Int J STTT (2002) / Digital Object Identifier (DOI) 10.1007/s100090200073

Comparison of SPIN and VIS for protocol verification

Hong Peng, Sofiène Tahar, 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

Published online: 1 March 2002 –

Abstract. In this paper, we compare and contrast SPIN and VIS, two widely used formal verification tools. In particular, we devote 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 Asynchronous Transfer Mode Ring (ATMR) medium access protocol using SPIN and its hardware model using VIS. We believe that this study is of particular interest as more and more protocols, like ATM protocols, are implemented in hardware to match high-speed requirements.

Keywords: SPIN – VIS – Model checking – Formal verification – Protocols

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 [6]. 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, namely, a given design is validated against specific and general properties. Two different fields of model checking have arisen: formal verification of software protocols and software systems, such SPIN [9], and formal verification of digital hardware, such as VIS [2].

The SPIN software verification tool, developed by G. J. Holzmann at Bell Labs in 1989, 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 if the protocol specification is logically consistent. It reports errors in the protocol such as deadlock, livelock, or unreachable code. It also validates properties specified as linear time temporal logic (LTL) [8] formulas.

The VIS (Verification Interacting with Synthesis) tool, developed in 1995 by University of California at Berkeley and University of Colorado at Boulder, is based on synchronous models where any number of components can change state at a time. VIS integrates formal verification, simulation, and synthesis of finite-state hardware systems. It uses the Verilog hardware description language (HDL) as its input language. VIS supports branching time temporal logic (CTL) [8] symbolic model checking with fairness constraints [13].

The aim of this paper is to compare and contrast the SPIN (XSPIN version 3.3.3) and VIS (VIS release 1.3) tools using a software and a hardware model of the ATMR protocol [12] as a case study. We developed the software and hardware models independently and formally verified them in SPIN and VIS, respectively. Since the modeling language of SPIN and VIS are different, we cannot say explicitly that the two verified models, the VIS and the SPIN one, are exactly the same with respect to their semantics. However, we did follow the modeling and coding style of each of these tools. 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 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 protocol (Sect. 2); we then describe the ATMR specification and verification in SPIN (Sect. 3) and VIS (Sect. 4), respectively;

finally, we conclude the paper with the comparison and contrast of SPIN and VIS (Sect. 5). The PROMELA and Verilog codes of the ATMR protocol are provided in the Appendix.

2 ATM ring protocol

The Asynchronous Transfer Mode Ring (ATMR) protocol [12] is an ISO standard based on a high-speed shared medium connecting a number of access nodes by channels in a ring topology. 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. 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 Fig. 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 state transition diagram of the ATMR is shown in Fig. 3, where "?" means receiving a message. The protocol entity of an access unit begins 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

