Compositional Deadlock Detection for Rendezvous Communication

Baolin Shao
Columbia University
New York, New York
bshao@cs.columbia.edu

Nalini Vasudevan
Columbia University
New York, New York
naliniv@cs.columbia.edu

Stephen A. Edwards
Columbia University
New York, New York
sedwards@cs.columbia.edu

ABSTRACT

Concurrent programming languages are growing in importance with the advent of multi-core systems. However, concurrent programs suffer from problems, such as data races and deadlock, absent from sequential programs. Unfortunately, traditional race and deadlock detection techniques fail on both large programs and small programs with complex behaviors.

In this paper, we present a compositional deadlock detection technique for a concurrent language—SHIM—in which tasks run asynchronously and communicate using synchronous CSP-style rendezvous. Although SHIM guarantees the absence of data races, a SHIM program may still deadlock if the communication protocol is violated. Our previous work used NuSMV, a symbolic model checker, to detect deadlock in a SHIM program, but it did not scale well with the size of the problem. In this work, we take an incremental, divide-and-conquer approach to deadlock detection.

In practice, we find our procedure is faster and uses less memory than the existing technique, especially on large programs, making our algorithm a practical part of the compilation chain.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Algorithms, Verification

Keywords

SHIM, concurrency, static analysis, deadlock, divide-and-conquer

1. Introduction

Today's parallel hardware demands concurrent programming languages, yet because of computer science's long neglect of this difficult subject, most concurrent languages provide only error-prone low-level concurrency primitives such as locks and shared memory. In such a setting, the language and compiler can provide few assurances of correctness, leaving the burden on the already overworked programmer.

We believe concurrent programming languages must provide higher-level abstractions that come with assurances of correctness if they are to be anywhere as easy to use as their polished sequential counterparts. This work is part of an ongoing project designed to test the viability of this approach.

Our concurrent SHIM model and language [14, 25] provides certain program correctness guarantees and makes others easy to check by adopting CSP's rendezvous [17] in a Kahn network [19] setting. In particular, it prevents data races by providing scheduling independence: given the same input, a program will produce the same output regardless of what scheduling choices its runtime environment makes.

SHIM's scheduling independence makes other properties easier to check because they do not have to be tested across all schedules; one is enough. Deadlock is one such property: for a particular input, a program will either always or never deadlock; scheduling choices cannot cause or prevent a deadlock. We exploited this property in earlier work [27], where we transformed asynchronous SHIM models into synchronous state machines and used the symbolic model checker NuSMV [8] to verify the absence of deadlock. This is unlike traditional techniques, such as Holzmann's SPIN model checker [18], in which all possible interleavings must be considered. While our technique was fairly effective because it could ignore interleavings, we improve upon it here.

In this paper, we use explicit model checking with a form of assume-guarantee reasoning [24] to quickly detect the possibility of a deadlock in a SHIM program. Step by step, we build up a complete model of the program by forming the product machine of an automaton we are accumulating with another process from the program, each time checking the accumulated model for deadlock.

Our key trick: we simplify the accumulated automaton after each step, which often avoids exponential state growth. Specifically, we abstract away internal channels—those that do not appear in any other processes.

Figure 1 shows our technique in action. Starting from the (contrived) program, we first abstract the behavior of the first two tasks into simple automata. The first task communicates on channel a, then on channel b, then repeats; the second task does the same on channels b and c. We compose these automata by allowing either to take a step on unshared channels but insisting on a rendezvous when a channel is shared. Then, since channel b is local to these two tasks, we abstract away its behavior by merging two states. This produces a simplified automaton that we then compose with the automaton for the third task. This time, channel c is local, so again we simplify the automaton and compose it with the automaton for the fourth task.

The automaton we obtained for the first three tasks insists on communicating first on a then d; the fourth tasks communicates on d then a. This is a deadlock, which manifests itself as a state with no outgoing arcs.

EMSOFT'09 October 12–16 2009 Grenoble France.

```
void main()
{
  chan int a, b, c, d;
      for(;;) {
    recv a; b = a + 1; send b;
  } par for(;;) {
    recv b; c = b + 1; send c;
  } par for(;;) {
    recv c; d = c + 1; send d;
  } par for(;;) {
    recv d; a = d + 1; send a;
  }
}
```

