SPIN vs. VIS: A Case Study on the Formal Verification of the ATMR Protocol

Hong Peng, Sofiène Tahar and Ferhat Khendek
Dept. of Electrical & Computer Engineering, Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3G 1M8 Canada
E-mail: {pengh,tahar,khendek}@ece.concordia.ca

Abstract

Nowadays, there exists a wide variety of verification tools. Some, like SPIN, are designed and mainly used for the verification of interleaving software systems, such as communications protocols. Others, like VIS, are designed and used for synchronous hardware systems verification. In this paper, we compare and contrast SPIN and VIS. In particular, we devote a special attention to the efficiency of these tools for the verification of communications protocols that can be implemented either in software or hardware. As a basis of our comparison, we formally describe and verify the ATMR (Asynchronous Transfer Mode Ring) medium access protocol using SPIN and its hardware implementation using VIS. We believe that this study is of particular interest as more and more protocols, like the ATM protocol stack, are implemented in hardware to match the high speed requirements. However, this is not a formal comparison of SPIN and VIS.

1 Introduction

For the last two decades, verification techniques have been applied successfully in software and hardware engineering. Various techniques have been proposed in the literature [4]. They range from pure simulation to model checking. The widely used simulation techniques cannot cover all design errors, especially for large systems. Like testing techniques, they are used to detect errors, but not to prove the correctness of the design. During the past decade, model checking techniques have established themselves as significant means for design validation. A given design is validated against specific and general properties. Two different kinds of model checking tools have arisen: tools for the formal verification of software protocols and interleaving systems, like SPIN [5], and tools for the formal verification of synchronous digital hardware, like VIS [1].

The SPIN software verification tool, developed by J. Holzmann at Bell Labs, is based on an interleaving model of concurrency, in which unlike hardware, only one component of the system state is allowed to change at a time. SPIN checks for the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flag incompleteness, race conditions, and unwarranted assumptions. It also validates properties specified as LTL formulas.

The VIS (Verification Interacting with Synthesis) tool, developed at University of California at Berkeley in 1995, is based on synchronous models where any number of components can change state at a time. VIS integrates the verification, simulation, and synthesis of finite-state hardware systems. It uses a Verilog front-end and supports fair CTL model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and hierarchical synthesis.

Generally, when a protocol is implemented in VLSI, such as the ATM protocol stack, it is difficult to be handled by SPIN because the latter cannot reflect the synchronous concurrency aspects of a hardware implementation. The aim of this paper is to compare and contrast the SPIN (XSPIN version 3.2.1) and VIS (VIS release 1.2) tools using a software and a hardware implementation of the ATMR protocol [6] as a case study. We modeled the ATMR in software and in hardware independently and formally verified it in SPIN and VIS, respectively. To expose the advantages and disadvantages of these two types of tools, we report and compare the verification CPU time, memory usage, and state space generated. Furthermore, we describe the modeling techniques of asynchronous and synchronous protocols in SPIN and VIS, and also analyze the source of the complexity in the verification.

The rest of the paper is structured as follows. We begin with an overview of the ATMR model (Section 2). We then describe the ATMR specification and verification in SPIN and VIS in Section 3 and Section 4, respectively. Finally, we conclude the paper with the comparison and contrast of SPIN and VIS (Section 5).

