensor or actuator could be modeled as an independent SHIM process that is always willing to communicate: a source or a sink. More complicated restrictions on environmental behavior would have to be modeled by more SHIM processes. See our earlier paper [8] for a more thorough discussion of how to model in SHIM.

While we could build an explicit synchronous product automaton from the automata we build for each function, doing so would subject us to the state space explosion problem. Instead, we use a symbolic model checker that analyzes the product of these automata more efficiently.

5. Modeling Our Automata in NuSMV

To check whether our abstracted program (concurrently-running automata) deadlocks, we use the NuSMV [3] BDD- and SAT-based symbolic model checker. While it can analyze both synchronous and asynchronous finite-state systems, we only consider synchronous systems. The specifications to check can be expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL).

Using NuSMV involves supplying it with a model and a property of the model to be checked. We model each of our automata with two variables: one that represents the control state of the automaton and one that helps us determine when a deadlock has occurred. Figure 6 shows the code we generate for the three automata in Figure 4.

Translating a non-deterministic automaton into NuSMV is straightforward. We use the following template:

```
VAR state : {s1, s2, ... };
ASSIGN
  init(state) := s1;
  next(state) := case
    state = s1 & ... : {s2, s3, ...};
    state = s2 & ... : {s1, s3, ...};
    ...
    state = sn & ... : {s1, s2, ...};
    1 : state;
  esac;
```

```
MODULE main
DEFINE ready_a :=
  (main in {entry, exit} |
   main in {par_0} & (main_1 != exit & main_1 != entry |
                      main_2 != exit & main_2 != entry)) &
  main_1 in {entry, exit, wait_a_10, wait_a_6} &
  main_2 in {entry, exit, wait_a_0};
DEFINE ready_b :=
  (main in {entry, exit} |
   main in {par_0} & (main_1 != exit & main_1 != entry |
                      main_2 != exit & main_2 != entry)) &
  main_1 in {entry, exit, wait_b_13} &
  main_2 in {entry, exit, wait_b_1};
VAR main: {entry, exit, par_0};
ASSIGN init(main) := par_0;
  next(main) := case
  main = par_0 & main_2 = exit & main_1 = exit: exit;
  1: main;
esac;
VAR changed_main: boolean;
ASSIGN init(changed_main) := 1;
next(changed_main) := case
  main = par_0 & main_2 = exit & main_1 = exit: 1;
  1: 0;
esac;
VAR main_2: {entry, exit, wait_a_0, wait_b_1};
ASSIGN init(main_2) := entry;
next(main_2) := case
  main_2 = entry & main = par_0: wait_a_0;
  main_2 = wait_a_0 & ready_a: wait_b_1;
  main_2 = wait_b_1 & ready_b: exit;
  main_1 = exit & main_2 = exit: entry;
  1: main_2;
esac;
VAR changed_main_2: boolean;
ASSIGN init(changed_main_2) := 1;
next(changed_main_2) := case
  main_2 = entry & main = par_0: 1;
  main_2 = wait_a_0 & ready_a: 1;
  main_2 = wait_b_1 & ready_b: 1;
  main_1 = exit & main_2 = exit: 1;
  1: 0;
esac;
VAR main_1: {entry, exit, wait_a_10, wait_a_6, wait_b_13};
ASSIGN init(main_1) := entry;
next(main_1) := case
  main_1 = entry & main = par_0: {wait_a_10, wait_a_6, exit};
  main_1 = wait_a_6 & ready_a: wait_b_13;
  main_1 = wait_a_10 & ready_a: wait_b_13;
  main_1 = wait_b_13 & ready_b: {wait_a_10, wait_a_6, exit};
  main_1 = exit & main_2 = exit: entry;
  1: main_1;
esac;
VAR changed_main_1: boolean;
ASSIGN init(changed_main_1) := 1;
next(changed_main_1) := case
  main_1 = entry & main = par_0: 1;
  main_1 = wait_a_6 & ready_a: 1;
  main_1 = wait_a_10 & ready_a: 1;
  main_1 = wait_b_13 & ready_b: 1;
  main_1 = exit & main_2 = exit: 1;
  1: no;
esac;
SPEC AG(main != exit ->
        changed_main | changed_main_2 | changed_main_1)
SPEC EG(main != exit ->
        changed_main | changed_main_2 | changed_main_1)
```

Figure 6. NuSMV code for the program in Figure 3

For this automaton, *state* is a variable that can take on the symbolic values *s1*, *s2*, ... Each rule in the *case* statement is of the form *predicate*:*values*; and the predicates are prioritized by the order in which they appear.

