Case Studies in Symbolic Model Checking

Ganesh Gopalakrishnan, Dilip Khandekar, Ravi Kuramkote and Ratan Nalumasu

UUCS-94-009

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

March 15, 1994

Abstract

Formal verification of hardware and software systems has long been recognized as an essential step in the development process of a system. It is of importance especially in concurrent systems that are more difficult to debug than sequential systems. Tools that are powerful enough to verify real-life systems have become available recently. Model checking tools have become quite popular because of their ability to carry out proofs with minimal human intervention. In this paper we report our experience with SMV, a symbolic model verifier on practical problems of significant sizes. We present verification of a software system, a *distributed shared memory* protocol, and a hardware system, the *crossbar arbiter*. We discuss modeling of these systems in SMV and their verification using temporal logic CTL queries. We also describe the problems encountered in tackling these examples and suggest possible solutions.

Case Studies in Symbolic Model Checking

Ganesh Gopalakrishnan, Dilip Khandekar, Ravi Kuramkote and Ratan Nalumasu

Department of Computer Science
University of Utah
Salt Lake City, UT 84112
email: {ganesh,khands,kuramkot,ratan}@cs.utah.edu

1 Introduction

The need to formally verify hardware and software systems before they are deployed the real world has been recognized for several decades now. This is especially true of concurrent systems that are even more difficult to debug than sequential systems. For example, many of the protocols that get employed in real-life systems often look deceptively simple at first glance, and yet often contain hidden errors. In many cases, these errors cannot be revealed through simulation alone. Tools that are powerful enough to verify real-life systems of significant sizes have, however, only recently become available, thanks to developments such as efficient Boolean reasoning methods. As a result, there is a real opportunity amongst practitioners of formal methods to apply these tools to real-life examples and to teach tomorrow's designers—today's students—the use of these tools for solving practically significant problems. This paper is about our efforts in this regard.

Among formal verification tools that can verify concurrent systems, *model checking* tools have become quite popular. There are several reasons for this. First, their ability to carry out proofs with relatively very little human intervention makes it possible for designers to automatically carry out many of the proofs, thus freeing them up for more creative tasks. Second, most concurrent hardware and software systems are one-ofs, and hence it is very difficult to recoup human effort put into one project for use in another.

In this paper, we report on our experience in using SMV [1], a symbolic model checker, in a graduate class on Program Verification, on practical problems of significant sizes. SMV has previously been extensively used by several researchers [1, 2] in verifying non-trivial hardware and software systems. In this paper we present the following case studies: (i) verification of a distributed shared memory protocol [3] that is widely known; (ii) verification of a new crossbar arbiter that the first author's group has developed. We also discuss the problems encountered while tackling these examples using SMV, and suggest possible solutions. We divide this discussion into three sections: (i) aspects related to the expressive power of the CTL formalism; (ii) aspects related to state explosion; and (iii) aspects related to the particular implementation of SMV. The main contribution of this paper is that it provides detailed verification case studies on problems of great interest to designers of distributed protocols as well as designers of asynchronous circuits that might be used in realizing these protocols.

The remainder of the paper is organized as follows. In Section 2 we discuss the verification of the centralized version of the distributed shared memory (DSM) protocol [3]. Section 3 presents the verification of a distributed version of the same protocol. This widely used protocol implements a demand-paged virtual memory system across a collection of computing nodes (Figure 1). Both the centralized and the distributed DSM protocols described in this paper are widely referred to, and form the basis of several new protocols of a similar nature, for example [4, 5]. Descriptions of these protocols were taken verbatim, in the form of pseudo-code, from [3], and encoded in SMV.

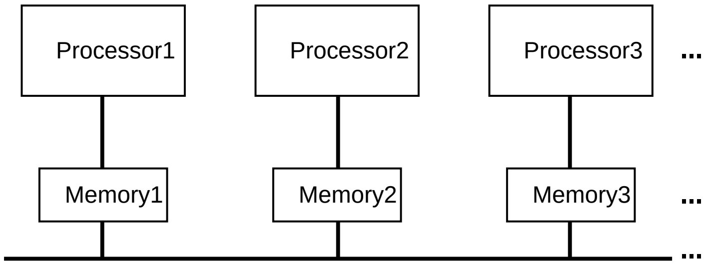

Figure 1: Abstraction of distributed shared memory provided by computing nodes

Admittedly this is a less than perfect process, because pseudo-code descriptions can be ambiguous and therefore can be mis-interpreted. However, pseudo-code descriptions form the link between the designers' thoughts and the final implementation; hence, it is of considerable practical importance to scrutinize the purported algorithm while it is at a pseudo-code level, for this offers the opportunity to detect errors sufficiently early, and helps in resolving ambiguities, if any. Also, we do not have access to the final implementation of the algorithms of [3]; therefore, remarks regarding correctness made in this paper apply only to the pseudo-code.

