Compositional Verification of a Switch Fabric from Nortel Networks

Hong Peng, Sofiene Tahar and Yassine Mokhtari

Dept. of Electrical & Computer Engineering,
Concordia University
1455 de Maisonneuve W., Montreal, Quebec, H3G 1M8 Canada
pengh,tahar,mokhtari@ece.concordia.ca

With the development of ASIC designs, simulation cannot cover all the corner cases in a complicated design. Model checking is a fully automatic approach to verify a finite state machine against its temporal specifications. However, its application is limited by the size of the system to be verified. Compositional verification and model reduction are two possible methods to tackle this problem. In this paper, we propose a verification framework based on assume-guarantee compositional model checking, where we can apply model checking to do exhaustive verification at the module level and conduct global properties via compositional reasoning. In this framework, temporal specifications are synthesized into Verilog modules. In case a module under verification is beyond the capability of model checking, the proposed model reduction algorithm is used. We implemented the framework on top of the VIS tool and applied it on an ATM switch fabric from Nortel Networks.

1. Introduction

With the development of ASIC designs, simulation cannot cover all the corner cases in a complicated design. Model checking [6] is a fully automatic approach to verify a finite state machine against its temporal specifications. However, its application is limited by the size of the system to be verified. Current model checking tools [2,13,3] are limited to several hundred Boolean state variables due to state space explosion. There are two main methods to tackle this problem: *compositional verification* and *model reduction*. Compositional verification is to verify each partition in the system separately and then derive the system specification from the partial proofs. Model reduction is to reduce the size of the system such that it can be handled by a verification tool. One active research area is on how to introduce model checking into the verification flow of a complicated design.

In this paper, we propose a framework to perform model checking by integrating compositional reasoning and model reduction. To illustrate our approach, we used a Nortel ATM (Asynchronous Transfer Mode) switch fabric as a real case study. Using this framework, we succeed to verify the switch fabric whose size is beyond the capability of current model checking tools. Our main contributions in this paper are to integrate two novel techniques: environment (stimulus) synthesis [14] and syntactic

model reduction [17] into the framework, and make the verification by conducting global properties from module level local properties [18].

In the compositional verification [18], properties are only true under certain environments. One of the problems in the compositional reasoning approach is to generate the *environment assumption*, i.e., stimulus for the module (partition) under verification. In our approach, we provide the environment assumptions as temporal logic formulas in **ACTL** [7] and then synthesize the formulas into Verilog modules [14]. We then compose this environment module with the RTL block under verification and feed it into a model-checking tool (here VIS [2]). However, in case the size of the composed module is still beyond the capability of model checking, we use a new syntactic model reduction algorithm based on cone of influence reduction and which analyzes the (Verilog) source code and removes the redundant variables and values [17].

The rest of paper is structured as follows. Section 2 introduces the verification flow we adopt. Section 3 describes the compositional verification and the environment synthesis. Section 4 presents the model reduction method. Section 5 introduces the ATM Switch Fabric case study and discusses its modeling and verification. In Section 5, we compare the experimental results obtained using our framework with those using the FormalCheck [3] tool. Finally Section 6 concludes the paper.

2. Verification Flow

Traditionally, outgoing from the requirement specification of a product, a design group starts to implement the RTL design, while verification groups develop a behavioral model and a test suite by using either HDLs such as Verilog and VHDL or HVLs such C, e, and OpenVera. The test suite endeavors to cover all test cases. The behavioral model is written at a higher level and cannot be synthesized, but only simulated, which can be developed much quicker than the RTL model. Test benches generate test vectors for both behavior and RTL models, and thus after simulation, their outputs can be compared. The test benches are tested using the behavioral model. Because of the increasing complexity of modern ASIC chips, this verification methodology cannot discover all the bugs and takes too long. Moreover, the behavioral model itself can be bug-proned.