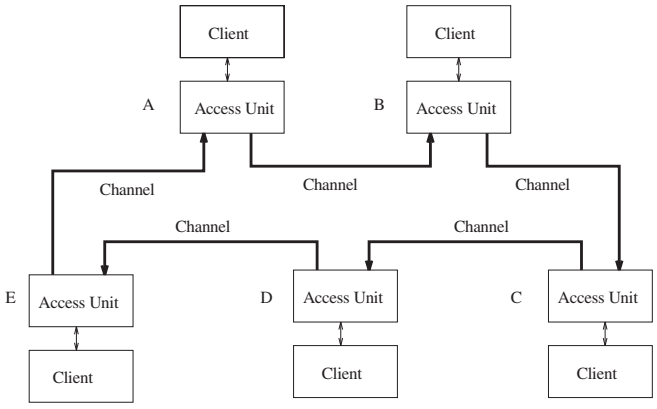

Fig. 1. ATMR structure with 5 nodes

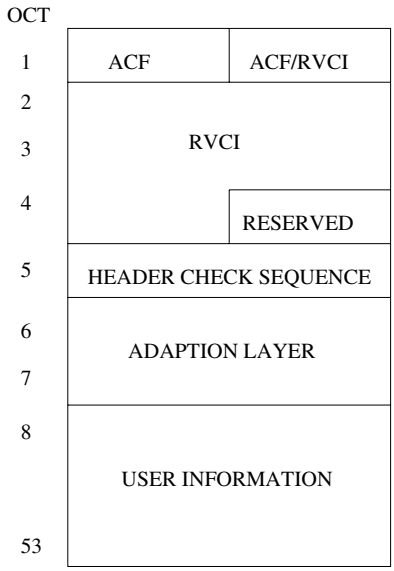

Fig. 2. Format of an ATMR cell

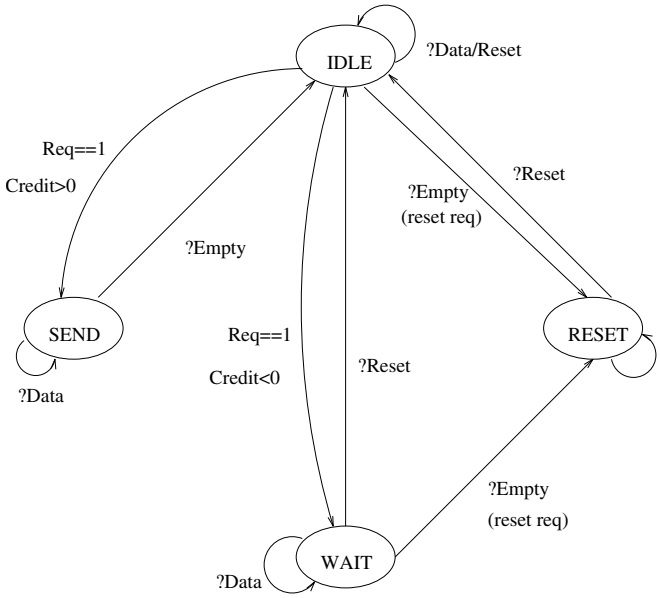

Fig. 3. FSM of an ATMR entity

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 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.

The ATMR protocol was first modeled and checked by Charpentier and Padiou [4] 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 modeling and verification of the ATMR protocol in SPIN and VIS, respectively.

3 Verification using SPIN

SPIN [9] 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. [3, 10]. The tool checks the logical consistency of a protocol specification and reports design errors like deadlock, livelock, unreachable code and so on.

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 Büchi Automata (called never claims). If a property is invalid, an error trace is provided by the tool.

SPIN uses PROMELA (Process Meta Language) [9] as input 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 verified.

3.1 ATMR specification

In order to test the capability of the SPIN tool, we tried to build a model as large as possible and let the tool do the reduction work. In this way, the verification engine works on its up-limit load, so that we can test the performance of the engine in a real situation. As an ATMR protocol can have n nodes and p channels [12], we will perform our verification on the model shown in Fig. 1 including 3, 4, 5, and 6 ATMR nodes and a channel size of 6, 8, 10, and 12 cells. The channel length between two neighboring nodes is two cells. We realized through experimentation that the five node model is the maximum model size that can make a comparison between SPIN and VIS within the memory available in the machine we used (Sun Sparc with 2 GB memory). However, for the purpose of comparison, we also put the experimental results of the ATMR model with three, four, five, and six nodes.

In the SPIN ATMR model, each node is specified as a process

```
proctype Atmr(byte ID; chan in, out)
```

where *ID* is the identification of the present node; *in* is the input channel and *out* is the output channel of the node. Since the nodes are in a ring form, the input channel of node B, for instance, will be the output channel of node A (Fig. 1).

Since SPIN's strength is in proving properties of interactions in a distributed system, but not in proving things about local computation or data dependency, we can try to make the model more general, more abstract. Namely, we will put only the behavior between the access unit and the channel into the model. Besides this, we assume that the queue between the client and the access unit will be automatically refilled once it is empty. Thus, we can have a simple model while not affecting the behavior of the access unit.

An additional way of reducing the complexity is to remove everything that is not related to the property we are trying to prove, such as redundant data. For example, due to state space explosion, we did not succeed in verifying the whole data-path 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 Fig. 4, where only 5 bits ACF and 3 bits RVCI will be used (non-shaded boxes in Fig. 4). 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. After the reduction, the structure of the cell becomes:

```
typedef MSDU_struct {
          byte Busy_Add;
          byte Dest_Add;
                    };
```

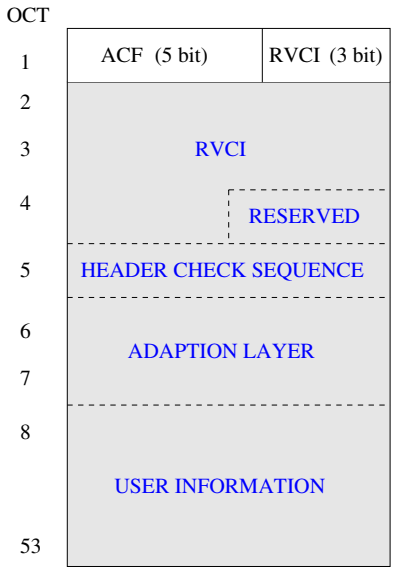

Fig. 4. Simplified cell format

where *Busy_Add* is the busy address and *Dest_Add* is the destination address. *MSDU_struct* is the type definition of the cell. The cells are classified into *DataCell*, which contains user data, *EmptyCell*, which is available for loading, and *ResetCell*, which is to reset the credit of the access units in the ring.