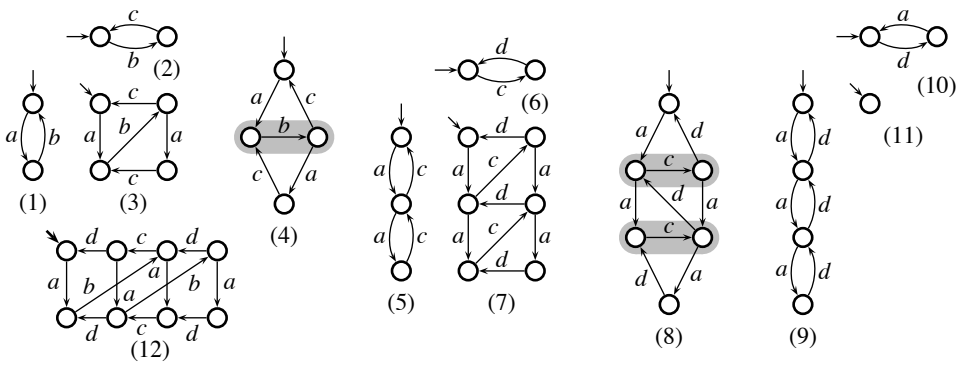


Figure 1: Analyzing a four-task SHIM program. Composing the automata for the first (1) and second (2) tasks gives a product automaton (3). Channel *b* only appears in the first two tasks, so we abstract away its effect by identifying (4) and merging (5) equivalent states. Next, we compose this simplified automaton (5) with that for the third task (6) to produce another (7). Now, channel *c* will not appear again, so again we identify (8) and merge (9) states. Finally, we compose this (9) with the automaton for the fourth task (10) to produce a single, deadlocked state (11) because the fourth task insists on communicating first on *d* but the other three communicate first on *a*. The direct composition of the first three tasks without removing channels (12) is larger—eight states.

For programs that follow such a pipeline pattern, the number of states grows exponentially with the number of pipeline stages (precisely, n stages produce 2^n states), yet our analysis only builds machines with $2n$ states before simplifying them to $n+1$ states at each step. Although we still have to step through and analyze each of the n stages (leading to quadratic complexity), this is still a substantial improvement.

Of course, our technique cannot always reduce an exponential state space to a polynomial one, but we find it often did on the example programs we tried.

In the rest of this paper, we introduce the SHIM programming language and how we model SHIM programs (Section 2), then show how we check these models for deadlock (Section 3) following our compose-and-abstract procedure described above. We present experimental results in Section 4 that shows our technique is superior to our earlier work using a symbolic model checking, review related work in Section 5, and conclude in Section 6.

2. SHIM Programs

SHIM [26] 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; *send c* and *recv c* are blocking communication operators that synchronize on channel *c*. SHIM tasks communicate exclusively through this multi-way rendezvous; there are no global variables or other shared data.

In Figure 2(a), two peer tasks communicate on channels *a* and *b*. Tasks 1 and 2 run in parallel. The *send a* in task 1 waits for task 2 to receive the value. The tasks rendezvous then continue after the communication takes place. Next, the two tasks rendezvous at *b*. This time, task 2 sends and task 1 receives.

In Figure 2(b), 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.

If the compiler finds two senders on a particular channel, the compiler rejects the program to guarantee determinism.

It is easy to manually see deadlocks in small programs like Figure 2(b); it is much harder in programs with nested *par* statements, large number of tasks, or programs with thousands of lines of code.

```
void main()
{
 chan int a, b;
 { // Task 1
   a = 5;
   send a; // Send 5 on a
   // now a = 5
   recv b; // Receive b
   // now b = 10
 } par { // Task 2
   recv a; // Receive a
   // now a = 5
   b = 10
   send b; // Send 10 on b
   // now b = 10
  }
}
```

(a)

```
void main()
{
  chan int a, b;
  {
    a = 5;
    send a; // Deadlocks here

    recv b;

  } par {
    b = 10;
    send b; // Deadlocks here

    recv a;

  }
}
```

(b)

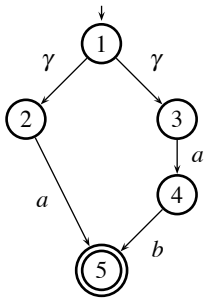
Figure 2: (a) A SHIM program in which two tasks exchange data on channels *a* and *b* and (b) one with deadlocks.

2.1 Modeling SHIM Programs

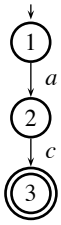
A sound abstraction not only simplifies the model of a program but also greatly reduces the complexity of analysis. To detect deadlock, we abstract away a program's computation and focus on communication patterns. By doing this, we assume that any branch of a conditional statement can be taken at any time, which is sound but may lead us to find false deadlocks. The improvement in speed more than makes up for the loss of precision, though. This abstraction is sound: if we report a program is deadlock-free, there is no way it can reach a deadlocked state.

After abstraction, we construct an SHIM automaton for each task and compose them one at a time to detect deadlock. We show how this works for a simple program in Section 2.2, then we provide a formalism for our model and our algorithms for compositional deadlock detection. Finally, we illustrate the difference between our approach and more traditional methods with a set of examples.

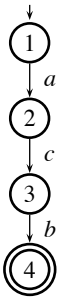
```
void main()
{
  int i;
  chan int a, b;

  { // Task 1
    if (i % 10) {
      a = 1;
      send a;
    } else {
      a = 0;
      send a;
      recv b;
    }

  } par { // Task 2
    recv a;
    c = 1;
    send c;

  } par { // Task 3
    recv a;
    recv c;
    b = 10;
    send b;
  }
}
```



Task 1



Task 2



Task 3

Figure 3: A SHIM program and the automata for its tasks

2.2 An Example

Consider the SHIM program in Figure 3. The *main* function starts three tasks that communicate through channels a, b and c. The first task has a conditional statement, which we model as a nondeterministic choice. One of its branches synchronizes on channel a. The other branch synchronizes on both a and b. The second task synchronizes on channels a and c; the third task synchronizes on channels a and c, and then on b. The ownership is as follows: channel a is shared by all three tasks, channel b is shared by task 1 and task 3, and channel c is shared by tasks 2 and 3. This program does not deadlock. First all three tasks synchronize on channel a exhibiting multiway-rendezvous. Next, tasks 2 and 3 rendezvous on channel c. Task 3 then synchronizes with task 1 on channel b if the branch is not taken. Otherwise, it waits for task 1 to terminate and then does a dummy send on channel b. This is because task 3 is no longer compelled to wait for a terminated process (task 1).