In Section 4, we present the verification of a symmetric crossbar arbiter—an asynchronous arbiter described in [6]. This example provides useful insight into asynchronous circuit verification using SMV. We provide concluding remarks in Section 5.

2 Shared Virtual Memory: Centralized Manager

Consider a collection of computer nodes, each supporting one or more processes. Suppose each of these processes wants to view the aggregate of the memory available on all the computer nodes as one homogeneous shared memory. This view can be supported by treating the memory available at each node as a *cache*, and providing mechanisms for demand-paging across the computer nodes. Any scheme of this nature has to maintain the *coherence* of the individual memory units with respect to the logical abstraction of a shared memory.

In the centralized algorithm, one of the nodes is selected as the *manager* node. The manager node has an *information-table* with each entry in it corresponding to a page. More specifically, an information-table entry for a page consists of the *owner* of the page (the node that last wrote into the page), the *copyset* of the page (*i.e.*, which nodes have the page available for use in the *read* mode), and a *lock* (semaphore) to give the manager exclusive access to the information-table entry. In addition, every node (including the manager) maintains a *page table* that has, for each entry corresponding to a page, information on the *page access mode* and a *lock* to provide exclusive access to a page table entry. Every node has a *read-fault handler* and a *write-fault handler* to handle *local* read/write-faults. Each node also has a *read server*, a *write server*, and an *invalidate server*. These servers handle remote requests as elaborated in Section 2.2.

2.1 Overview of the Results

Our main result concerning the verification of the centralized protocol is summarized before we go into the details. The pseudo-code appearing in [3] is scanty in detail about certain boundary cases. If special care is not taken in modeling these boundary cases, deadlocks will result. We were satisfied that this observation was confirmed by SMV, as detailed in Section 2. However, to our pleasant surprise, we could also spot another deadlock that was more subtle and was previously unknown to us. This experience clearly demonstrates SMV's ability to detect errors in pseudo-code statements of non-trivial algorithms.

2.2 Details of the Centralized DSM Protocol

We now present the algorithms used by the various fault handlers and servers.

2.2.1 Read-fault Handler

Upon encountering a read-fault, the read-fault handler (RFH)

RFH1: locks the page-table entry corresponding to the faulted page;
RFH2: asks the manager for the page in read mode;
RFH3: upon receipt of the page, sends confirmation to the manager;
RFH4: sets the local page-table entry access mode to *read*;
RFH5: unlocks the page-table entry.

2.2.2 Write-fault Handler

Upon encountering a write-fault, the write-fault handler (WFH)

WFH1: locks the page-table entry corresponding to the faulted page;
WFH2: asks the manager for the page in write mode;
WFH3: upon receipt of the page, sends confirmation to the manager;
WFH4: sets the local page-table entry access mode to *write*;
WFH5: unlocks the page-table entry.

2.2.3 Read Server Running on the Manager Node

Upon encountering an external read request, the manager read server (MRS)

MRS1: locks the information-table entry for the page in question;
MRS2: includes the requester in the copyset;
MRS3: asks the node that owns the page being requested to send a copy directly to the requester;
MRS4: waits for confirmation from the requester;
MRS5: unlocks the information-table entry.

2.2.4 Write Server Running on the Manager Node

Upon encountering an external read request, the manager write server (MWS)

MWS1: locks the information-table entry for the page in question;
MWS2: for each node in the copyset of the page in question, invokes its invalidate server (only the requester node can hold a page in the write mode - all the copies of this page must be invalidated);
MWS3: assigns the copyset to the *empty set*;
MWS4: asks the owner node to send a copy of the page directly to the requester;
MWS5: waits for confirmation from the requester;
MWS6: marks the requester as the new owner in the information-table entry for this page;
MWS7: unlocks the information-table entry.

2.2.5 Read Server at the Owner Node

Upon encountering a read request, the owner read server (ORS)

ORS1: locks the page-table entry;

ORS2: sets the access mode of the page to *read*, in the page-table entry (so if the owner had *write* access before, it relinquishes this privilege);

ORS3: sends copy of the page to the requester;

ORS4: unlocks the page-table entry.

2.2.6 Write Server at the Owner Node

Upon encountering a write request, the owner write server (OWS)

OWS1: locks the page-table entry;

OWS2: sets the access mode to *invalid*;

OWS3: sets a copy of the page to the requester;

OWS4: unlocks the page-table entry.

2.2.7 Invalidate Server (anywhere)

An *invalidate server* at any node merely sets the access mode of the page to *invalid*. It does not lock the page-table entry, as setting the invalidation bit is an atomic step.