Asynchronous channels are a significant source of complexity in the verification since there are lots of interleavings in the channel. Generally, the exclusive read/write option provided by SPIN is a good partial-order reduction approach [11], which can reduce the verification CPU time.

Besides this, in order to reduce the interleavings in the model, one of the possible solutions is to make as many statements as possible become *atomic*. For example, in the initialization process, we put all the initialization statements as *atomic*:

```
init
{
 MSDU_struct d;

 atomic{ d.Dest_Add=0;
         Credit[1]=MaxCredit;
      ......
        }
```

We can also reduce the interleavings of the model significantly by making atomic each state transition. For example, instead of

```
:: (State == state_name)->
      other_statements
```

we can use

```
:: atomic{(State == state_name) ->
      other_statements}
```

The exhaustive experiments we conducted show that the state space can be reduced for at least one order of magnitude in this way. However, in this case, the PROMELA model becomes synchronous, which is not our intention. In the sequel, we did not use these atomic statements.

The PROMELA ATMR model is shown in Appendix A, where *ID* is the identifier of this unit, and *in* and *out* are the incoming and outgoing channels of this unit, respectively. There are four states, *Idle, Send, Reset*, and *WaitCredit*. In each state, the unit can receive *DataCell, EmptyCell, ResetCell*.

An advantage of SPIN is that we can easily check deadlock using timeout statement in the model. Since in the deadlock status, the state transition stops, the timeout statement in a state can be easily checked out.

3.2 ATMR verification

Once the ATMR model was established, we validate it against a set of basic consistency properties. For illustration purposes, we present here six properties including liveness and safety. In the following descriptions, "[]", "<>", "==" and "→" mean "always", "eventually", "logic equality", and "imply", respectively.

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

$$[]((credit == 0) \rightarrow <> (credit == 6))$$

where *credit* stands for the number of credits which is being held by an access unit and 6 is the preset maximum value.

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

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

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

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

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

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

$$[]((credit = 6) \rightarrow <> (credit == 0))$$

here, 6 is the preset maximum credit value. We expect that all the credits will be consumed during the sending procedure.

Property 5: The number of reset cells in the ring cannot exceed the number of access units.

$$[](NumofRst < NumofUnit)$$

where *NumofRst* is the number of reset cells in the ring and *NumofUnit* is the number of access units.

Property 6: In the SEND state, a given station cannot send more cells than allowed by its credits.

$$[]((state == SEND) \rightarrow (Outmsgs <= 6))$$

here, *Outmsgs* is the number of cells sent by a given station in the SEND state.

The verification of the above six properties was performed on a Sun Sparc workstation with 2 GB 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, which can only provide maximum coverage search. In the ATMR verification, we first tried the exhaustive exploration. But this approach could not finish the ATMR verification due to an out-of-memory error, even when we applied the model compress techniques (-DCOLLAPSE, -DMA).

In contrast, the supertrace/bitstate (bit-state hashing) could finish the verification of the properties. Although the coverage is not 100%, this latter still can give us some confidence about the correctness of the 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 stored. Graphic illustrations of the experimental results are plotted in Figs. 5, 6, and 7.

From the graphic illustrations, we found that the increment of the state space is becoming steady when the model becomes larger, and so does the CPU time. This means that SPIN can handle larger models, while, affecting the state coverage (i.e., the number of visited stated relative to the number of actual states), however. Generally, For a hash-factor between 10 and 100, SPIN gives an expected coverage of 98% on average.

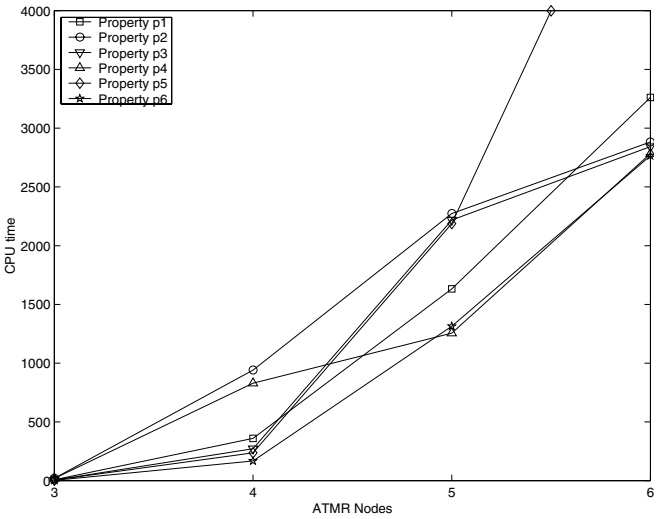

Fig. 5. SPIN verification CPU time

Bit-state hashing is an approximate approach. On the other hand, when compared with classical random simulation techniques, it is always better to use bit-state hashing because the coverage is usually much better than that achieved with random simulation. During the verification, we found that the more nodes are included in the ATMR model, the less is the coverage. In the 3-node verification, the coverage is greater than 99.9%, but in the 6-node verification, the coverage is less than 98%.