As a complementary approach to simulation, formal methods, in particular model checking, have proven to be very useful in design verification coverage. However, the size of the blocks that can be actually verified is very limited. In this paper, we propose a model-checking framework based on an assume-guarantee [19] compositional reasoning and model reduction. In this framework, temporal specifications are synthesized into Verilog modules acting as “test benches” in module level model checking [14], and then module level local properties are composed into global properties by using compositional reasoning [18]. In case the module under model-checking is beyond the capability of model checking, syntactic model reduction is used [17]. The proposed verification flow is illustrated in Figure 1

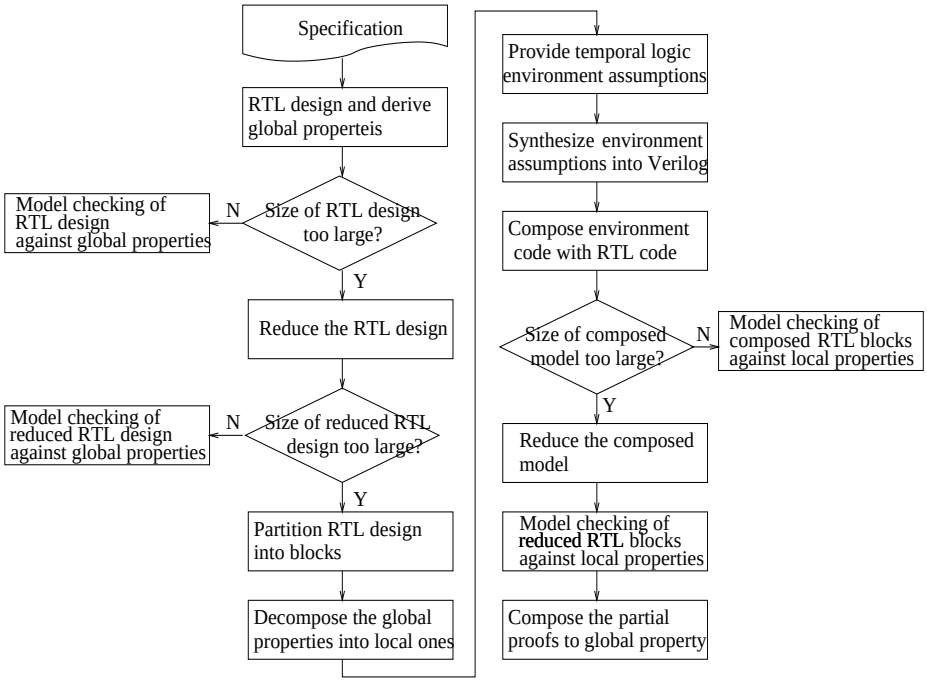

Figure 1 Compositional Verificatioin Frameworks

1. Given an RTL design and global properties derived from the specification. If the size of the RTL design, even after the reduction, is beyond the capability of model checking, then we will do the following compositional verification steps.
2. Partition the RTL design into modules.
3. Obtain local properties with respect to each RTL module (this is derived from the RTL design and the global property).
4. Derive the environment assumptions (stimulus in temporal logic **ACTL** formulas) with respect to each RTL module, and then synthesize the formulas into Verilog environment modules as illustrated in [14]. Later, compose the RTL block and the Verilog environment module.
5. In case the size of the composed code is beyond the capability of the model-checking tool, apply the syntactic model reduction in [17] with respect to the local properties, and get the reduced composed model.
6. Verify the reduced composed model against the corresponding local properties using model checking, respectively.
7. Deduce the satisfaction of the global properties on the RTL design from these local properties using compositional reasoning rules illustrated in [18].

For our framework, we have chosen the model checker VIS [2] as our evaluation tool because it provides neither compositional reasoning nor model reduction options. Furthermore, VIS has a Verilog front-end such that we can feed our design into the tool directly. Throughout the compositional verification, the global properties are

correct if and only if all the local properties are correct. For now, in terms of verification, partitioning the RTL design, deriving environment assumption formulas and local properties have to be done manually. Once we have the local properties and the corresponding environment assumptions, the following verification steps, i.e., the environment synthesis, the syntactic model reduction, and model checking, then are executed *automatically*. Another advantage of this framework is that the compositional reasoning allows us to do design verification at the system level even before the RTL modules are implemented since we can replace the missing modules by their temporal assumptions. Moreover, module verification facilitates debugging more than chip level verification does. We have applied the above verification flow on a 4*4 ATM switch fabric from Nortel Networks [20].