Predicates are Boolean expressions over the state variables; values are sets of new values among which the model checker chooses non-deterministically. We model conditional branches in a SHIM program with non-deterministic choice. We generate one predicate/value pair for each state and start each predicate with a test for whether the machine is in that state. The final predicate/value pair is a default that holds the machine in its current state if it was not able to advance.

The NuSMV language has a rich expression syntax, but we only use Boolean connectives (& and |), comparison (=), and set inclusion (*in*).

For an automaton to leave its *entry* state, its parent must be in the *par* state for the automaton. By construction, both the parent automaton and the *par* state for a task is unique. For example, in Figure 6, the parent of *main_2* is *main*, and *main* calls *main_2* in the *par_0* state, so the rule for *main_2* to leave its *entry* state is

main_2 = *entry* & *main* = *par_0*: *wait_a_0*;

since in *main_2*, the successor to the *entry* state is *wait_a_0*.

For an automaton to leave a *par* state, all the children at the call site must be in their *exit* states. In Figure 6, *main_1* and *main_2* are invoked by *main* in the *par_0* state, so the complete rule for *main* to leave its *par_0* state is

main = *par_0* & *main_2* = *exit* & *main_1* = *exit*: *exit*;

since the successor of *par_0* in *main* is *exit*.

A state labeled *wait_c_p* represents a task waiting to synchronize on channel *c*. Since a task may wait on the same channel in many places, we also label it with a program counter value *p*. An automaton transitions from a *wait* state when all other automata that are compelled to rendenzvous have also reached matching *wait* states.

The rules for when rendezvous can occur on a channel are complicated because tasks do not always run. When a task connected to channel *c* is running children, it passes its connection to its children. However, if all its children connected to *c* terminate (i.e., reach their *exit* states) the parent resumes responsibility for synchronization and effectively blocks communication on *c* until it reaches a *wait* on *c*.

For each channel *c*, we define a variable *ready_c* that is true when a rendezvous is possible on the channel. We form it by taking the logical *and* of the rendezvous condition for each task that can connect to the channel (we know statically which tasks may connect to a particular channel).

For each task on a channel *c*, the rendezvous condition is true if the task is in the *entry* state (when it has not been started), in the *exit* state (when it ran and terminated, but its siblings have not terminated), in a *wait* state for the channel,

```
{ // task 8
  next a;
  { // task 3
    next a;
      next a; // task 1
    par
      next a; // task 2
  } par { // task 4
    next a;
    next b;
  } par { // task 5
    next b;
  }
  next a;
    next a; // task 6
  par
    next a; // task 7
}
```

```
(t_8 in {enter, exit, wait_a_2, wait_a_0} |
 t_8 in {par_3, par_1} & (t_7 != exit & t_7 != enter |
                          t_6 != exit & t_6 != enter |
                          t_4 != exit & t_4 != enter |
                          t_3 != exit & t_3 != enter)) &
(t_3 in {enter, exit, wait_a_0} |
 t_3 in {par_1} & (t_2 != exit & t_2 != enter |
                   t_1 != exit & t_1 != enter)) &
t_7 in {enter, exit, wait_a_0} &
t_6 in {enter, exit, wait_a_0} &
t_4 in {enter, exit, wait_a_0} &
t_2 in {enter, exit, wait_a_0} &
t_1 in {enter, exit, wait_a_0}
```

Figure 7. A SHIM code fragment and the conditions for rendezvous on the *a* channel

or in a *par* state when at least one of its children connected to *c* is still running (i.e., when the parent has not recovered its responsibility for the channel *c* from its children).