2.3 Discussions

State-explosion is a constant threat while using a symbolic model-checker for significantly sized problems. It is almost always necessary to take advantage of the symmetries in the problem, thereby minimizing the number of different situations modeled. For example, without any loss of generality, we can model a version of the protocol using only one page. As far as the number of nodes go, a minimum of three was felt necessary (to model the owner node, the manager node, and the requesting node as three separate entities). However, state explosion prevented us from doing this, and we could model only a maximum of two nodes, despite considerable efforts put into variable orderings. We feel that the practical applicability of SMV can be greatly enhanced if the system offers users with sufficient insight into the problem being modeled, what might be causing the state explosion, and also help him/her determine a suitable variable ordering. Work done in [1] and [7] relating to the complexity characteristics of BDDs should help in this regard.

2.3.1 The First Deadlock

The first deadlock situation detected by SMV is the following. Consider two nodes, 0 and 1, with node 1 owning the only page in the system in the read mode. (This can happen immediately after the following sequence: node 1 wrote into the page; node 0 had a read-fault into the page; therefore node 1 reverted back to the read mode, but still remaining the owner.) Now suppose node 1 has a write-fault for the page. Hence, the fault handler locks the page table entry at node 1 for the page (state WFH1). A request then goes to the manager write server for the page. The manager locks the information-table entry (MWS1). Since node 1 is the owner, the manager requests the owner to send page to itself (!). Node 1's read server tries to lock its page table entry (ORS1) which node