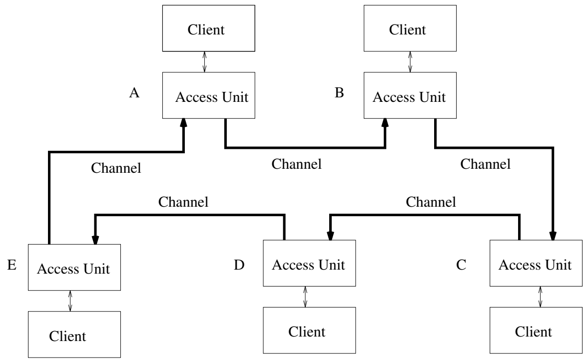

Figure 1. ATMR structure with 5 nodes

2 ATMR Model

The ATMR protocol [6] is a new ISO standard based on a high speed shared medium connecting a number of access nodes by channels in a ring form. Figure 1 gives an example ring with five nodes connected via a channel transferring cells between the nodes. For controlling access to this type of shared medium, the ring is first initialized with a fixed number of ATM cells continuously circulating around the channel from one node to another. Within each access node there is an access unit which performs both the physical layer convergence function and the access control function. Access to the ring is requested by the client and controlled by a combination of a window mechanism and a reset procedure. The client can issue a sending request to the access unit and receive a data cell nondeterministically. The window mechanism limits the number of cells a node can transmit at a time, called the “credits” of this node. The reset procedure reinitializes the window in all access units to a predefine credit value. The format of an ATMR cell is shown in Figure 2. It contains an access control field (ACF), which includes a reset bit, a monitor bit and a busy address. When an access node releases an empty cell, it will fill its own address in the busy address field. The ATM cell is routed by using a ring virtual channel ID (RVCI) in the cell header.

The protocol entity of an access unit starts from an *IDLE* state. When the access unit has cells queued for transmission, it enters a *SEND* state and sends them in empty slots received at the ring interface with the address of the destination in the RVCI field of the cell header. The RVCI field in the header of all cells received at the ring interface of each node is checked and, if the cell is addressed to this node, the cell contents are copied and passed to the appropriate convergence sublayer. The RVCI field is then set to zero, which indicates an empty cell, and the cell is relayed to the next node in the ring. If a match is not found, then, this cell remains unchanged. Transmissions on the ring occur

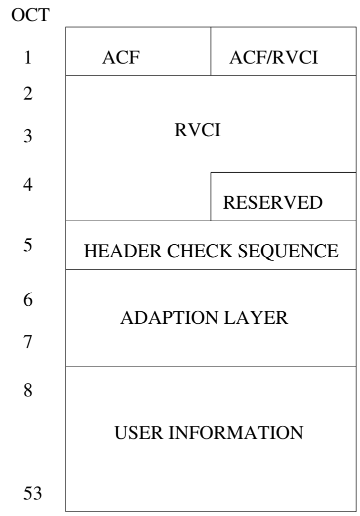

Figure 2. Format of an ATMR cell

in cycles during which each access unit is allocated a fixed window size credit. This credit indicates the number of cells the access unit can transmit in this cycle before issuing or receiving a reset cell from the ring interface. A window credit counter is maintained by each access unit. Whenever this value is less than zero, the protocol entity enters a *WAIT* state to wait for a new credit. This value is initialized to the window size credit each time the ring is reset, namely the protocol entity is in a *RESET* state and the credit is decremented by one each time the access unit transmits a cell from its transmission queue. This mechanism is followed by all access units in the ring and hence eventually all units become inactive and the flow of cells around the ring ceases.

To reinitialize the transmission of the cells, an active access unit always overwrites its own address in the busy address field in the head of all cells passing the ring interface. This way, if an active access node receives a cell with its own address in the busy address field, it concludes that other nodes are inactive. Then after completely sending any remaining queued data from the higher layer, it creates a reset cell by setting the reset bit in the header of the next cell passing the ring interface. The reset cell circulates around the ring and causes all other access units to reinitialize their window credit counters. Once reinitialized, any access unit having data queued for transmission regains the active state and restarts sending cells (see the Promela code given in the Appendix).

This protocol was first checked by Charpentier and Padiou [2] who used UNITY to conduct a pencil-and-paper verification of it. Their validation abstracts away from any implementations, be it in software or in hardware. In next sections, we describe the verification of the ATMR protocol in SPIN and VIS, respectively.

3 The SPIN Verification

SPIN [5] targets the verification of software systems and has been used in the past to trace design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. The tool checks the logical consistency of a specification and reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes.

SPIN uses full LTL model checking, supporting all correctness requirements expressible in linear time temporal logic. It can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed, and verified, without the use of LTL though. Correctness properties can be specified as system or process invariants (using assertions), or as general linear temporal logic requirements (LTL), either directly in the syntax of next-time free LTL, or indirectly as Buchi Automata (called never claims). If a property is invalid, an error trace is provided by the tool.