We assume the overall SHIM program is not recursive. We remove statically bounded recursion using Edwards and Zeng [15] and do not attempt to analyze programs with unbounded recursion.

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 attempt to send or receive 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 explosion from copying functions to ensure each has a unique call site rarely occurs in practice.

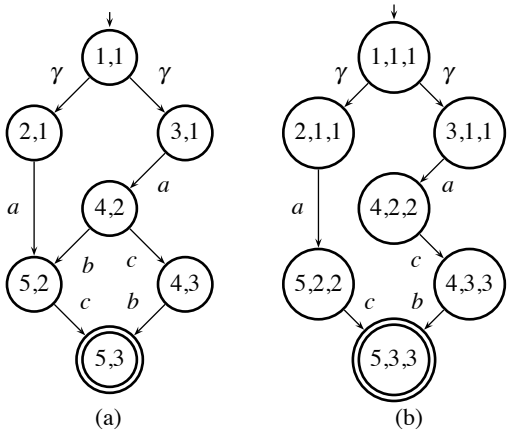


Figure 4: Composing (a) the automata for tasks 1 and 2 from Figure 3 and (b) composing this with task 3.

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).

After preprocessing, we build a SHIM automaton for each task from the compiler's intermediate representation. A SHIM automaton has two kinds of arcs: channel and γ. A transition labeled with a channel name represents communication on that channel; a γ transition models conditionals (nondeterministic choices).

Figure 3 shows the three SHIM automata we construct for the program. The *if-else* in task 1 is modeled as state 1 with two outgoing γ transitions. On the other hand, we use arcs labeled by channels to represent communication.

Figure 4(a) shows the composition of tasks 1 and 2 from Figure 3. First, we compose task 1's state 1 with task 2's state 1. We create the (1,1) state with two outgoing γ transitions, and we then compose each of state 1's successor in task 1 with state 1 of task 2, generating states (2,1) and (3,1). At state (2,1), we can say that task 1 is at state 2 and task 2 is at state 1. We then add a transition from (2,1) to (5,2) labeled a because both tasks are ready to communicate on a in state (2,1). Similarly, we create state (4,2).

Then, at state (4,2), task 1 can fire b (in the absence of task 3) and task 2 can fire c. Since task 1 shares channel b but not c and task 2 shares channel c but not b, either transition is possible so we have two scheduling choices at state (4,2), which is represented by two transitions b and c from (4,2). By similar rules, we compose other states and finally we end up with Figure 4(a) as the result. The composed automaton owns channels a, b, and c.

Following the same procedure, we compose the automaton in Figure 4(a) with task 3 in Figure 3 to produce the automaton in Figure 4(b). We compose states in a similar fashion. However, when composing state (4,2) of Figure 4(a) with state 2 of task 3 in Figure 3, state (4,2)'s transition on channel b is not enabled because task 3 does not have a transition on b from its state 2. On the other hand, state (4,2)'s transition on channel c does not conflict with task 3, allowing us to transit from state (4,2,2) to state (4,3,3) on channel c in Figure 4(b).

3. Compositional Deadlock Detection

3.1 Notation

Below, we formalize our abstraction of SHIM programs. We wanted something like a finite automaton that could model the external behavior of a SHIM process (i.e., communication patterns).

We found we had to distinguish two types of choices: a nondeterministic choice induced by concurrency that can be made by the scheduler (i.e., selecting one of many enabled tasks) and control-flow choices made by the tasks themselves. Although a running task is deterministic (it makes decisions based purely on its state, which can be supplied in part by the [deterministic] series of data that arrive on its channels), we chose to abstract data computations to simplify the verification problem at the expense of rejecting some programs that would avoid deadlock in practice. Thus, we treat choices made by a task (e.g., at an if-else construct) as nondeterministic.

These two types of nondeterministic choices must be handled differently when looking for deadlocks: while it is acceptable for an environment to restrict choices that arise from concurrency, an environment cannot restrict choices made by the tasks themselves.

Our solution is an automaton with two types of edges: those labeled with channels representing communication, which need not all be followed when composing automata; and those labeled with γ, which we use to represent an internal choice made by a task and must be preserved when composing automata.

DEFINITION 1. *A SHIM automaton a 6-tuple* $(Q, \Sigma, \gamma, \delta, q, f)$ *where* Q *is the set of states,* Σ *is the set of channels,* $\gamma \notin \Sigma$, $q \in Q$ *is the initial state,* $f \in Q$ *is the final state, and* $\delta = Q \times (\Sigma \cup \{\gamma\}) \to 2^Q$ *the transition function, where* $|\delta(s,c)| = 0$ *or* 1 *for* $c \neq \gamma$.

The δ transition function is key. For each state $s \in Q$ and channel $c \in \Sigma$, either $\delta(s,c) = \emptyset$ and the automaton is not ready to rendezvous on channel c in state s, or $\delta(s,c)$ is a singleton set consisting of the unique next state to which the automaton will transition if the environment rendezvous on c.