Figure 7 illustrates the rendezvous condition for a fairly complex set of tasks. Tasks 1, 2, 4, 5, 6, and 7 are leaves—they do not call other tasks. For each, the condition is that it be terminated or at a *wait* state for the channel.

Task 3 both synchronizes directly on *a* and invokes tasks 1 and 2. Its condition is that it be terminated, at its *wait* state for *a*, or that it be at its *par* state and at least one of task 1 or 2 be running.

Task 8 is the most complex. It synchronizes on *a* in two states (*wait_a_0* and *wait_a_2*) and has two *par* calls. At either of the *par* calls, at least one of its four children (tasks 3, 4, 6, and 7) must be running.

Note that since task 5 is not connected to channel *a*, its state is not considered.

Example	Lines	Channels	Tasks	Result	Runtime	Memory	States
Source-Sink	35	2	11	No Deadlock	0.2 s	3.9 MB	97
Pipeline	30	7	13	No Deadlock	0.1	2.0	95
Prime Number Sieve	35	51	45	No Deadlock	1.7	25.4	3.2×10^9
Berkeley	40	3	11	No Deadlock	0.2	7.2	139
FIR Filter	100	28	28	No Deadlock	0.4	13.4	4134
Bitonic Sort	130	65	167	No Deadlock	8.5	63.8	25
Framebuffer	220	11	12	No Deadlock	1.7	11.6	9593
JPEG Decoder	1020	7	15	May Deadlock	0.9	85.6	571
JPEG Decoder Modified	1025	7	15	No Deadlock	0.9	85.6	303

Table 1. Experimental results

6. Finding Deadlocks

We define a deadlock as a state in which at least one task is running yet no task can advance because they are all waiting on other tasks. In particular, we do not consider it a deadlock when a small group of tasks continue to communicate with each other but not the rest of the system, which remains blocked.

For each automaton, we generate an additional state bit (*changed*) that tracks whether it will proceed in this cycle or is blocked waiting for communication. Using additional state bits is unnecessary; in our first attempt we performed the check combinationally (i.e., in each state checked whether there was at least one task that could advance). However, introducing additional state bits improved the running time, so we adopted that style.

The rules we use for setting the *changed* bit for each automaton are similar to those for the automaton. The predicates are exactly the same, but instead of setting the state, the values set the *changed* bit to 1.

Once we have an *changed* bit for each automaton, the test for the absence of deadlock is simple: either at least one task was able to advance or the topmost task has terminated. This is easy to express in temporal logic for NuSMV:

AG(*root* != *exit* −> *changed_t1* | *changed_t2* | ...)

where *root* is the state of the topmost task and *advanced_tx* indicates that task *x* was able to make progress. In words, this says that in each state if the topmost task has not terminated then at least one task was able to make progress.

We also check whether a program will inevitably deadlock: if all possible paths lead to a deadlock state irrespective of the conditional predicates, then the program absolutely will deadlock. The temporal logic expression for its absence in NuSMV is

EG(*root* != *exit* −> *changed_t1* | *changed_t2* | ...)

I.e., in each state, if the topmost task is running, there is some path where at least one task was able to advance.

7. Results

We ran our deadlock detector on various SHIM programs on a 3 GHz Pentium 4 machine with 1 GB RAM. Table 1 lists our results. The Lines column shows for each example the number of lines of code including comments. The Channels and Tasks columns list the number of channels and concurrently running tasks we find after expanding the tasks into a tree and removing non-trivial tasks. Runtimes include the time taken for compilation, abstraction, generating the NuSMV model, and running the NuSMV model checker. We check for both the possibility and inevitability of a deadlock. As expected, the model checking time dominates on the larger examples. The Memory column reports the total resident set size used by the verifier. The States column reports the number of reachable states NuSMV found.