3. Compositional Verification and Environment Synthesis

Compositional verification has been proposed for some time as an efficient way to address the state space explosion problem in model checking. Given P and Q two modules (partitions) of a system under verification, and φ a system property to be verified, a classical compositional reasoning can be illustrated as follows [7]

$$\frac{P \models \varphi_p \qquad (\varphi_p, Q) \models \varphi}{P \,||\, Q \models \varphi}$$

where $P||Q$ means the parallel composition of module P and Q; $P|=\varphi_p$ means that the module P satisfies the **ACTL** specification φ_p; $(\varphi_p,Q)|= \varphi$ means model Q satisfies formula φ under the environment given by φ_p. In our approach, we propose to replace $(\varphi_p,Q)|= \varphi)$ by the composition of the synthesized Verilog module of the *tableau* of φ_p and module Q, where a tableau is a Kripke structure to represent φ_p. The composed system then can be fed into a model checker like VIS.

The environment synthesis is implemented using a *tableau construction* approach. Given a formula φ, the tableau construction of φ builds a *Kripke structure* (state transition graph) K consisting of states labeled by atomic propositions derived from φ and transitions between states, such that every model of φ is represented as an infinite path in K.

As is often the case with tableaus for temporal logics, e.g., [7,12], a state of the tableau consists of a set of formulas that are supposed to hold along all paths leaving the state. We propose therefore to define a reduced tableau of **ACTL** formulas consisting of less states and transitions but accepting precisely the models of the formulas. Here, the formulas in the states are interpreted over a formula or its negation, or none of them. If the latter occurs, it reflects a don't care situation, and we call this state a *dummy state*.

In [6], E. M. Clarke et al. proposed the method of constructing concurrent programs from **CTL** formulas. The result program covers one, but not all, behavior of the formulas. A. Arora et al. [1] used the same approach for real-time applications. In [11,7], D. Long et al. proposed a tableau construction approach to connect the

simulation relation and the satisfaction of an **ACTL** formula. However their tableau size is exponential to the size of the formula. In [16], C. S. Pasareanu et al. proposed an environment synthesis approach for LTL formulas in the context of software model checking using the same tableau construction approach as that in SPIN [8].

Our work distinguishes itself from the above through the following facts: (1) We are constructing the tableau for the full range of **ACTL** formulas; (2) We obtain a smaller tableau by interpreting states over a three-valued domain; (3) We apply rewriting rules to reduce the tableau size further more; (4) We describe the fairness constraint by generalized Buchi conditions; (5) We synthesize the tableau into Verilog code. In [14], we have proved the following theorem:

*Given a simulation relation $\preceq$ and an **ACTL** formula φ, for every structure K'_{φ}, $K'_{\varphi} \models \varphi$ iff $K'_{\varphi} \preceq K_{\varphi}$, where K_{φ} is the reduced tableau of φ.*

Based on the above ideas, we implemented in Java a tableau construction and Verilog synthesis for the model checker VIS [2]. We hence support here the Verilog subset of VIS.

An overview of the proposed approach is depicted in Figure 2, where "Rewriting formulas" is a pre-processor to remove the redundancy in the input **ACTL** formulas [14].

Figure 2 Reduced tableau construction and Verilog synthesis

4. Syntactic Model Reduction

Beyond compositional approaches, model reduction is the most important technique for solving the state explosion problem. Model reduction is a general approach [5,4], which allows to reduce a concrete system (M) under verification to a more abstract and smaller one (M'). Both systems M and M' are connected by an abstraction relation which is *safe* with respect to a given property φ, namely it preserves the property. This means if the property holds for the abstract system, it holds for the concrete one as well. More formally, the property φ is either *weakly* preserved if $M' \models \varphi \Rightarrow M \models \varphi$, or *strongly* preserved if $M' \models \varphi \equiv M \models \varphi$. It should be intuitively clear that the more weakly the property is preserved, the more reduction can be achieved.