The special "channel" γ denotes computation internal to the system. If $\delta(s,\gamma) \neq \emptyset$, the automaton may transition to any of the states in $\delta(s,\gamma)$ from state s with no rendezvous requirement on the environment.

A state $s \in Q$ such that $\delta(s,c) = \emptyset$ for all $c \in \Sigma \cup \{\gamma\}$ corresponds to the system terminating normally if $s = f$ and is a deadlock state otherwise.

Next, we define how to run two SHIM automata in parallel. The main thing is that we require both automata to rendezvous on any shared channel.

DEFINITION 2. *The composition* $T_1 \cdot T_2$ *of two SHIM automata* $T_1 = (Q_1, \Sigma_1, \gamma, \delta_1, q_1, f_1)$ *and* $T_2 = (Q_2, \Sigma_2, \gamma, \delta_2, q_2, f_2)$ *is* $(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \gamma, \delta, \langle q_1, q_2\rangle, \langle f_1, f_2\rangle)$, *where*

$$\delta(\langle p_1, p_2\rangle, \gamma) = \big(\delta_1(p_1,\gamma) \times \{p_2\}\big) \cup \big(\{p_1\} \times \delta_2(p_2,\gamma)\big),$$

and for $c \in \Sigma_1 \cup \Sigma_2$,

$$\delta(\langle p_1, p_2\rangle, c) = \begin{cases} \delta_1(p_1,c) \times \delta_2(p_2,c) & \text{when } c \in \Sigma_1 \cap \Sigma_2; \\ \delta_1(p_1,c) \times \{p_2\} & \text{when } c \in \Sigma_1 - \Sigma_2 \text{ or} \\ & p_2 = f_2; \text{ and} \\ \{p_1\} \times \delta_2(p_2,c) & \text{when } c \in \Sigma_2 - \Sigma_1 \text{ or} \\ & p_1 = f_1 \end{cases}$$

Here, we defined two cases for the composed transition function. On γ (corresponding to an internal choice), either the first automaton or the second may take any of its γ transitions independently, hence the set union. Note that $\emptyset \times \{p_2\} = \emptyset$.

For normal channels, there are two cases. For a shared channel ($c \in \Sigma_1 \cap \Sigma_2$), both automata proceed simultaneously if each has a transition on that channel, i.e., have rendezvoused. For non-shared channels or if one of the tasks has terminated, the automaton connected to the channel may take a step independently (and implicitly assumes the environment is willing to rendezvous on the channel).

There should be no difference between running T_1 in parallel with T_2 and running T_2 in parallel with T_1, yet this is not obvious from the above definition. Below, we formalize this intuition by defining what it means for two automata to be equivalent, then showing the composition operator produces equivalent automata.

Algorithm 1 compose(automata list L)

```
1:  T_1, ..., T_n = reorder(L)
2:  T = T_1
3:  for i = 2 to n do
4:      T = T · T_i                          {Compose using Definition 2}
5:      q = initial state of T
6:      for all channels c in T that are not in T_{i+1}, ..., T_n do
7:          for all δ(p,c) = {q} do
8:              Set δ(p,c) to ∅               {p is the parent of q}
9:              Add q to δ(p,ε)
10:             Add p to δ(q,ε)
11:     T = subset-construction(T)           {Remove ε transitions}
12: if T has a deadlock state then
13:     return deadlock
14: else
15:     return no-deadlock
```

DEFINITION 3. *Two SHIM automata* $T_1 = (Q_1, \Sigma_1, \gamma, \delta_1, q_1, f_1)$ *and* $T_2 = (Q_2, \Sigma_2, \gamma, \delta_2, q_2, f_2)$ *are* equivalent *(written* $T_1 \equiv T_2$*) if and only if* $\Sigma_1 = \Sigma_2$ *and there exists a bijective function* $b : Q_1 \to Q_2$ *such that* $q_2 = b(q_1)$, $f_2 = b(f_1)$, *and for every* $s \in Q_1$ *and* $c \in \Sigma_1 \cup \{\gamma\}$, $\delta_2(b(s),c) = b(\delta_1(s,c))$.

LEMMA 1. *Composition is commutative:* $T_1 \cdot T_2 \equiv T_2 \cdot T_1$.

PROOF. Follows from Definition 2 and Definition 3 by choosing $b(\langle p_1, p_2\rangle) = \langle p_2, p_1\rangle$. □

LEMMA 2. *Composition is associative:* $(T_1 \cdot T_2) \cdot T_3 \equiv T_1 \cdot (T_2 \cdot T_3)$.

PROOF. Follows from Definition 2, Lemma 1, and Definition 3 by choosing $b\big(\langle\langle p_1, p_2\rangle, p_3\rangle\big) = \langle p_1, \langle p_2, p3\rangle\rangle$. □

3.2 Our Algorithm

We are finally in a position to describe our algorithm for compositional deadlock detection. Algorithm 1 takes a list of SHIM automata as input and returns either a composed SHIM automaton or failure when there is a deadlock. Since the order in which the tasks are composed does affect which automata are built along the way and hence memory requirements and runtime (although, because of Lemma 1, not the final result), the reorder function (called in line 1) orders the automata based on a heuristic that groups tasks with identical channels. Once we compose tasks, we abstract away channels that are not used by other tasks, simplifying the composed automaton at each step.