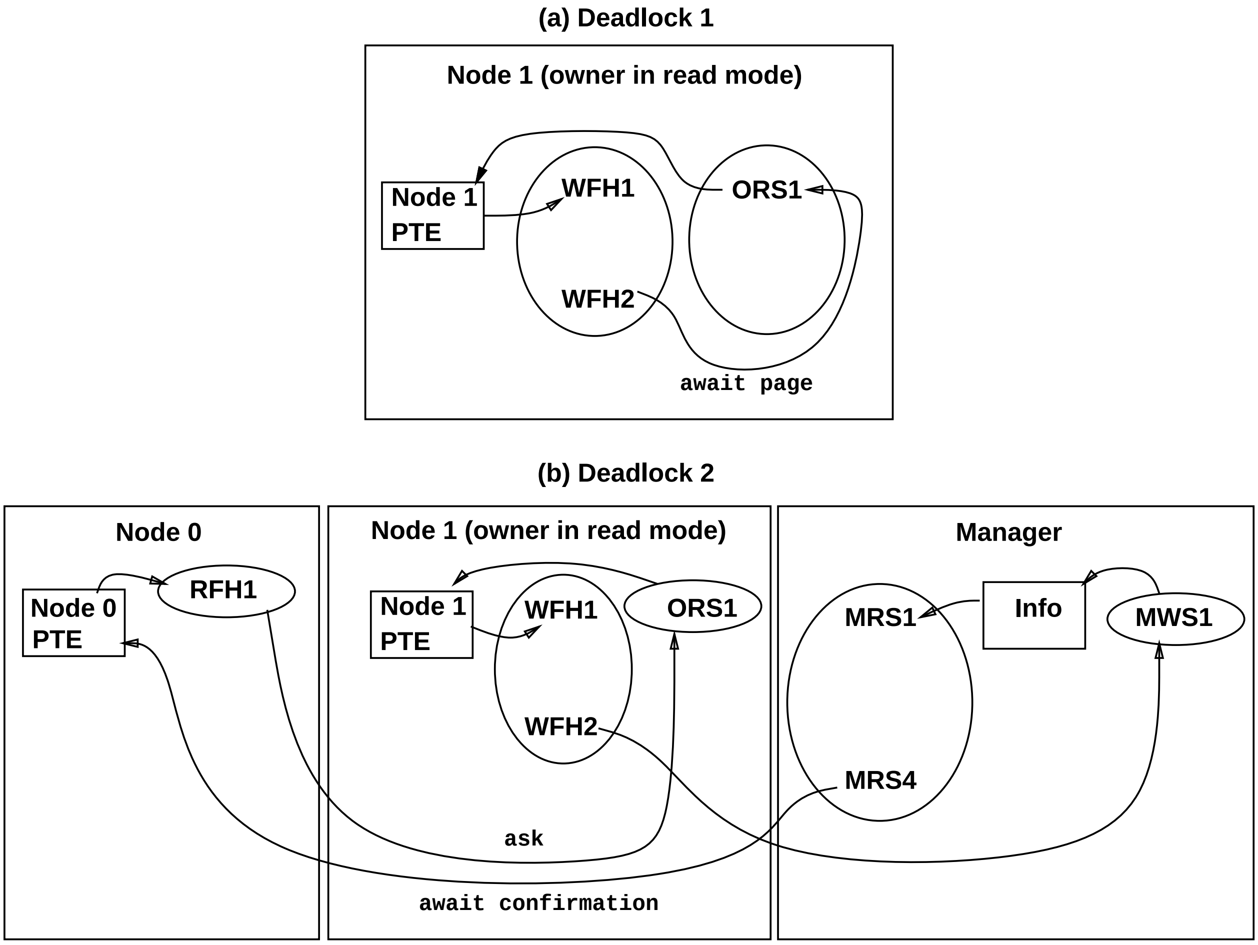

Figure 2: Resource Dependency Graphs Upon Deadlocks: Centralized Algorithm

1 has already locked at WFH1! This is a direct deadlock (see Figure 2(a) which shows the cycle in the resource dependency graph.) The problem arose because the pseudo-code did not first check whether node 1 was the owner—if it did, it could have avoided going to the centralized manager! Pseudo-code routines typically leave out detail such as this.

This deadlock was known to us even before we embarked on verification. As this example shows, considerable caution has to be exercised in translating pseudo-code into actual code. Therefore, it was satisfying that SMV could detect this deadlock.

2.3.2 The Second Deadlock

The second deadlock is more subtle in nature, and its existence was previously unknown to us. The scenario is as follows. Node 1 owns the page in the read mode and has a write-fault for the page. The write-fault handler of node 1 locks the page table entry for the page (WFH1).

Meanwhile node 0 has a read-fault and its request goes to the manager. The manager locks its information-table entry (MRS1) and requests the owner's read server (*i.e.*, node 1's read server) to send the page directly to node 0 which is the requester. Node 1's read server now tries to lock the page table entry (ORS1), but hangs as node 1 has already locked this entry at WFH1.

Node 1's fault handler is meanwhile not blocked. It sends a request to the manager to have the page's access mode converted from *read* to *write*. The manager tries to lock its information-table entry (MWS1) which has already been locked at MRS1. At this stage there is a deadlock! (See Figure 2(b) which shows the cycle in the resource dependency graph.)

The solution is to prevent the race between node 1's write-fault handler's actions and node 0's read-fault handler's actions. Several solutions are possible and we have not pursued any particular solution, as that was not the purpose of our exercise.

3 Shared Virtual Memory: Distributed Manager

The verification of the dynamic distributed manager (DDM) algorithm proposed in [3] is detailed in this section. In the DDM, there is no centralized manager. Every node has sufficient information to locate the required page. The nodes use message passing mechanism to communicate with each other. A page has three modes associated with it: *invalid*, *read*, and *write*. A processor not having a local copy of the page has it mapped *invalid*. A page can be shared by several processors in *read* mode, while a page can reside in *write* mode only at one node. All processors sharing a page must invalidate it when one node wants to write into the page.

Each node maintains a probable owner for each page it has a copy of. The probable owner is that processor which this node "thinks" to be the actual owner of the page. It may be the case that the probable owner may not be the actual owner; in case it isn't, the probable owner will have its own probable owner for that page. The algorithm guarantees that a message forwarded along such a "probable owner chain" will finally reach the actual owner of that page, thereby ensuring that the page will be located.

3.1 Overview of the Results

In our SMV descriptions, we could model two processors. Modeling three processes would have covered most situations (we think three processes are sufficient, but do not have a proof for this). For certain queries, we could model the behavior of three processes, however only after manually eliminating many of the capabilities of the processes. This approach of specializing the process descriptions in response to the queries being handled is error-prone. We eagerly await more powerful versions of SMV that have the ability to handle much more state and/or offer insight into where state explosion is happening. Modulo these limitations, however, we could establish a large number of interesting properties with success. One fairly obvious deadlock (that can be blamed on the abstractness of the pseudo-code) was also detected.

3.2 Details of the Distributed DSM Protocol

Details of the DDM protocol are now provided for each of the operations supported. Upon read hit, the state of the page does not change. Upon write hit, the access type of the page is changed to *write*.

3.2.1 Read-fault Handler

Upon encountering a read-fault, the node

RFH1: asks the probable owner of the page to give *read* access to the page;

RFH2: when the page arrives, it sets the probable owner for that page to itself, and changes the page access type to *read*.

3.2.2 Write-fault Handler

Upon encountering a write-fault, the node

WFH1: asks the probable owner to give write access to the page;
WFH2: sends an invalidation message for the page to the copyset of the page;
WFH3: sets its probable owner field to point to itself;
WFH4: when the page arrives, it sets the access mode of the page to *write*.