SPIN uses PROMELA (Process Meta Language) [5] as a modeling language. PROMELA allows abstractions in the protocol description by neglecting details that are irrelevant to process interaction. The intended use of SPIN is to verify fractions of process behavior, which for one reason or another are considered suspect. The relevant behavior is modeled in PROMELA and then verified. The Promela code for ATMR is given in the Appendix.

3.1 ATMR Specification

As an ATMR protocol can have n nodes and p channels [6], we restrict our verification to the model shown in Figure 1 including five ATMR nodes and a channel size of ten cells. The channel length between two neighboring nodes is two cells. We realized through experimentation that this is the maximum model size that can be actually verified within the capability of SPIN and the memory available in the machine we used (Sun Ultra 2 with 768 MB memory).

For the purpose of comparison, we built both an $asynchronous$ and a $synchronous$ ATMR protocol model. In the synchronous model, the channel size is set to zero and the message passing between two neighbor nodes is synchronous. Actually, the channel does not exist in this case.

Due to state space explosion, we did not succeed in verifying the whole datapath of the ATMR model. In order to simplify this latter, we abstract away all the information which will not affect the behavior of the ring accessing scheme, namely the HCS field, the adaption layer field and the user information field. The reduced cell format on which we based our verification is shown in Figure 3, where only 5 bits ACF and 3 bits RVCI will be used (non-shaded boxes in Figure 3). Because we kept all the access control information in the header format, namely the ACF and RVCI fields, the control behavior of ATMR with simplified cell format is exactly the same as that of the original one.

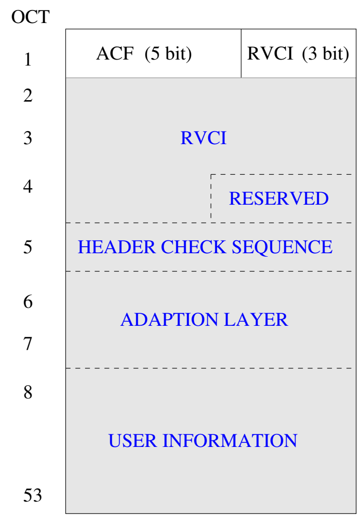

Figure 3. Simplified cell format

3.2 ATMR Verification

After the model is established, it has to be validated against a set of basic consistency properties. For illustration purposes, we present here four properties. In the following descriptions, "[]", "<>", "==" and "→" mean "always", "eventually", logical "equality" and "implication", respectively.

Property 1: Once an access unit exhausts its window size credit, the credit will eventually be renewed, i.e.,

$$[]((credit == 0) \rightarrow <> (credit > 0))$$

where $credit$ stands for the number of credits which is being held by an access unit.

Property 2: A client's request will be eventually acknowledged, i.e.,

$$[]((req == 1) \rightarrow <> (ack == 1))$$

where req is a cell sending request signal from a client to an access unit. If the requested cell has been sent out, the access unit will return an ack signal to the client.

Property	Asynchronous			Synchronous		
	CPU Time	Memory	States	CPU Time	Memory	States
P1	6,033	360.9	49,485,700	40	172	427,995
P2	3,777	141.4	49,358,900	40	173	327,887
P3	–	–	–	46	208	426,044
P4	6,827	166.3	55,348,800	58	145	682,231

Table 1. ATMR verification with SPIN

Property 3: An access unit will eventually exit the RESET state and enter the SEND state, i.e.,

$$[]((state == RESET) \rightarrow <> (state == SEND))$$

where $state$ stands for the current state of an access unit.

Property 4: An access unit will eventually exhaust its window size credit, i.e.,

$$[]((credit > 0) \rightarrow <> (credit == 0))$$

here, we expect that all the credits will be consumed during the sending procedure.

The verification of the above four properties was performed on a 295MHz Sun Ultra-2 workstation with 768MB of memory. We used two kinds of reachability analysis methods provided by the SPIN tool. One is the exhaustive exploration, the other is the supertrace/bitstate exploration which is an approximate approach providing maximum coverage search. In the ATMR verification, we first tried the exhaustive exploration. But this approach obviously cannot finish the ATMR verification, even when we applied the model compress techniques (-DCOLLAPSE, -DMA) in the verification.