There are some variance in the memory usage, especially in the 6-node model for Property 3. We think there may be two reasons. One is that we are using the approximate method. This method is actually a "random" approach. The other is that we are working in a multi-user operating system. The variance in the system load will affect the experimental results.

Table 1. ATMR verification with SPIN

	3 nodes			4 nodes		
Property	CPU Time (s)	Memory (MB)	States	CPU Time (s)	Memory (MB)	States
P1	7.5	46.084	1 335 725	359.8	89.802	6.61796e6
P2	19.7	45.982	340 879	941	87.345	1.56626e7
P3	5.2	46.084	122 671	271.6	90.007	4.648e6
P4	18.6	46.084	390 387	828.9	73.521	1.5796e7
P5	5.5	46.084	92 322	236.6	89.086	4.3443e6
P6	3.2	44.982	101 015	167.3	45.187	5.95003e6
	5 nodes			6 nodes		
Property	CPU Time (s)	Memory (MB)	States	CPU Time (s)	Memory (MB)	States
P1	1632.5	127.225	3.26899e7	3261.4	1192.077	4.36862e7
P2	2273.6	264.906	3.41975e7	2883	264.19	4.98607e7
P3	2218.1	962.66	3.22636e7	2842.7	258.998	3.61413e7
P4	1255.3	258.25	2.9685e7	2783.9	298.487	3.69046e7
P5	2187.5	1441.086	4.32634e7	–	–	–
P6	1313.8	584.864	3.33867e7	2765	1729.955	4.53676e7

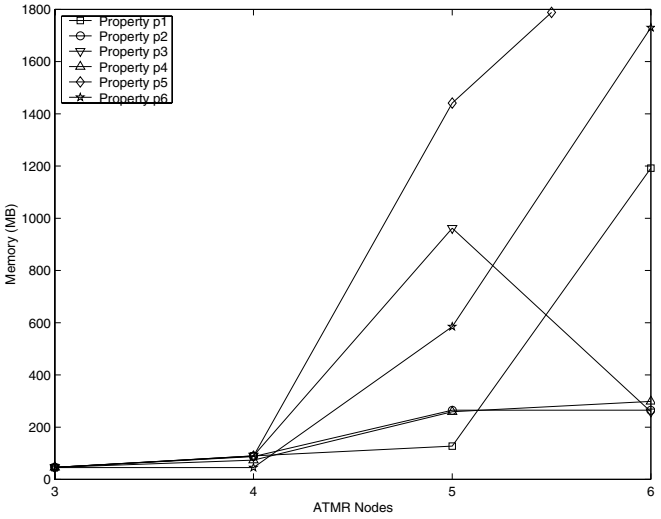

Fig. 6. SPIN verification memory usage

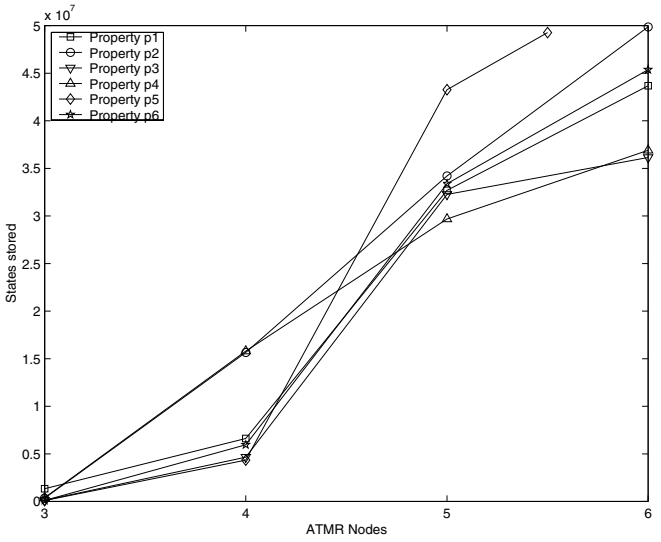

Fig. 7. SPIN verification state space

4 Verification using VIS

VIS [2] 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 the Verilog HDL as the input language and supports CTL model checking with fairness constraints. 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 [5]. It includes a compiler from Verilog to BLIF-MV and extracts a set of interacting FSMs that preserves the behavior of the Verilog program defined in terms of simulated results. Through the interacting FSMs, VIS performs CTL model checking under Büchi fairness constraints, i.e., sets of states that must be visited infinitely often. The language of a design is given by sequenceso ver the set of reachable states that do not violate the fairness constraint. If model checking fails, VIS reports the failure with a counter-example.