3.2.3 Read Server at the Owner Node

Upon encountering a read request, the owner read server (ORS)

ORS1: if it is the owner of the page, **then**

ORS1a: adds self to the copyset of the page;
ORS1b: changes the access-type of the page to *read*;
ORS1c: sends the page and its copyset to the requester;
ORS1d: in the local page table, records that the probable owner of the page is the requestor.

else

ORS2a: forwards the request to the probable owner of the page;
ORS2b: sets the probable owner to be the requestor.

3.2.4 Write Server at the Owner Node

Upon encountering a write request, the owner write server (OWS)

OWS1: If it is the owner, **then**

OWS1a: sets the access mode of the page to *invalid*;
OWS1b: sends the page and the copyset of the page to the requestor;
OWS1c: in the local page-table, sets the probable owner of the page to the requestor.

else

OWS2a: forwards the request to the probable owner;
OWS2b: sets the probable owner to the requester.

3.2.5 Invalidate Server (anywhere)

For each invalidation request,

IS1: sets the access mode to invalid;
IS2: sets the probable owner field to the requestor.

The processes in SMV are modeled as follows. Each node in the DSM system is a process at the top level in SMV. This process also acts like a read/write fault handler. Each process has as its sub-process the *read/write* server and *invalidate* server.

The communication between the processors was modeled in SMV using globally shared variables as well as more modular constructs that simulate message passing. Modeling all the process interactions without using shared variables would have resulted in SMV code that more closely resembles the pseudo-code. This direction was abandoned as it resulted in state explosion.

3.3 Discussions

Using SMV, the following properties were established of the specification of the DDM algorithm:

- "Suppose processor P3 does not have the page, wants read access to the page, and thinks that probable owner is P1. Suppose processor P1 is not the owner and thinks that P2 is the probable owner; suppose P2 is the actual owner. Then the the message from P3 to P1 will be forwarded to P2, which will then grant *read* access to P3."
- "If the processor wants to *write* into the page which it does not currently have, it will eventually have access to the page in the *write* mode." Similarly the *read* access was also successfully validated.
- "A page can be shared by two processors in the *read* mode."
- "If a page is in the *write* mode at some node, it cannot reside in either the *read* or the *write* modes in any other processor."
- "A page has to reside somewhere; it cannot be *invalid* in all the processors."
- "If a page is being shared by the two processors in the *read* mode and one of them wants to *write* into the page, then it will get the page in the *write* mode, while the other node has to invalidate its copy of the page."

The following error (attributable to the abstract nature of the pseudo-code) was detected. Suppose a node N that owns a page has it in *read* mode and wants to *write* into it. The resulting write fault will cause the write fault handler on node N to lock the page table entry and send a message to the probable owner. The message will traverse the probable owner chain (whose length can be zero or more) and eventually arrive back at node N. The write server on node N tries to lock the page table entry, but will hang as it has already been locked by the write fault handler. Again the error is due to the pseudo-code not being very specific about boundary conditions.

It is, however, very easy to ignore these boundary conditions and proceed with the coding of the algorithm, thereby making such errors even more hard to detect. The use of model checking tools such as SMV early in the design process can prevent this from happening.

4 Hardware Verification: Verification of a Crossbar Arbiter

A symmetric crossbar arbiter [8] arbitrates requests for connections to be made on an $N \times N$ crossbar switch. Assume that at any particular instant of time, a subset the N^2 switches can be requested to be closed. In response to any such request, the arbiter must grant the maximum possible number of requests (at most N) that do not conflict on any row or column (*i.e.*, that do not share any row- or column-wire of the crossbar). In [6], we have developed a family of arbiters that meet the above specifications. As opposed to the circuits used by [8], our circuits are all asynchronous in nature. Furthermore, they are based on a new asynchronous component developed by the first author called the *lockable C-element* [9]. We illustrate our verification efforts on one of these circuits given in Figure 3, called the wavefront arbiter.

The operation of the wavefront arbiter is as follows. Each element shown in the figure is a *lockable C-element*—or, LockC for short. A LockC behaves similar to a Müller C-element, except it has an extra input called *lock* and an extra output called *lack* (not shown in the diagram to avoid clutter—we also avoid showing some of the logic associated with each LockC, again to avoid clutter).