We then compose tasks one by one. At each step we check if the composed automaton is deadlock free. We remove (line 6 through line 11) any channels that are local to the composed tasks (i.e., not connected to any other tasks). For every channel c, we find all the transitions on that channel (i.e., $\delta(p,c) = \{q\}$) and add ε transitions between states p and q. Then, we use the standard subset construction algorithm [1] to merge such states.

Table 1: Comparison between our compositional analysis (CA) and NuSMV

Program	Lines	Channels	Tasks	Deadlock?	Runtime (s)		Memory (MB)	
					CA	NuSMV	CA	NuSMV
Source Sink	35	2	11	No	0.004	0.004	1.31	6.28
Berkeley	49	3	11	No	0.01	0.01	2.6	5.96
Bitonic Sort	74	56	24	No	1.83	4.01	7.82	53.20
31-tap FIR Filter	218	150	120	No	0.2	21.10	21.06	63.33
Pipeline (1000 pipes)	1003	1000	1000	Yes	397.8	607.8	24.7	813
FFT (50 FFT tasks)	978	100	52	No	34.73	327	16.7	719
Frame Buffer	220	11	12	No	1.81	4.90	5.50	7.5
JPEG Decoder (30 IDCT processes)	2120	180	31	No	51.9	1177	16.06	203.44

We do not abstract away channels connected to other tasks because the other tasks may impose constraints on the communication on these channels that lead to a deadlock. In general, adding another connected task often imposes order. For example, when task 1 and task 2 are composed, communications on b and c may occur in either order. This manifests itself as the scheduling choice at state (4,2) in Figure 4(a). However when task 3 is added, the communication on c occurs first.

The automata we produce along the way often have more behavior than the real system because at each point we have implicitly assumed that the environment will be responsive to whatever the automaton does. However, we never omit behavior, making our technique safe (i.e., we never miss a possible deadlock). Extra behavior generally goes away as we consider more tasks (our abstraction of data means that our automata are always over-approximations, however). For example, when we compose Figure 4(a) with task 3, we get Figure 4(b). We get rid of the impossible case where communication on b appears before c generated in Figure 4(a).

We can only guarantee the absence of deadlock. Since we are ignoring data, we check for all branches in a conditional for deadlock freedom; even if one path fails, at best we can only report the possibility of a deadlock. It may be that the program does not in fact deadlock due to correlations among its conditionals.

4. Experimental Results

We ran our compositional deadlock detector on the programs listed in Table 1 using a 3.2 GHz Pentium 4 machine with 1 GB memory. The *Lines* column lists the number of lines in the program; *Channels* is the number of channels declared in the program; *Tasks* is the number of non-trivial tasks after transforming the call-graph into a tree. *Deadlock?* indicates the outcome.

The *Runtime* columns list the number of seconds taken by both our new compositional tool and our previous work [27], which relies on NuSMV to model-check the automaton. Similarly, the *Memory* columns compare the peak memory consumption of each.

Source-Sink is a simple example centered around a pair of tasks that pass data through a channel and print the results through an output channel. The Berkeley example contains a pair of tasks that communicate packets through a channel using a data-based protocol. After ignoring data, the tasks behave like simple sources and sinks, so it is easy to prove the absence of deadlock. The verification time and memory consumption are trivial for both tools in these examples because they have simple communication patterns.

The Bitonic Sort example uses twenty-four comparison tasks that communicate on fifty-six channels to order eight integers. Although bitonic sort has twenty-four tasks, every channel is owned at most by 2 tasks, which gives our tool an opportunity to abstract away channels when it is not used by the rest of the tasks during composition. This helps to reduce the size of the automaton.

The FIR filter is a parallel 31-tap filter with 120 tasks and 150 channels. Each task consists of a single loop. Figure 5 compares our approach and NuSMV model checker for filters of sizes ranging from 3 to 31. The time taken by our tool grows quartically with the number of taps and exponentially with NuSMV. Figure 5(b) shows the memory consumption.

"Pipeline" is the example from Figure 1. Like the FIR, we tested our tool on a varying number of tasks. Although both tools seem to achieve $O(n^4)$ asymptotic time behavior, ours remains faster and uses less memory. Figure 9 illustrates how our tool performs exponentially on this example if we omit the channel abstraction step.

The FFT example is similar to the pipeline: most of the tasks' SHIM automata consist of a single loop. However, there is a master task that divides and communicates data to its slaves. The slaves and the master run in parallel. The master then waits for the processed data from each of the slaves. Figure 7 shows we perform much better as the size of the FFT increases.

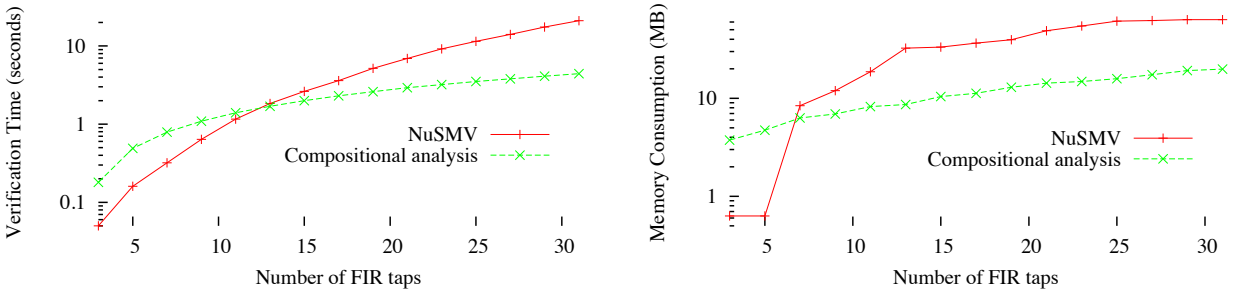
The Framebuffer and JPEG examples are the only programs we tested with conditionals. Framebuffer is a 640×480 video framebuffer driven by a line-drawing task. It has a complicated, nondeterministic communication pattern, but is fairly small and not parametric. Our technique is still superior, but not by a wide margin.