One popular abstraction technique is the *cone of influence reduction* (COI) [10]. This method decreases the size of the concrete system by focusing on the variables of the concrete system that are referred to in the property and eliminating variables that do not influence the variables of interest in the property. In this way, the property satisfaction is preserved, while the size of the model that needs to be verified is smaller. However, sometimes, there are still lots of redundant information in the COI reduced model. We can easily find a case in practice where a variable A depends on variable B, but the value of variable B does not affect the value of variable A. For

example, a two-input AND gate, if one of the inputs is set to zero, then no matter what value the other input takes, the output of the gate is always at zero.

Based on the above observation, we give a refined dependency definition by examining the values of the variables that influence the truth of the property. In this approach, a system under verification is considered as a program, which syntactic and semantic structure will be analyzed. Throughout the analysis, the value domains of the state variables are extracted based on the *control flow diagram* (CFD), and the values of state variables in the program are partitioned into *active values*, and *deactive values* according to their dependency in the property. The deactive values then can be replaced by a typical *abstract* value, and thus the value domains of the variables are much smaller than the original ones. Accordingly, we can have a reduced program with respect to the abstracted variables. After the above procedures, the state space of the reduced program is smaller than that of the original one, while the correctness of the properties are preserved. In [17], we have proved the following theorem.

There is a simulation relation between the models K_P *and* $K_{P^\wedge}$ *where P and P^ are the concrete model and the reduced model, respectively. Namely* $K_P \leq K_{P^\wedge}$

In [5], abstract interpretation is a classical static program analysis approach. It has been used intensively in formal verification and model reduction [4,9]. Our proposed approach distinguishes itself from the above through the fact that the abstract domain of a variable is generated throughout the analysis of the program, which makes the reduction automatic. In [21], K. Yorav proposed ways to use the high level description (program text) of a system in order to improve the model checking process by reduction. The approaches are based on program static analysis, and analyze the control flow graph of a program to reveal runtime information of the program, without actually running it. This approach reduces the state space by analyzing the path between breakpoints where a breakpoint is a state that influence the specification. Hence, the states between these breakpoints are removed. In a similar way, we identify the breakpoints but our approach is focused on the dependency between values that influence the specification. In [15], K. S. Namjoshi et. al. proposed a reduction approach which translates a variable with large value domain, for example an integer, into a set of predicates. These predicates are determined by the automated syntactic analysis of the program under verification. Our reduction is different from this approach since we work on the finite domain, and will not generate predicates but abstract domains. Moreover, we keep only one value in the abstract domain. Our approach is also related to other works on cone-of-influence reduction [10]. However, our method is more efficient because we analyze the dependency between the *values* of variables in addition to the dependency between variables, thus the dependency relation is more accurate.

5. Case Study: Nortel ATM Switch Fabric

The basic purpose of an ATM (Asynchronous Transfer Mode) switch fabric is to transport valid (i.e., uncorrupted) ATM cells arriving at its ingress ports to the

designated egress ports as shown in Figure 3where cell 6 is a corrupted cell. Invalid ATM cells are to be discarded. Besides valid and invalid ATM cells, ATM cell streams may also contain idle cells, which serve to adapt the cell streams to the transmission bit rates employed. Cell type identification and cell switching is based on the contents of ATM cells. More precisely, an ATM cell is a fixed-length cell consisting of a 5 octet header field and a 48 octet payload field. The payload field is available for actual user information. The header field carries the information for identification and transportation of the cell. The header of an ATM cell is further decomposed into subfields as illustrated in Figure 4.