When no external requests are present, all the *lock* inputs are kept deasserted. As a result, any LockC can fire whenever it is enabled. Under these circumstances, the circuit shown in Figure 3,

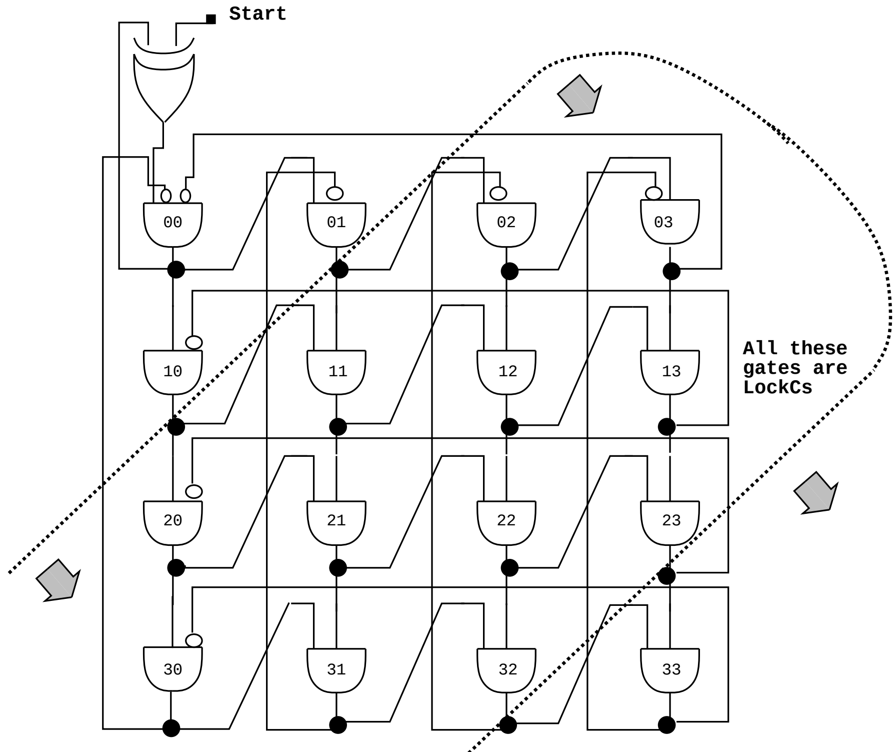

Figure 3: A Symmetric Crossbar Arbiter Design

propagates *diagonal wavefronts* that propagate from the top-left corner towards the bottom-right corner. More precisely the array will always (except during a brief moment) have two diagonal wavefronts flowing through it from the top-left corner towards the bottom-right corner. Furthermore, these two diagonals will always be the closest two such diagonals that do not have any row-wire or column-wire in common. (This spacing is achieved by the wrapped torus connection to the resetting input of the LockC elements.)

One snapshot of these two diagonals is shown in Figure 3. We connect the two diagonals with a curved line to signify that these two diagonals are "connected": when one diagonal moves forwards, so does the other. For example, the position of the diagonals shown in the figure is 01, 10, 32, and 23; the next position of the diagonals will be 02, 11, 20, and 33; after that the diagonals will be at 03, 12, 21, 30 (at this time there will be only one diagonal); the next diagonal will then manifest at 00 and 31, 22, 13.

Suppose station i, j wants to close the crossbar switch i, j. It requests permission by applying a *lock* input to LockC i, j. If a wavefront is passing through LockC i, j or is just about to do so, the effect of the *lock* input is non-deterministic: the wavefront may either be pinned down at location i, j, or it may be allowed to slip through. (Whatever be the outcome, the decision is crisp, without metastability.) If the wavefront slips through, then the request at i, j has to be held till the wavefront comes to i, j once again. In this case, the wavefront is sure to be pinned down because the *lock* was applied much earlier with respect to this arrival of the wavefront.

When a wavefront is finally pinned down at location i, j, a *lack* output is produced (with the help of a few logic gates which are not shown). The *lack* signal can be taken as permission to close switch i, j. After the use of switch i, j is over, *lock* is deasserted, whereupon *lack* is deasserted, and the wavefront that is pinned down is allowed to move forwards once again. Because of the asynchronous signaling protocols employed, even after a wavefront is pinned down at one location (for example at 01), it can still move ahead at other fronts. In other words, the wavefront can warp till it is about to encroach into "forbidden regions". So, for example, the portion of the wavefront at location 10 can move to 20 and 30 even after being pinned down at 01.

4.1 Overview of the Results

We specified the wavefront arbiter in SMV and verified the following properties:

- **Safety Conditions:** "A column cannot be assigned to two rows simultaneously, or a row can not be assigned two columns simultaneously." For example,

  ```
  AG (c[0][0].trapped -> !c[1][0].trapped);
  AG (c[0][0].trapped -> !c[0][1].trapped);
  ```

- **Deadlocks:** "No LockC loses the ability to access a column (and similarly, a row)." For example,

  ```
  AG EF c[0][1].trapped;
  ```

- **Progress:** "If lock is asserted, the wave is guaranteed to be eventually trapped." For example,

  ```
  AG (c[0][0].ll.lock -> AF c[0][0].trapped);
  ```

Due to state explosion, the size of the largest array that could be verified was 3x3. An informal ("paper and pencil") inductive proof of correctness for arbitrary sizes is easy to provide. Carrying out induction in the framework of SMV (through the use of a suitable network invariant [1]) is presently being looked into.