Besides model checking, VIS supports equivalence checking, cycle-based simulation, and synthesis functions, such as state minimization and state encoding.

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 build a pseudo-asynchronous ATMR protocol to simulate the ATMR protocol in the synchronous VIS environment. There are many methods to simulate an asynchronous system in a synchronous environment [1]. 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 Fig. 8).

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. The channel is defined as follows:

```
channel (ch_out, ch_in, ID);
```

where *ch_out* and *ch_in* are wired connections to and from the nodes; *ID* is the identification of the channel. In this case, the access unit becomes:

```
mac_ctrl_node (req, ack, ch_out, ch_in, ID);
```

where *req* is the cell request signal from the client; *ack* is the acknowledgment; *ch_out* and *ch_in* are the output

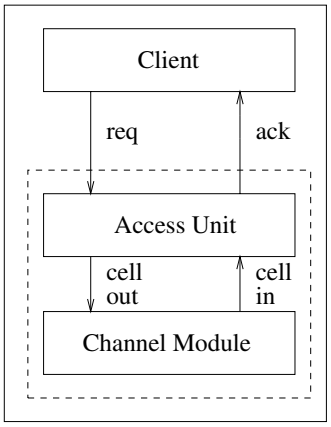

Fig. 8. Modified ATMR ring structure

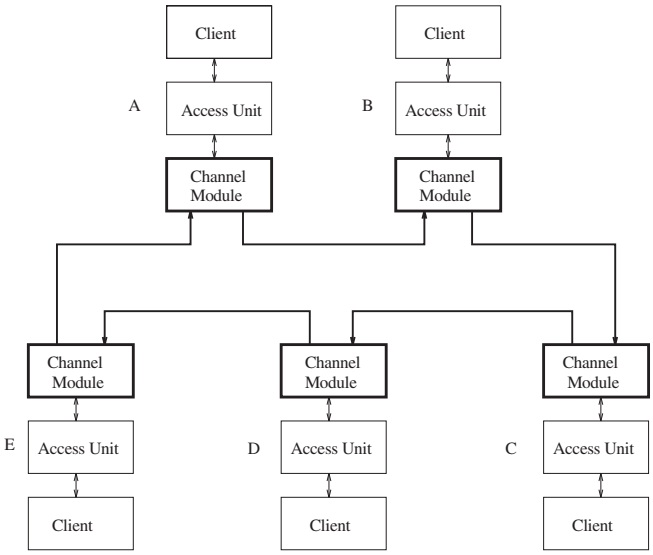

Fig. 9. Modified ATMR ring structure

and input channels for each node; and ID is the identification of the node. Here, we do not put the clock signal because we use the implicit clock source provided by VIS. The *req*/*ack* pair follows the same rule as we defined in the SPIN modeling, namely, once *ack* becomes true, *req* will be true in the next clock. Because Verilog instances are synchronized by the clock, we have to put the *req* generator in another instance and put a wire connection between these two instances.

Except for the above features, the ATMR model (Fig. 9) we verified in VIS is very similar to that we used in SPIN. The cell format is here again a simplified one, containing only the ACF and RVCI fields (Fig. 4). Note that given the specification nature in SPIN and VIS, all components in VIS are true concurrent, while they are interleaved in SPIN.

The Verilog pseudo-asynchronous ATMR model is given in Appendix B, where *clk* is the system clock; *req* and *ack* are the signals from/to the clients; *out_cell* and *in_cell* are the output/input cells of this unit; *id* is the identifier of this unit. The states and the cell types are the same as that of SPIN model. The only difference is that because Verilog does not have *chan* (channel) data type and *mtype* (message type) variable, we have to examine the data bit in the cell format explicitly.

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 six liveness and safety properties of Sect. 3.2 in CTL. In the following descriptions, "=", "$\rightarrow$", and " $*$" mean logical " equality", "implication", and "and", respectively. "AG" and "AF" mean "all paths in all states" and "all paths in future states", respectively.

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

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

where *credit* is composed of three bits: $credit[2]$, $credit[1]$ and $credit[0]$.

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

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

Property 3: An access unit will eventually exit the RESET state and enter the SEND state (see Fig. 4).

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

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

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

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

Property 5: The number of reset cells in the ring cannot exceed the number of access units.

$$AG(NumofRst < NumofUnit);$$

In this property, *NumofUnit* is set to the number of access units in the verification, i.e., 3, 4, 5, 6, respectively.

Property 6: At SEND state, a given station cannot send more cells than allowed by its credits.

$$AG((state == SEND) \rightarrow (Outmsgs <= 6))$$

here, *Outmsgs* is the number of cells which a given station sends at the SEND state.