In this case, we have to use the supertrace/bitstate instead of the exhaustive exploration. SPIN can finish the verification of some properties, see Table 1. Although the coverage is not one hundred percent, it still can give us some confidence about the correctness of the model. In order to test the verification coverage of SPIN, we put a piece of unreachable code in the ATMR model. The supertrace/bitstate model checking experimental results are reported in Table 1, including CPU time in seconds, memory usage in MB and the number of states visited.

Table 1 shows that the piece of unreachable code was detected and three of the properties can be completed. There is still one property (Property 3) which cannot be completed due to an "out of memory" error although the piece of unreachable code was detected. From Table 1, we also notice that the consumption of CPU time and memory for the asynchronous ATMR are considerable due to the large quantities of cell interleaving in the *channel*. Compared to the asynchronous case, the synchronous ATMR verification is much easier because there are no spare cells in the channel.

We are sure that more reduction and abstraction shall be required to reduce the complexity of the model in order to complete the verification. The most important problem in the abstraction is to preserve the correctness of the properties, namely, if the abstract model satisfies the properties then the concrete model should satisfy the properties as well, and false properties in the abstract model are still false in the concrete model. For the moment, how to make a good abstraction is still an unsolved problem in formal verification. Fortunately, in most protocol verifications, we can use the data independent abstraction technique as described in [9]. A data independent system is a system where the datapath values do not affect the course of the computation. For example, a protocol whose only function is to move data from a sender to a receiver (without payload error checking, etc.) is typically data independent. Verification for such systems can be done using only the control structure; the data can be abstracted away entirely. In our case, since we want to check the ring access mechanism, so we can remove the payload and other irrelevant information in the cell, and this abstraction is sound.

Since SPIN is a protocol verification tool, some protocol correctness features can be directly expressed in the model, like deadlock, live-lock, and unreachable code. However, it is not easy to verify a temporal formula in SPIN, because we have to find a balance point between "max search depth" and "max physical memory" during the verification [5]. Another problem in SPIN is that it requires so much memory during the verification that some properties cannot be completed due to an "out of memory" error.

4 The VIS Verification

VIS [1] is a verification and synthesis tool for finite-state hardware systems, developed at University of California at Berkeley and University of Colorado at Boulder. It uses a Verilog front-end and supports fair CTL model checking, language emptiness checking, combinational and sequential equivalence checking, cycle-based simulation, and