The JPEG decoder is one of the largest applications we have written and is parametric. JPEG decoder has a number of parallel tasks, among which is a producer task that nondeterministically communicates with rest of the IDCT tasks. Figure 8(a) shows our tool exhibiting better asymptotic behavior than NuSMV.

Although our tool worked well on the examples we tried, it has some limitations. Our tool is sensitive to the order in which it composes automata. Although we use a heuristic to order the automata, it hardly guarantees optimality.

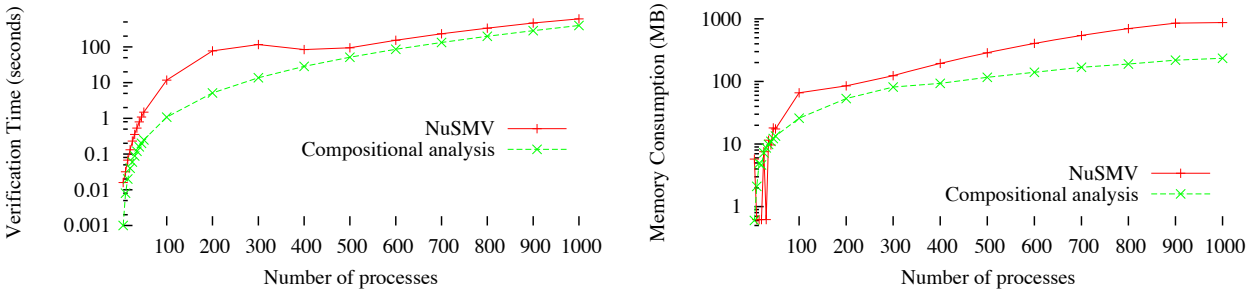
By design, our tool is not a general-purpose model checker; it cannot verify any properties other than the absence of deadlock. Furthermore, it can only provide abstract counter-examples because we remove channels during composition. We have not examined how best to present this information to a user.

Our compositional approach is forced to build the entire system for certain program structures. Consider the call graph shown in Figure 10. The main function forks two parallel tasks, f and g. Both f and g share channels $a_1, \ldots, a_n$. We first compose the children of f and then inline the composed children in f before composing f with g. If f is a pipeline program with a structure similar to the one described in Figure 1, when we compose f's children, we cannot abstract away any channel because g also owns all the channels. This leads to exponential behavior, but we find SHIM programs are not written like this in practice.



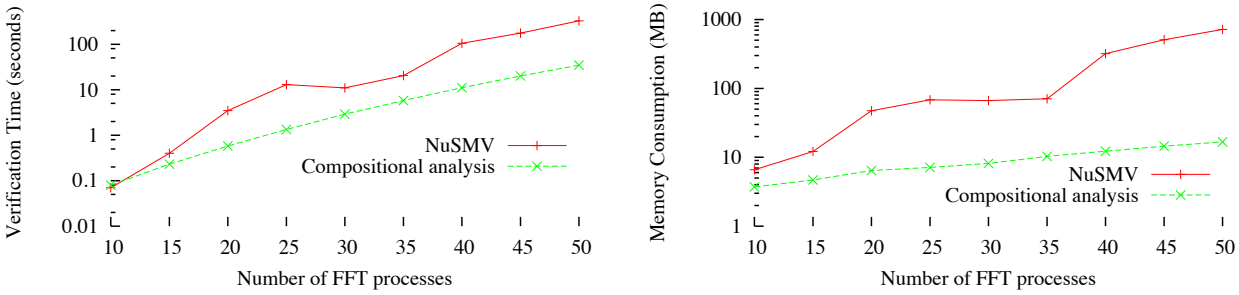
(a) Verification time (b) Memory Consumption

Figure 5: n-tap FIR



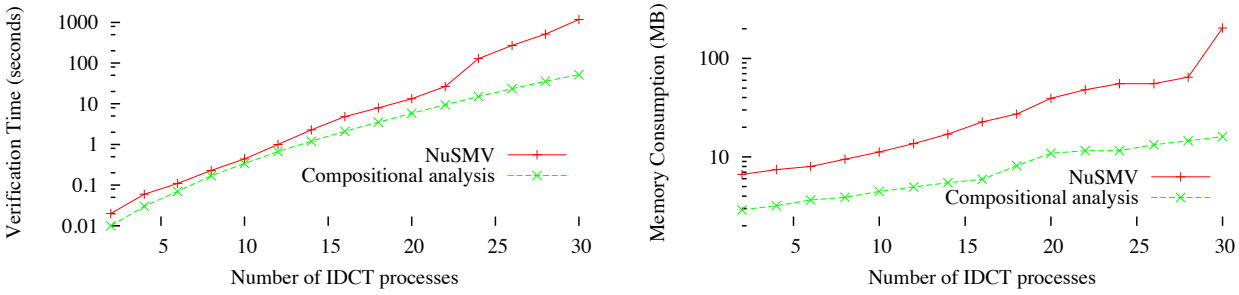
(a) Verification time (b) Memory Consumption