The experimental results of the 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 allocated. The graphical representations are given in Figs. 10, 11, and 12. These experiments were conducted on the same machine as the SPIN verification. During the verification, we used the advanced ordering, *window* and *sift* [2] to reduce the BDD/MDD size. VIS also provides a cone of influence model reduction [7] technique for invariant properties. However, in the verification of liveness properties, this technique cannot be applied. Besides this, VIS provides a limited abstraction mechanism, namely the user must explicitly specify which signal in the model can be abstracted in one specific property verification. This technique, however,

Table 2. ATMR verification with VIS

	3 nodes			4 nodes		
Property	CPU Time (s)	Memory (MB)	States	CPU Time (s)	Memory (MB)	States
P1	57	13.59	1 513 196	4290	26	680 963 412
P2	4.6	10.04	318 106	10.1	11.85	732 167
P3	6.3	11.91	528 411	1962.1	201.69	346 168 207
P4	25.6	13.72	1 166 100	467.7	19.99	83 736 894
P5	4.1	9.87	308 435	8.1	11.72	550 583
P6	4.8	9.98	314 168	9.6	11.66	565 248
	5 nodes			6 nodes		
Property	CPU Time (s)	Memory (MB)	States	CPU Time (s)	Memory (MB)	States
P1	19640.7	124.49	1 811 363 276	–	–	–
P2	31	13.79	3 920 178	79324.5	844.01	3 947 015 525
P3	20146.9	118.98	1 829 837 953	–	–	–
P4	11372.3	105.04	1 152 672 565	–	–	–
P5	14734.3	289.57	1 231 510 174	–	–	–
P6	24.2	13.77	2 492 626	70920.1	752.69	2 606 471 216

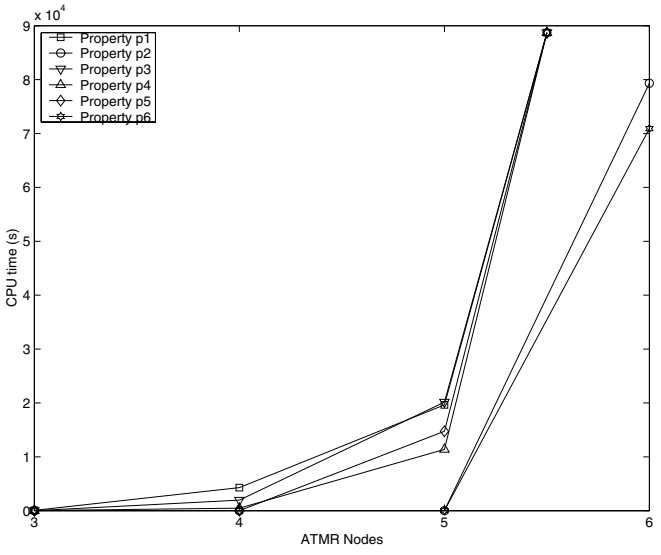

Fig. 10. VIS verification CPU time

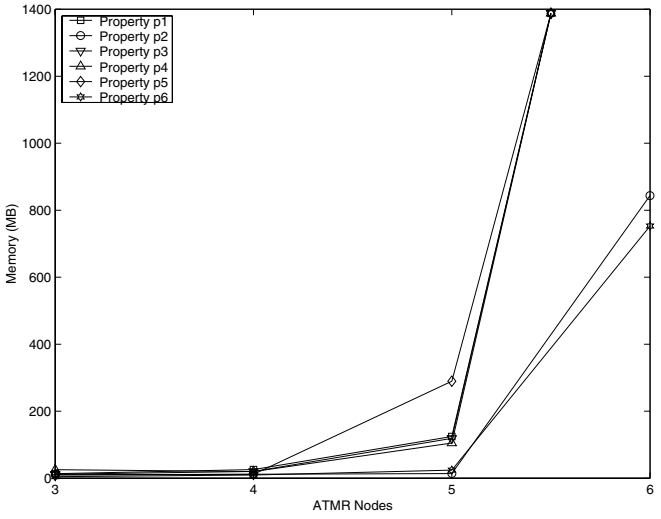

Fig. 11. VIS verification memory usage

can only be used in a fairly simple situation and cannot be applied in our case. Since the modeling language of SPIN and VIS are different, we cannot say explicitly that the two verified models, the PROMELA one and the Verilog one, are exactly the same with respect to their semantics. However, what we did is trying to follow the modeling methods and coding styles of Verilog and PROMELA, respectively. We also tried to keep these two models to their minimum size in either tool, in order to be able to compare the efficiency of SPIN and VIS in the verification of interleaving and concurrent models, respectively.