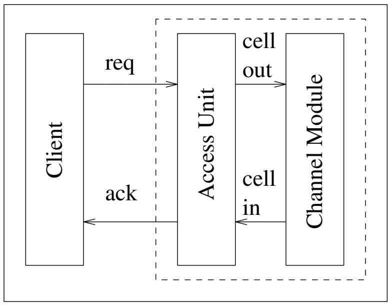

Figure 4. Modified ATMR node as a closed system

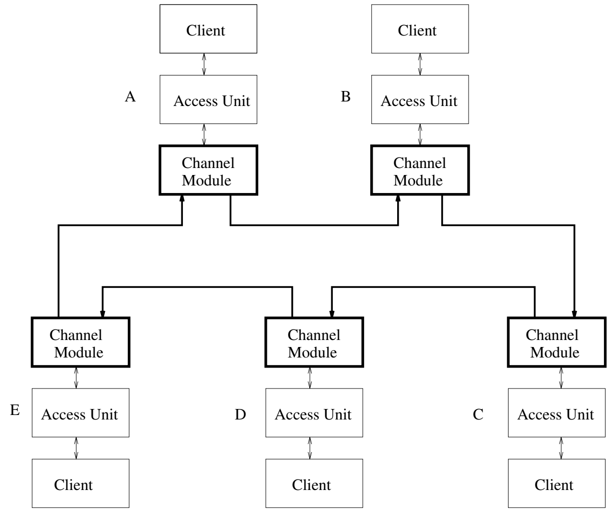

Figure 5. Modified ATMR ring structure

hierarchical synthesis. Its fundamental data structure is a multi-level network of latches and combinational gates. The variables of a network are multi-valued, and logic functions over these variables are represented by an extension of BDDs: multi-valued decision diagrams. VIS operates on the intermediate format BLIF-MV. It includes a compiler from Verilog to BLIF-MV and extracts a set of interacting FSMs which preserves the behavior of the Verilog program defined in terms of simulated results. Through the interacting FSMs, VIS performs fair CTL model checking under Buchi fairness constraints. The language of a design is given by sequences over the set of reachable states that do not violate the fairness constraint. Also VIS can check the combinational and sequential equivalence of two designs. Sequential verification involves building the product FSM, and checking whether a state where the values of corresponding outputs differ can be reached from the set of initial states of the product machine. If model checking or equivalence checking fails, VIS reports the failure with a counterexample trace.

VIS has three main parts: a front-end to read and traverse a hierarchical system described in BLIF-MV; a verification core, to perform model checking of fair CTL and test language emptiness; and a capability to optimize logic modules.

4.1 ATMR Specification

Since VIS is built on synchronous models, it is impossible to directly describe the original asynchronous ATMR in VIS, e.g., how to describe the cell transmission between two access units using synchronous Verilog. We hence need to simulate the ATMR protocol in the synchronous VIS environment. There are many methods to simulate an asynchronous system in a synchronous environment. Here, because we only request that cell transmission be asynchronous and the module itself be synchronous, we propose to simply add a module *channel* in the Verilog specification.

This channel model will play the role of a queue between two ATMR nodes (see Figure 4).

All the cells sent or received by the access unit will hence be queued in the channel module. When the access unit wants to read a cell from the channel, it actually reads the cell from the head of the queue. If the destination is the current node, the cell will be processed in this access unit. Otherwise, the cell will be forwarded to the next node via the channel module. This way, the sending and the receiving processes within the ring can remain asynchronous.

Except above feature, the ATMR model (Figure 5) we verified in VIS is very similar to that we used in SPIN. The cell format is here again a simplified one, which only has ACF and RVCI fields (Figure 3). The client module randomly sends data request to the access unit and receives the acknowledgment. Note that given the specification nature in SPIN and VIS, all components in VIS are true concurrent, while they are interleaved in SPIN.

As we did in the SPIN verification, we built a synchronous model of the ATMR MAC. This implementation is hypothetical and does not comply with the intended ATMR MAC specification which is supposed to be asynchronous. Its structural model is similar to the asynchronous version, with five nodes in the ring and each node having a client and an access unit. In difference, however, no channel module is needed. The cell transmission between two neighboring nodes is achieved directly from the output/input port of one node to the input/output port of the next one. The communication is synchronized by the system clock. Besides this feature, the behavior of the access unit as well as the cell format are exactly the same as the asynchronous case.

4.2 ATMR Verification

We verified the same properties as in the SPIN study. The only difference is that, in VIS, properties will be expressed in CTL and not in LTL. We present here the four properties in CTL. In the following descriptions, "=", "→" and " ∧" mean logical " equality", "implication" and "and", respectively. "AG" and "AF" mean "in all paths always" and "in all paths eventually", respectively.

Property 1: Once an access unit exhausts its window size credits, the credits will eventually be renewed.

$$AG(((credit[2] = 1) \wedge (credit[1] = 1) \wedge$$
$$(credit[0] = 0)) \rightarrow AF(credit[0] = 1));$$

$credit$ stands for the number of credits which is being held by an access unit. $credit$ is composed of three bits: $credit[2]$, $credit[1]$ and $credit[0]$.

Property 2: A client's request will eventually be acknowledged.

$$AG((req = 1) \rightarrow AF(ack = 1));$$

req is a cell sending request signals from a client to an access unit. If the requested cell has been sent out, the access unit will return an ack signal to the client.

Property 3: An access unit will eventually exit the RESET state and enter the SEND state.

$$AG((state = RESET) \rightarrow AF(state = SEND));$$

$state$ stands for the current state of an access unit.

Property 4: An access unit will eventually exhaust its window size credit.

$$AG(((credit[2] = 1) \wedge (credit[1] = 1) \wedge$$
$$(credit[0] = 0)) \rightarrow AF((credit[2] = 0) \wedge$$
$$(credit[1] = 0) \wedge (credit[0] = 0)));$$

In this property, we expect that all the credits will be consumed during the sending procedure.