Figure 6: Pipeline



(a) Verification time (b) Memory Consumption

Figure 7: Fast Fourier Transform



(a) Verification time (b) Memory Consumption

Figure 8: JPEG Decoder

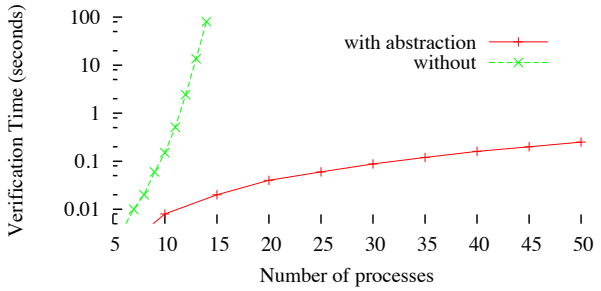


Figure 9: The importance of abstracting internal channels: verification times for the n-task pipeline with vs. without.

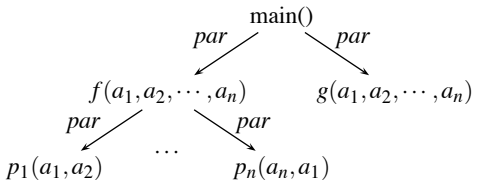


Figure 10: A SHIM program's call graph

5. Related work

Many have tried to detect deadlocks in models with rendezvous communication. For example, Corbett [13] proposes static methods for detecting deadlocks in Ada programs. He uses partial order reduction, symbolic model checking, and inequality necessary conditions to combat state space explosion. However, these techniques do build the entire state space with some optimizations. These may be necessary for Ada, which does not have SHIM's scheduling independence. By contrast, we avoid building the complete state space by abstracting the system along the way. Masticola et al. [21] also propose a way to find deadlocks in Ada programs. Their technique is less precise because they use approximation analysis that runs in polynomial time. Secondly, their method only applies to a subset of Ada programs. By contrast, our technique can be applied to any SHIM program, but can run in exponential time on some.

Compositional verification is a common approach for alleviating the state explosion problem. It decomposes a system into several components, verifies each component separately, and infers the system's correctness. This approach verifies the properties of a component in the absence of the whole system. Two variants of the method have been developed: assume-guarantee [24] and compositional minimization [9].

In the assume-guarantee paradigm, assumptions are first made about a component, then the component's properties are verified under these assumptions. However, it is difficult to automatically generate reasonable assumptions, often requiring human intervention. Although there has been significant work on this [2, 7, 12, 16, 22], Cobleigh et al. [11] report that, on average, assume-guarantee reasoning does not show significant advantage over monolithic verification either in speed or in scalability. Compared to assume-guarantee reasoning, which verifies a system top down with assumptions, our work incrementally verifies the system bottom up. In addition, the assumptions we make along the way are somehow trivial: the environment is assumed to be merely responsive to our tasks' desire to rendezvous.

Instead of assuming an environment, compositional minimization models the environment of a component using another component called the interface and reasons about the whole system's correctness through inference rules. Krimm et al. [23] implemented this algorithm to generate state space from Lotos programs, then extended their work [20] to detect deadlocks in CSP programs with partial order reduction. Our work is similar in that we iteratively compose an interface with a component and later simplify the new interface by removing channels and merging equivalent states. However, they provide little experimental evidence about how their algorithm scales or compares with traditional model checkers.

Zheng et al. [28] apply the compositional minimization paradigm to hardware verification. They propose a failure-preserving interface abstraction for asynchronous design verification. To reduce complexity, they use a fully automated interface refinement approach before composition. Our channel abstraction technique is analogous to their interface refinement, but we apply it to asynchronous software instead of synchronous hardware.

There are many other compositional techniques for formal analysis. Berezin et al. [4] survey several compositional model checking techniques used in practice and discuss their merits. For example, Chaki et al. [5, 6] and Bensalem et al. [3] combine compositional verification with abstraction-refinement methodology. In other words, they iteratively abstract, compose and refine the system's components, once a counter-example is obtained. By contrast, we do not apply any refinement techniques but build the system incrementally to even find a counter-example.

Compared to our previous work on deadlock detection [27] in SHIM, what we present here uses explicit model checking, incremental model building, and on-the-fly abstraction instead of throwing a large model at a state-of-the art symbolic model checker (we used NuSMV [8]). Experimentally, we find the approach we present here is better for all but the smallest examples.

6. Conclusions

We presented a static deadlock detection technique for the SHIM concurrent language. The semantics of SHIM allow us to check for deadlock in programs compositionally without loss of precision.

We expand a SHIM program into a tree of tasks, abstract each as a communicating automaton, then compose the tasks incrementally, abstracting away local channels after each step.

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. However, 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. We illustrated this in previous work [27].