4.2 Details of the Crossbar Arbiter and its Verification

The wavefront arbiter was specified at the structural level. Each cell contains an XOR gate and a LockC gate. Modules such as the LockC can be elegantly specified in SMV owing to its capability to describe concurrent processes. For illustration, the description of the LockC is given below.

```
MODULE LockC(row-in, column-in, out) -- A LockC is specified the way it is used
                                     -- in the wavefront arbiter
   VAR
      ll         : process lock-lack(a,b,c)
   ASSIGN
      next(out)  := case
                       ll.lack          : out;    -- after lack, freeze out
                       row-in=column-in : row-in; -- o/w, when enabled, fire
                       1                : out;    -- when not enabled, hold
                    esac;
    DEFINE -- defines when token is trapped
       trapped := lack & (row-in = column-in) & (row-in = !out);
    FAIR
       running
-- END Lockable-C-Element

MODULE lock-lack(a,b,c)    -- Modify lock, and lack asynchronously
   VAR
     lock : boolean;
     lack : boolean;
   ASSIGN         -- Locally generate to simulate the PE(i) requests.
     init(lock) := 0;
     init(lack) := 0;
     next(lack) := lock; -- arbitrary delay between lock and lack is achieved
                         -- due to the use of the variable 'running'
     next(lock) := case
                     (a=b) & (a=!c) & lack : 0; -- unlock after token trapped
                     1                     : 1; -- o/w, try trapping token
                   esac;
   FAIR
      running
-- END MODULE
```

The wavefront arbiter was described by replicating the XOR and LockC gates using the FOR construct of SMV.

4.3 Discussions

The state space of the wavefront arbiter grows exponentially with the array size, as the cells of the arbiter can be in all possible combinations of their states. This was observed in our inability to verify arbiters of sizes higher than 3x3. Application of induction techniques to the arbiter circuit ended to be not so straightforward as the examples dealt with in [1]. The main idea used in [1] is to identify a network invariant and then to design a *generic* module that can simulate an arbitrary number of the modules in the original design. This approach is straightforward to apply when the design consists of entities such as a single shared global bus on which an arbitrary number of components can be replicated. In that case, a cut-point on the bus can be identified and a generic module representing an arbitrary number of modules connected to a shared bus can be plugged in at the cut-point. The number of inputs and outputs of the generic module do not depend on the size of the arbitrary-sized array being modeled by it.

Unfortunately, in case of arbiter, each of the cells takes one input from its top neighbor and another input from its left neighbor. A generic module that represents "the remainder of the wavefront arbiter array" does not have a fixed number of inputs. Induction can still be carried out in the two dimensions separately. However, in that case, the behavior of a row-slice or a column-size is not quite as intuitive.

The wavefront arbiter presented in Figure 3 is inefficient in one respect. When a wavefront is trapped at location i, j, location $i+1, j+1$ cannot make any connections, even though it does not share a row- or column with i, j. This disadvantage is overcome by another arbiter designed by us,