The experimental results of CTL model checking obtained in VIS are reported in Table 2, including CPU time in seconds, memory usage in MB and the number of BDD nodes generated. These experiments were conducted on the same machine as the SPIN verification. Generally, from the table, we can see that the state space of the hardware implementation is larger than that of the software implementation because of the parallel composition of the system components. However, all properties checking could be finished thanks to the powerful BDD manufacture techniques in VIS. During the verification, we used the advanced ordering option [1] to reduce the BDD/MDD size.

Here again, the VIS verification approach is not directly scalable to large designs due to state space explosion. To solve this problem the data complexity must be decreased by abstraction and reduction as we did in the SPIN verification, and these abstraction and reduction techniques are sound in VIS as well.

In the verification, the verified 5-node model has 220 latches, which is beyond the capability of VIS. For instance, Figure 6 shows the growth of the state space with respect to

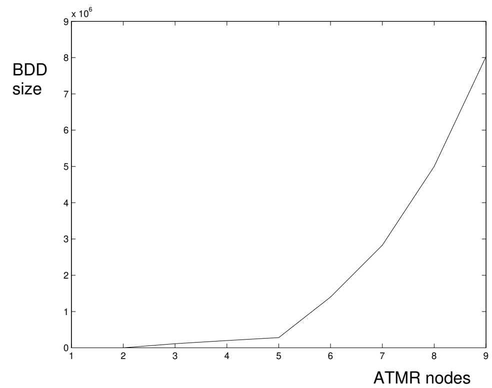

Figure 6. BDD size vs. number of ring nodes

the number of nodes for Property 2. The actual state space during the verification, however, varies depending on different properties. For example, the BDD size of Property 2 is 4,423,598, while that of Property 3 is 1,928,041,758. Regarding Property 3, an ATMR node will enter the reset state only after all the remaining cell in the ring are sent out. It is supposed to be the most difficult property to be verified with respect to the BDD size. From Table 2, we can see that it takes 27,400.5 seconds and 371 MB memory to verify this property. There are several reasons why the BDD size may blow up. First, the datapath in the ring will increase according to the number of nodes (Figure 6). Second, the variables are circular dependent due to the ring access. Third, the performance of the model checking itself is not steady in various applications.

As we see from these results, even though we used a reduced cell format and an abstract environment, the CPU time and memory usage are large, e.g., up to 20 hours of machine time and 623 MB memory for Property 1.

Given a transition system M and a linear temporal formula ϕ, the model checking problem for M and ϕ is to decide whether ϕ holds in all the computation of M. When ϕ is a branching temporal logic formula, the problem is to decide whether ϕ holds in the computation tree of M.

Property	Asynchronous			Synchronous		
	CPU Time	Memory	States	CPU Time	Memory	States
P1	49,616.9	623	1,376,130,560	5.6	12	614,833
P2	509.1	28	4,423,598	5.4	12	308,419
P3	27,400.5	371	1,928,041,758	5.7	12	921,727
P4	44,093.6	579	1,193,682,251	6.0	12	1,230,117

Table 2. ATMR verification with VIS

The "folk" wisdom is that the linear temporal logic model checking is PSPACE-complete [7], while the branching time temporal logic model checking is linear [3]. This obviously makes the SPIN verification less efficient than that of VIS. However, SPIN uses an on-the-fly model checking algorithm [5] which does not construct the whole state space at the beginning of the verification. This makes the verification of an LTL formula easy in some cases. The verification results depend on what type of properties are to be verified. Tables 1 and 2 show that in the worst case (Property 3) the branching time computation has superiority over that of linear time computation. In the verification of other properties, this superiority is not that clear.

In the verification, we also checked all the above correctness properties against the synchronous model. The experimental results of model checking obtained with VIS are given in Table 2. Compared to the results of the asynchronous verification (Table 2), we see the tremendous cost of the simulation of the asynchronous model. This is mainly due to the fact that the simulated asynchronous model contains 220 latches, while the synchronous model has only 125 latches. The introduction of the channel modules creates a much larger state space.