Source-Sink is a simple example centered around a pair of processes that pass data through a channel and print the results through an output channel. The large number of tasks arise from I/O functions used to print the output of this test case. Most of the examples here include many extra tasks for this reason.

Pipeline and the Prime Number Sieve are examples from Edwards and Zeng [10] that use bounded recursion. As mentioned earlier, we use their technique to remove the recursion before running NuSMV. The Sieve has many states because most of its tasks perform data-dependent communication and our model ends up considering all apparent possibilities, even though the system enters far fewer states in practice. Nevertheless, this illustrates the power of symbolic simulation: analyzing these three billion states takes NuSMV less than two seconds.

The Berkeley example contains a pair of tasks that communicate packets through a channel using a data-based protocol. After ignoring data, however, the tasks behave like simple sources and sinks, making it easy to prove the absence of deadlock.

The FIR filter is a parallel five-tap filter with twenty-eight tasks and channels (the core consists of seventeen tasks). Each task's automaton consists of a single loop (the

filter is actually a synchronous dataflow model [16]) so the analysis is fairly easy.

Bitonic Sort is one of our most parallel examples: it uses twenty-four comparison tasks to order eight integers. All the additional tasks are sources, sinks, and (repeated calls to I/O routines). The communication behavior of the tasks is straightforward, but the tool has many tasks to consider.

Framebuffer is a 640×480 video framebuffer driven by a line-drawing task. Its communication is complicated.

The JPEG decoder is one of our largest applications to date, and illustrates some of the challenges in coding SHIM to avoid deadlocks. Our tool reported the possibility of a deadlock on the initial version (which actually works correctly) because of the code in Figure 8.

This code attempts to run three IDCT processors in parallel on an array of n macroblocks. For all but the last iteration of the loop, the dispatcher communicates on channels *I1*, *I2*, and *I3*, then receives data from *O1*, *O2*, and *O3*. However, since n may not be a multiple of three, this code is careful not to overrun the array bounds and may only perform one or two IDCT operations in the last iteration.

While this program works (provided the predicates on the *if* statements are written properly), our tool does not understand, say, the second and fourth conditionals are correlated and reports a potential deadlock.

Figure 9 illustrates how to avoid this problem by duplicating code and factoring it differently. Although our tool still treats the conditional branches as non-deterministic, it does not perceive a deadlock because, e.g., the synchronizations on I3 and O3 remain matched.

Figure 8, however, will not report an unavoidable deadlock because it has a non-deadlocking path.

Overall, our tool is able to quickly check these programs (in seconds) while using a reasonable amount of memory. While larger programs will be harder to verify, our technique is clearly practical for modestly sized programs.

8. Conclusions

We presented a static deadlock detection technique for the SHIM concurrent language. SHIM programs behave identically regardless of scheduling policy because they are based on Kahn networks [14]. This allows us to check for deadlock on synchronous models of SHIM programs and know the result holds for asynchronous implementations.

We expand each SHIM program into a tree of tasks, abstract each task as an automaton that performs communication and invokes and waits for its children, then express these automata in a form suitable for the NuSMV symbolic model checker. This is a mechanical process.

We abstract away data-dependent decisions when building each task's automaton. This both greatly simplifies their structure and can lead to false positives: our technique can report a program will deadlock even though it cannot. How-

```
{
  // ...
  for (int j = 0 ; j < n ; j += 3) {
    next I1 = iblock[j];
    if (j+1 < n) {
      next I2 = iblock[j+1];
      if (j+2 < n)
        next I3 = iblock[j+2];
    }
    oblock[j] = next O1;
    if (j+1 < n) {
      oblock[j+1] = next O2;
      if (j+2 < n)
        oblock[j+2] = next O3;
    }
  }
 // ...
} par {
  for (;;)
    next O1 = IDCT(next I1);
} par {
  for (;;)
    next O2 = IDCT(next I2);
} par {
  for (;;)
    next O3 = IDCT(next I3);
}
```