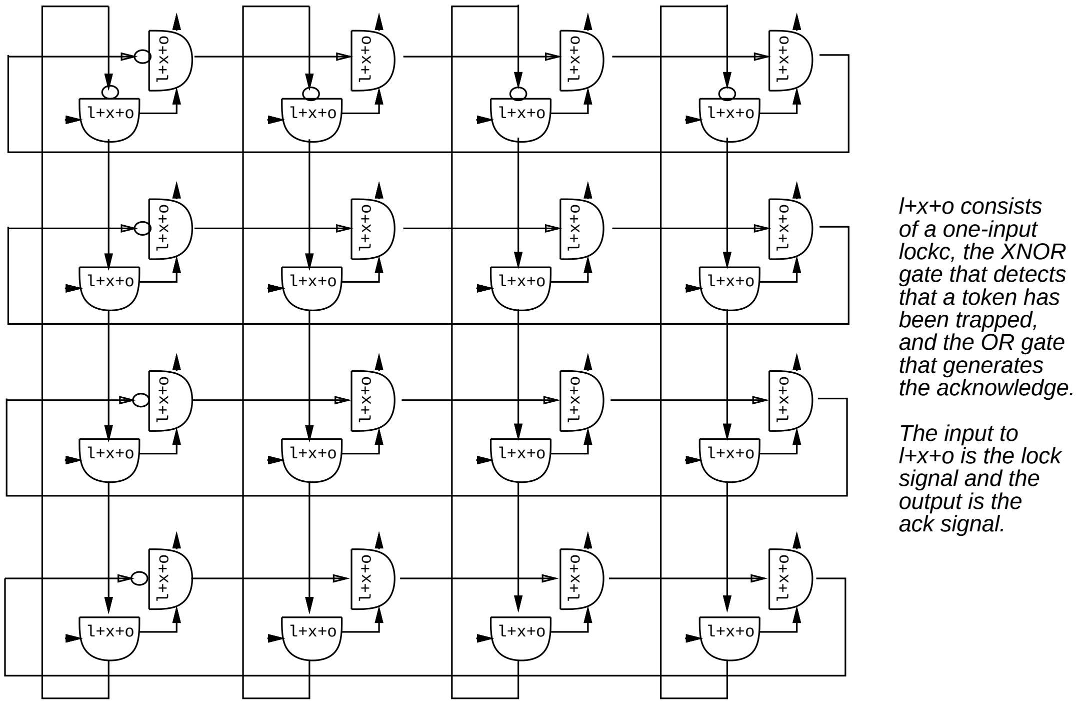

Figure 4: Crisscrossing One Dimensional Arbiters

called the "crisscrossing one-dimensional arbiters" (Figure 4). This circuit has also been verified using SMV.

5 Concluding Remarks

In the long run, SMV must be interfaced to design systems. As a preliminary step in that direction, we have developed a graphical interface to SMV using which Petri-nets can be drawn and automatically compile into SMV descriptions. We are sure that this tool called `Petriland` (which was developed by Jim St.Germain, a student of the Program Verification class taught by the first author) will greatly simplify the encoding of designs.

The present implementation of SMV is not very much oriented towards specifying systems with shared writable variables (that can be written from multiple places). Due to its emphasis on compositional specifications, the SMV system requires the programmer to explicitly indicate every shared writable variable update, even if the update is merely to hold the same value across one time-step. This makes the modeling of many protocols notationally very tedious. A tool such as Petriland can again help here because it generates SMV code that uses only TRANS assertions to directly specify the state transitions underlying the Petri-net being modeled.

The circuits used to realize our crossbar arbiters require certain one-sided timing constraints to be obeyed in their implementation. Although these timing constraints can be encoded in SMV, we believe that other formalisms (*e.g.*, [10, 11, 12]) may be more suitable for this level of verification.

In conclusion, we are pleased with how SMV has fared in our hardware and software verification experiments. Coding styles that will prevent state explosion from occurring must be developed and discussed. The SMV system must also provide insight to the user on the source of state explosion and provide better insight into its operation.

The examples discussed in this paper, a few other examples (including the description and validation of the Cache Coherence protocol obeyed by the Alpha Demonstration Unit [13] written by Yarden Livnat), and the code of Petriland are available upon request from ganesh@cs.utah.edu.

Acknowledgements: Many thanks to all the students who participated in CS 611, "Program Verification" whose efforts made these experiments possible, and to NSF who supported this work in part through award MIP-9215878.

References

[1] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, 1993.

[2] 1992. *Tutorial #9 on Formal Verification offered during the 1992 DAC by Edmund Clarke et. al.*

[3] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[4] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM transactions on Computer Systems*, August 1994. To appear.

[6] Ganesh Gopalakrishnan. Some unusual micropipeline circuits. Technical Report UUCS-93-015, University of Utah, Department of Computer Science, 1993.

[7] Alan Hu and David Dill. Reducing bdd size by exploiting functional dependencies. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 266–271, 1993.

[8] Yuval Tamir and Hsin-Chou Chi. Symmetric crossbar arbitration. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):13–27, January 1993.

[9] Armin Liebchen and Ganesh Gopalakrishnan. Dynamic reordering of high latency transactions in time-warp simulation using a modified micropipeline. In *International Conference on Computer Design (ICCD)*, pages 336–340, 1992.

[10] Jerry Burch. *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems.* PhD thesis, Carnegie-Mellon University, August 1992. *Technical Report CMU-CS-92-179.*

[11] David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* MIT Press, 1989. *An ACM Distinguished Dissertation.*

[12] Ganesh Gopalakrishnan, Nick Michell, Erik Brunvand, and Steven M. Nowick. A correctness criterion for asynchronous circuit verification and optimization. *IEEE Transactions on Computer-Aided Design*, 1992. *Accepted for Publication.*

[13] Charles P. Thacker, David G. Conroy, and Lawrence C. Stewart. The alpha demonstration unit: A high-performance multiprocessor. *Communications of the ACM*, 36(2):55–66, February 1993.