5 Conclusions

In this paper, we modeled and formally verified the asynchronous and synchronous ATMR protocol in both SPIN and VIS. Generally, when a protocol is implemented in hardware, it cannot be handled only by a software (protocol) verification tool, like SPIN, since most of these tools are based on an interleaving semantics and cannot reflect the true concurrency aspects of a hardware implementation. A verification in VIS leaves us however, with the obligation of simulating an asynchronous protocol in a synchronous environment. An example of such simulation for the asynchronous ATMR protocol in the synchronous VIS environment is shown in this paper.

Because of the inherent limitations of model checking, both SPIN and VIS are not directly scalable to large systems due to state space explosion. Thus, it is important to find techniques that can be used in conjunction with model checking tools like SPIN and VIS to extend the size of the systems that can be verified. In this paper, we used a data abstraction approach to reduce the model of the ATMR protocol for both the SPIN and VIS verifications.

Unlike VIS, SPIN is based on interleaving models, and hence runs generally faster than VIS because each state update is a simpler operation, being restricted to one component only. Also LTL verification is more efficient than CTL verification for the same state space. However, in some special cases, VIS runs faster than SPIN depending on the verification model. For example, the verification of Property 2 in VIS is faster than in SPIN. In all synchronous cases, we noticed that VIS is faster than SPIN. On the other hand, in order to verify an asynchronous protocol in VIS, we have to simulate the asynchronous procedure in the synchronous environment. This kind of simulation tremendously increases the verification time and memory usage (see Table 2). The reason is that in the simulated asynchronous model there are 220 latches while in the synchronous model only 125 latches are used. The latch difference creates a much larger state space.

In addition, it seems that the memory is more efficiently used by VIS than SPIN when the state space is very large. For instance, in VIS, all the four example properties can be completed successfully, while SPIN can only complete three of them because it runs out of memory for the fourth one.

Another weakness in VIS is that it cannot directly report deadlocks, live-locks and unreachable code. We have to express these properties with temporal formulas. For example, a deadlock, is expressed as: "Sender is not in send state and receiver is not in receiving state and there is at least one cell in the channel." Generally, this specification is awkward in CTL.

Finally, two practical features of these tools are worth mentioning. VIS has a Verilog front-end allowing industrial designs to be imported and verified. On the other hand unlike VIS, SPIN comes with a graphic user interface, which greatly eases the usage of the tool. Anyhow, further case studies have to be considered for the formal comparison of SPIN and VIS.

Based on the results obtained through out the ATMR case study in this paper, Table 3 gives a relative summary of the main comparison features discussed.

Feature	SPIN	VIS
Target system	Software	Hardware
Basic model	Interleaving model	Synchronous model
Property language	LTL	CTL
Modeling language	PROMELA	Verilog
Verif. of asynch. protocol	Yes	Additional channel module
CPU time usage	Faster	Slower
Main memory usage	Larger	Smaller
Detect dead-lock, live-lock, etc.	Yes	Indirectly via temporal formulas
Commercial front-end	No	Yes
Graphic User Interface	Yes	No

Table 3. Comparison of SPIN and VIS for ATMR

Acknowledgments

This work is partially supported by a Concordia graduate student scholarship and NSERC research grants no. OGP0194302 and no. OGP0194234.

References

[1] R. K. Brayton et al. VIS: A system for verification and synthesis. In T. Henzinger and R. Alur, editors, *Eigth Conference on Computer Aided Verification (CAV'96)*, pages 428–432. Springer-Verlag, Rutgers University, NY, July 1996.

[2] M. Charpentier and G. Padiou. Specification and verification of the ATMR protocol using UNITY. In *International Workshop on Formal Methods for Parallel Programming*, pages 26–36, University of Geneva, Switzerland, April 1997.

[3] E. M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finitestate concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[4] E. M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *REX school/symposium on A decade of concurrency: reflections and perspectives*, pages 124–175, Noordwijkerhout, The Netherlands, June 1993.

[5] G. J. Holzmann. The spin model checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[6] ISO. *Specification of the Asynchronous Transfer Mode Ring (ATMR) Protocol*, 2.0 edition, Jan. 1993.