The VIS verification approach is not directly scalable to large designs due to state space explosion. From the verification results, we see that in the 3-, 4-, and 5-node model, the verification can be finished. However, the state space blows up quickly with respect to the model

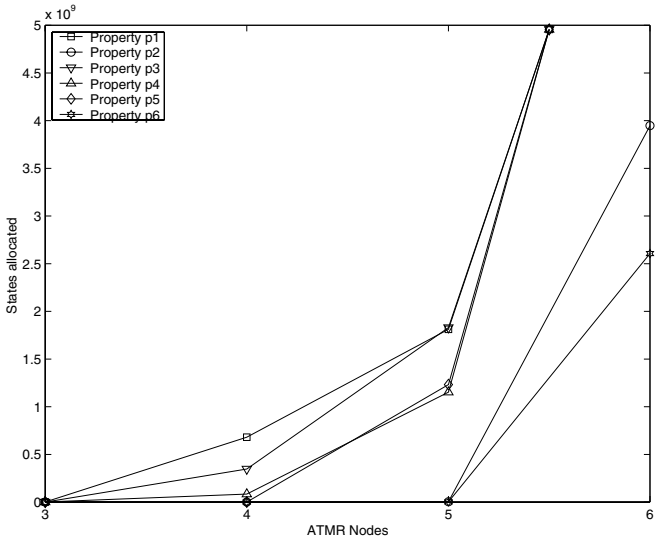

Fig. 12. VIS verification state space

size. In the verification of the 6-node model, only Property 2 can be finished. The other properties fail short of memory. There are two reasons for the state space explosion. One is the introduction of the channel module which is composed of 19 latches. The other is the circular dependency of the nodes in the ring. To solve this problem, we believe that the data complexity must be decreased by more efficient abstraction and reduction techniques. Finally, for small models (less than 6 node), we found out that the memory usage in VIS is more efficient than SPIN since VIS can finish an exhaustive search.

5 Conclusions

In this paper, we formally verified the asynchronous 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 current model 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.

Because of the inherent weakness of model checking, both SPIN and VIS are not directly scalable to large designs due to the 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. Comparing the two sets of verification results, we can find generally the verification in SPIN is faster. For example, in the 3-node model verification, the verifications of Properties 1, 3, 4, and 6 in SPIN are faster than those in VIS. Although the approximate technique used in the SPIN verification may contribute to this difference, we do not think this is the major factor because the SPIN coverages of the 3-node model properties are greater than 99.9%. From this point of view, it is a disadvantage that VIS does not provide an easy-to-use approximate technique. In SPIN, one possible way to reduce the interleavings is to make the statements atomic if these statements can be synchronous. Experiments showed that in this way the state space can be reduced for at least one order of magnitude.

SPIN uses explicit state enumeration while VIS uses implicit state enumeration (symbolic model checking). Generally, VIS can use the memory more efficiently. From our experiments, we found that VIS can finish the exhaustive search in the 3-, 4-, and 5-node models. The on-the-fly approach in SPIN does not show advantages because the model is large and the properties are global.

Since both SPIN and VIS are not scalable to large designs, model reduction techniques are very important for verification. Both tools provide model reduction options, namely partial order and cone of influence, respectively. Partial order reduction can only be used in the interleaving model and is not feasible in a tool like VIS. The model reduction techniques in VIS are limited and need a lot of human interaction.

Another weakness in VIS is that it cannot directly report deadlocks, livelock, 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 property is difficult to specify in CTL. In SPIN, a deadlock can be easily found using a timeout statement.

Finally, two practical features of these tools are worth mentioning. Namely, while VIS has a Verilog front-end allowing industrial designs to be imported and verified, SPIN comes with a graphic user interface which greatly eases the use of the tool compared to VIS.

A summary of the main comparison mentioned above and throughout the paper is given in Table 3.

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

Table 3. Comparison of SPIN and VIS

Feature	SPIN	VIS
Target system	Software	Hardware
Basic model	Interleaving model	Synchronous model
Property language	LTL	CTL
Specification 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
Graphic User Interface	Yes	No