We have compared our compositional technique with our previous work (which used the NuSMV general-purpose model checker) on different examples with varying problem sizes. Experimentally, we find our compositional algorithm is able to detect or prove the absence of deadlock faster: on the order of seconds for large sized examples. We believe this is fast enough to make deadlock checking a regular part of the compilation process.

Tardieu and Edwards [26] added concurrent, deterministic exceptions to the SHIM model, 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.

We also plan to explore counterexample-guided abstraction [6, 10] to consider some data and improve the precision of our analysis.

Currently, our tool only checks for deadlocks, but we believe we may be able to extend our abstraction techniques to check other safety properties. We plan to do this in the future.

7. References

[1] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, second edition, 2006.

[2] Howard Barringer, Dimitra Giannakopoulou, , and Corina S. Păsăreanu. Proof rules for automated compositional verification through learning. In *Proc. 2nd International Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pages 14–21, Helsinki, Finland, 2003. Iowa State U. Tech. Rpt. #03–11.

[3] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for component-based systems and application. In *Proc. Automated Technology for Verification and Analysis (ATVA)*, volume 5311 of *Lecture Notes in Computer Science*, pages 64–79, Berlin, Heidelberg, 2008. Springer.

[4] Sergey Berezin, Sérgio Campos, and Edmund M. Clarke. Compositional reasoning in model checking. In *Compositionality: The Significant Difference (COMPOS)*, volume 1536 of *Lecture Notes in Computer Science*, pages 81–102, Bad Malente, Germany, September 1998.

[5] Sagar Chaki, Edmund Clarke, Joël Ouaknine, and Natasha Sharygina. Automated, compositional and iterative deadlock detection. In *Proc. Formal Methods and Models for Codesign (MEMOCODE)*, San Diego, California, June 2004.

[6] Sagar Chaki, Joël Ouaknine, Karen Yorav, and Edmund Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. *Electronic Notes Theoretical Comp. Sci.*, 89(3):417–432, 2003.

[7] Sagar Chaki and Nishant Sinha. Assume-guarantee reasoning for deadlock. In *Prof. Formal Methods in Computer-Aided Design (FMCAD)*, pages 134–141, San Jose, California, November 2006.

[8] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV version 2: An OpenSource tool for symbolic model checking. In *Proc. Computer-Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364, Copenhagen, Denmark, July 2002.

[9] E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proc. Logic in Computer Science (LICS)*, pages 353–362, Piscataway, NJ, USA, 1989. IEEE Press.

[10] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proc. Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169, Chicago, Illinois, July 2000.

[11] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Trans. Softw. Eng. Methodol.*, 17(2):1–52, 2008.

[12] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2619 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, 2003.

[13] James C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Software Engineering*, 22(3):161–180, March 1996.

[14] Stephen A. Edwards and Olivier Tardieu. SHIM: A deterministic model for heterogeneous embedded systems. In *Proc. Embedded Software (Emsoft)*, pages 37–44, Jersey City, New Jersey, September 2005.

[15] Stephen A. Edwards and Jia Zeng. Static elaboration of recursion for concurrent software. In *Proc. Partial Evaluation and Program Manipulation (PEPM)*, pages 71–80, San Francisco, California, January 2008.

[16] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Shaz Qadeer. Thread-modular abstraction refinement. In *Proc. International Conference on Computer-Aided Verification (CAV)*, volume 2725 of *Lecture Notes in Computer Science*, pages 262–274. Springer, 2003.

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

[18] Gerard J. Holzmann. The model checker SPIN. *IEEE Trans. Software Engineering*, 23(5):279–294, May 1997.

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

[20] Jean-Pierre Krimm and Laurent Mounier. Compositional state space generation with partial order reductions for asynchronous communicating systems. In *Proc. of Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *Lecture Notes in Computer Science*, pages 266–282. Springer, 2000.

[21] Stephen P. Masticola and Barbara G. Ryder. A model of Ada programs for static deadlock detection in polynomial times. In *Proc. Parallel and Distributed Debugging (PADD)*, pages 97–107, New York, NY, USA, May 1991. ACM.

[22] Wonhong Nam, P. Madhusudan, and Rajeev Alur. Automatic symbolic compositional verification by learning assumptions. *Journal of Formal Methods in System Design*, 32(3):207–234, June 2008.

[23] Jean pierre Krimm and Laurent Mounier. Compositional state space generation from Lotos programs. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1217 of *Lecture Notes in Computer Science*, pages 239–258, Enschende, The Netherlands, April 1997. Springer.

[24] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In *Proc. NATO Advanced Study Institute on Logics and Models of Concurrent Systems*, pages 123–144, La Colle-sur-Loup, France, 1985. Springer.

[25] Olivier Tardieu and Stephen A. Edwards. R-SHIM: Deterministic concurrency with recursion and shared variables. In *Proc. International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, page 202, Napa, California, July 2006.

[26] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proc. Embedded Software (Emsoft)*, pages 142–151, Seoul, Korea, October 2006.

[27] Nalini Vasudevan and Stephen A. Edwards. Static deadlock detection for the SHIM concurrent language. In *Proc. Formal Methods and Models for Codesign (MEMOCODE)*, pages 49–58, Anaheim, California, June 2008.

[28] Hao Zheng, Jared Ahrens, and Tian Xia. A compositional method with failure-preserving abstractions for asyn chronous design verification. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1343–1347, July 2008.