[7] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[8] H. Peng, S. Tahar, and F. Khendek. Comparison of SPIN and VIS for protocol verification. Technical report, Department of Electrical and Computer Engineering, Concordia University, Montreal, Canada, Feburary 1999.

[9] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 184–192, St. Petersburgh, January 1986.

Appendix: PROMELA model of the ATMR [8]

```
proctype AccessUnit(byte ID; chan in, out){
byte State=Idle;
MSDU_struct data;
start: do
  ::(State==Idle)->
    if
    ::(Msgs[ID]>0)->
      if
      ::(Credit[ID]>0)->State=Send;
        goto start;
      ::(Credit[ID]<=0)->
        State=WaitCredit;goto start;
      fi
    ::(Msgs[ID]==0)->
      Msgs[ID]=MaxMsgs;
      ack[ID]=0; req[ID]=1;
      if
      ::in?DataCell(data)->
        if
```

```
        ::(data.Dest_Add==ID)->
          data.Dest_Add=0;
          out!EmptyCell(data);
        ::(data.Dest_Add!=ID)->
          out!DataCell(data);
        fi
        ::in?ResetCell(data)->
          Credit[ID]=MaxCredit;
          out!ResetCell(data);
        ::in?EmptyCell(data)->
          if
          ::(data.Busy_Add==ID)->
            Receive_RstReq[ID]=1;
            out!ResetCell(data);
            Send_Rst[ID]=1;
            State=Reset;
          ::(data.Busy_Add!=ID)->
            out!EmptyCell(data);
            State=Idle;
          fi;
        Msgs[ID]=
         ((A[ID]*PreMsgs[ID]+C[ID]);
        ack[ID]=0; req[ID]=1;
        PreMsgs[ID]=Msgs[ID];
        fi;
  fi;
::(State==Send)->
      if
      ::in?EmptyCell(data)->
        data.Dest_Add=
         ((A[ID]*PreDest[ID]+C[ID]);
        if
        ::(data.Dest_Add==ID)->
           data.Dest_Add=(data.Dest_Add;
        ::(data.Dest_Add!=ID)->;
        fi;
        PreDest[ID]=data.Dest_Add;
        data.Busy_Add=ID;
        out!DataCell(data);
        Msgs[ID]--;
        ack[ID]=1; req[ID]=0;
        Credit[ID]--;State=Idle;
      ::in?DataCell(data)->
        data.Busy_Add=ID;
        if
        ::(data.Dest_Add==ID)->
           data.Busy_Add=ID;
           data.Dest_Add=0;
           out!EmptyCell(data);
        ::(data.Dest_Add!=ID)->
           data.Busy_Add=ID;
           out!DataCell(data);
        fi
      ::in?ResetCell(data)->
      Credit[ID]=MaxCredit;
      out!ResetCell(data);
      State=Idle;
      fi
::(State==Reset)->
      if
      ::in?DataCell(data)->
         if
         ::(data.Dest_Add==ID)->
            data.Dest_Add=0;
            out!EmptyCell(data);
         ::(data.Dest_Add!=ID)->
            out!DataCell(data);
         fi
      ::in?ResetCell(data)->
        Credit[ID]=MaxCredit;
        Send_Rst[ID]=0;
        Receive_RstReq[ID]=0;
        data.Busy_Add=ID;
        out!EmptyCell(data);
        State=Idle;
      ::in?EmptyCell(data)->
        out!EmptyCell(data);
      fi
::(State==WaitCredit)->
      if
      ::in?DataCell(data)->
         if
         ::(data.Dest_Add==ID)->
            data.Dest_Add=0;
            out!EmptyCell(data);
         ::(data.Dest_Add!=ID)->
            out!DataCell(data);
         fi
      ::in?EmptyCell(data)->
         if
         ::(data.Busy_Add==ID)->
            Receive_RstReq[ID]=1;
            out!ResetCell(data);
            Send_Rst[ID]=1;
            State=Reset;
         ::(data.Busy_Add!=ID)->
            out!EmptyCell(data);
         fi
      ::in?ResetCell(data)->
         Credit[ID]=MaxCredit;
         out!ResetCell(data);
         State=Idle;
      fi
od
}
```