Figure 8. Fragment of the JPEG Decoder that causes our tool to report a deadlock; it ignores the correlation among *if* statements

```
for(int j = 0 ; j < n ; j += 3) {
  next I1 = iblock[j];
  if (j+2 < n) {
    next I2 = iblock[j+1];
    next I3 = iblock[j+2];
    oblock[j] = next O1;
    oblock[j+1] = next O2;
    oblock[j+2] = next O3;
  } else if (j+1 < n) {
    next I2 = iblock[j+1];
    oblock[j] = next O1;
    oblock[j+1] = next O2;
  } else {
    oblock[j] = next O1;
  }
}
```

Figure 9. An equivalent version of the first task in Figure 8 for which our tool does not report a deadlock

ever, we believe this is not a serious limitation because there is often an alternative way to code a particular protocol that makes it insensitive to data and more robust to small modifications, i.e., less likely to be buggy.

Experimentally, we find NuSMV is able to detect or prove the absence of deadlock in seconds for modestly sized examples. We believe this is fast enough to make deadlock checking a standard part of the compilation process (i.e., instead of something too costly to run more than occasionally), which we believe is a first for concurrent languages.

Tardieu and Edwards recently added concurrent, deterministic exceptions to the SHIM model [19], which are a convenient mechanism for task control. We currently ignore them, which is safe but as a result, we may report as erroneous programs that throw exceptions to avoid a deadlock situation. While we do not know of any such programs, we plan to consider this issue in the future.

While we found our data abstraction technique works well on the examples we tried, we suspect it will cause too many programs to be rejected. We plan to explore counterexample guided abstraction refinement [4] to see whether we can further improve the precision of our analysis.

Finally, we also plan to explore more modular analysis techniques, perhaps related to the stuck-free property [11], that would make interaction without deadlocks something that could be checked as part of a type system.

Acknowledgments

Alfred Aho (Columbia University), Franjo Ivancic (NEC Labs), and Michael Theobald (D.E. Shaw Research) provided valuable suggestions and feedback. This work was supported by NSF grant CNS–0720292.

References

[1] S. Bensalem, J.-C. Fernandez, K. Havelund, and L. Mounier. Confirmation of deadlock potentials detected by runtime analysis. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD)*, pages 41–50, Portland, Maine, July 2006.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Nov. 2002.

[3] A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An OpenSource tool for symbolic model checking. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364, Copenhagen, Denmark, July 2002.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, Chicago, Illinois, July 2000.

[5] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, Mar. 1996.

[6] S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, Sept. 2005.

[7] S. A. Edwards and O. Tardieu. Efficient code generation from SHIM models. In *Proceedings of Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 125–134, Ottawa, Canada, June 2006.

[8] S. A. Edwards and O. Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(8):854–867, Aug. 2006.

[9] S. A. Edwards, N. Vasudevan, and O. Tardieu. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, pages 1498–1503, Munich, Germany, Mar. 2008.

[10] S. A. Edwards and J. Zeng. Static elaboration of recursion for concurrent software. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, San Francisco, California, Jan. 2008.

[11] C. Fournet, T. Hoare, S. K. Rajamani, and J. Rehof. Stuck-free conformance. In *Proceedings of the International Conference on Computer-Aided Verification (CAV)*, volume 3114 of *Lecture Notes in Computer Science*, pages 242–254, Boston, Massachusetts, USA, July 2004.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, Upper Saddle River, New Jersey, 1985.

[13] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.

[14] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, Aug. 1974. North-Holland.

[15] D. Lee and M. Kim. A distributed scheme for dynamic deadlock detection and resolution. *Information Sciences*, 64(1–2):149–164, 1992.

[16] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sept. 1987.

[17] F. Lerda, N. Sinha, and M. Theobald. Symbolic model checking of software. In B. Cook, S. Stoller, and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[18] O. Tardieu and S. A. Edwards. R-SHIM: Deterministic concurrency with recursion and shared variables. In *Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, page 202, Napa, California, July 2006.

[19] O. Tardieu and S. A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 142–151, Seoul, Korea, Oct. 2006.

[20] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, Apr. 2003.