References

1. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods Syst Design 15(1):7–48, 1999
2. Brayton, R.K., et al.: VIS: a system for verification and synthesis. In: Proc. Computer Aided Verification, Lecture Notes in Computer Science, vol. 1102. Springer, Berlin Heidelberg New York, 1996, pp. 428–432
3. Brinksma, E., Mader, A.: Verification and optimization of a PLC control schedule. In: Proc. the 7th SPIN Workshop, September 2000, Stanford University, California, USA, pp. 73–92
4. Charpentier, M., Padiou, G.: Specification and verification of the ATMR protocol using UNITY. In: Proc. International Workshop on Formal Methods for Parallel Programming, April 1997, University of Geneva, Switzerland, pp. 26–36
5. Cheng, S.T., Brayton, R.K., York, G., Yelick, K.A., Saldanha, A.: Compiling verilog into timed finite state machines. In: Proc. International Verilog Conference, 1995
6. Clarke, E.M., Grumberg, O., Long, D.: Verification tools for finite-state concurrent systems. In: REX School/symposium on a decade of concurrency: reflections and perspectives, June 1993, Noordwijkerhout, The Netherlands, pp. 124–175
7. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT, Boston, MAss., USA, 2000
8. Emerson, E.A.: Temporal and modal logic: handbook of theoretical computer science. Elsevier Sciences/North-Holland, Holland, 1990
9. Holzmann, G.J.: Design and validation of computer protocols. Prentice-Hall, Reading, Mass., USA, 1991
10. Holzmann, G.J.: The engineering of a model checker: the Gnu i-protocol case study revisited. In: Proc. Spin Workshop, September 1999, Toulouse, France, pp. 233–244
11. Holzmann, G.J., Peled, D.: An improvement in formal verification. In: Proc. International Conference on Formal Description Techniques for Distributed Systems and Communications Protocols, September 1994, Bern, Switzerland, pp. 177–194
12. ISO.: Specification of the Asynchronous Transfer Mode Ring (ATMR) Protocol, 2.0 edition, January 1993
13. McMillan, K.L.: Symbolic model checking. Kluwer, Boston, Mass., USA, 1993

Appendix A: PROMELA model of the ATMR

```
proctype AccessUnit(byte ID; chan in, out){
byte State=Idle;
MSDU_struct data;
xr in;
xs out;
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);
       ::timeout->
         data.Busy_Add=ID;
         out!ResetCell(data);NumofRst++;
       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
}
```

Appendix B: Verilog model of the ATMR

```
module mac_ctrl(clk, req, ack,  out_cell,
in_cell, id);
   input clk;
   input [0:7] in_cell;
   input       req;
   input [0:2] id;
   output [0:7] out_cell;
   output  ack;
reg [0:7] out_cell;
mac_state reg [0:1] state;
mac_celltype reg [0:1]  in_celltype,
out_celltype;
reg [0:2] crdt,out_BA,out_DA;
reg ack;
initial begin
in_celltype=Empty;out_celltype=Empty;
out_DA=0;out_BA=0;
ack=0;state=IDLE;crdt=6;//MaxCrdt;
out_cell[1:0]=1;
out_cell[4:2]=id;
out_cell[7:5]=0;
end
always @( clk or incell) begin
out_BA = in_cell[4:2];
out_DA = in_cell[7:5];
case(state)
    IDLE:
       if (req == 1)
          if (crdt > 0)
             state=SEND;
          else
             state=WAITCRDT;
       else
       begin
          case (in_celltype)
             Data: begin
                if (in_cell[7:5] == id)
                begin
                   out_DA=0;
                   out_celltype=Empty;
                end
             end
            Reset: begin
               crdt = 6;//MaxCrdt;
            end
            Empty: begin
               if (in_cell[4:2] == id)
               begin
                  out_celltype=Reset;
                  state=RESET;
               end
            end//empty
         endcase
      end
   SEND:
      case (in_celltype)
         Empty: begin
            out_celltype=Data;
            out_BA=id;
            out_DA=(id+1);
```

```
      if (out_DA >4)
      begin
         out_DA = 0;
      end
      crdt=crdt-1;
      ack=1;
      state=IDLE;
      end
      Data: begin
         out_BA=id;
         if (in_cell[7:5] == id)
         begin
            out_BA=id;
            out_DA=0;
            out_celltype=Empty;
         end
      end
      Reset: begin
         crdt = 6;//MaxCrdt;
         state=IDLE;
      end
    endcase
RESET:
   case (in_celltype)
      Data: begin
         if (in_cell[7:5] == id)
         begin
            out_DA=0;
            out_celltype=Empty;
         end
      end
      Reset: begin
         if (in_cell[4:2] == id)
         begin
            crdt=6;//MaxCrdt;
            out_BA=id;
            out_DA=0;
            out_celltype=Empty;
            state=IDLE;
         end
      end
      Empty: ;
   endcase
WAITCRDT:
   case (in_celltype)
      Data: begin
         if (in_cell[7:5] == id)
         begin
            out_DA=0;
            out_celltype=Empty;
         end
      end
      Reset: begin
         crdt=6;//MaxCrdt;
         state=IDLE;
      end
      Empty: begin
         if (in_cell[4:2] == id)
         begin
            out_celltype=Reset;
            state=RESET;
         end
      end
   endcase
endcase
case (out_celltype)
   Data: out_cell[1:0]=0;
   Empty: out_cell[1:0]=1;
   Reset: out_cell[1:0]=2;
endcase
out_cell[4:2]=out_BA;
out_cell[7:5]=out_DA;
end//always
endmodule
```