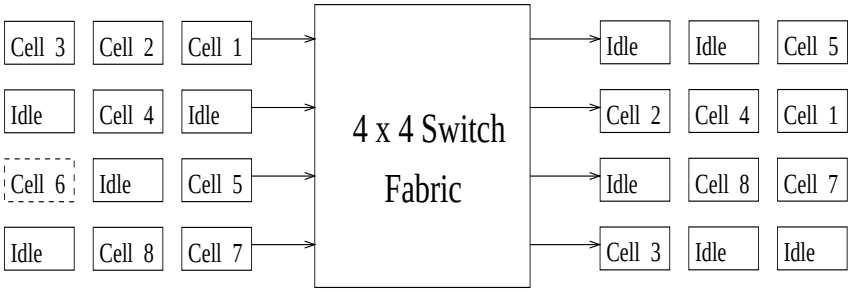

Figure 3 ATM switch fabric

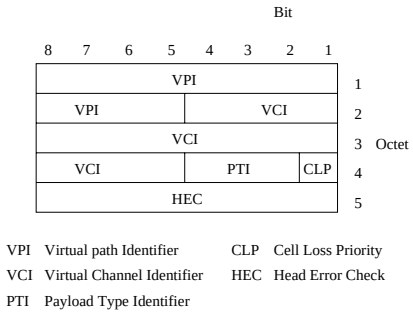

Figure 4 ATM cell header

The virtual path identifier (VPI) and the virtual channel identifier (VCI) together constitute the routing fields of the cell head. The payload type identifier (PTI) and cell loss priority (CLP) fields are not used explicitly for cell switching purposes. The last octet of the cell header contains the header error check (HEC) sequence used to check the integrity of the other header subfields. ATM cell switching can now be described in brief as follows. After receiving a cell at one of its ingress ports, an ATM switch fabric determines whether the cell is a corrupted or idle cell. A corrupted cell is a cell with an incorrect HEC sequence. An idle cell is a cell with its VPI, VCI and PTI bits all set to 0 and its CLP bit set to 1, and with a correct HEC sequence. If the ingress ATM cell is not corrupt or idle, an attempt is made to translate the value of the VPI/VCI field into a new VPI/VCI value and an egress port number by means of a VPI/VCI routing table. If the routing table contains an enabled entry for the VPI/VCI value of the ingress cell, this value is replaced by the new VPI/VCI value and a new

correct HEC sequence is generated. The resulting cell (i.e., with the new VPI/VCI value and HEC sequence) is then placed in the cell queue and switched onto the designated egress port.

5.1. Modeling the Switch Fabric

There are mainly four modules in the Nortel ATM switch fabric at hand, *ATM_SWITCH*, *ATM_MON*, *FIFO_QUEUE*, and *ATM_GEN* as shown in Figure 5. *ATM_SWITCH* is the root module, which includes the ATM cell routing functions. *ATM_MON* is the ingress part of the fabric, which includes the ATM cell monitor and detection functions. *FIFO_QUEUE* is the queuing module. *ATM_GEN* is the egress part of the fabric, which includes the ATM cell restructure functions.

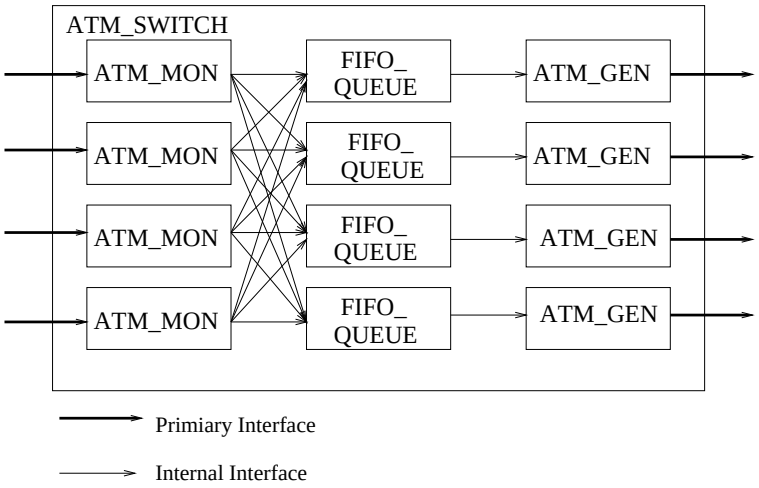

Figure 5 Nortel ATM switch fabric structure

The major property of such an ATM switch fabric is that “Valid cells (with good HEC and matching VPI/VCI) are switched correctly”. Trying to prove this property directly using model checking will fail because of state space explosion, even after model reduction. In order to prove this property, compositional verification is necessary. Here, since all the cells are queued in the *FIFO_QUEUE* module, we specify the ingress part and the egress part separately and extract the local properties respectively. Namely, in the ingress part, valid cells (with good HEC and matching VPI/VCI) are switched into the queue, and in the egress part, cells in the queue are restructured and sent.

In order to verify the ingress part, we decompose the ingress part as shown in Figure 6

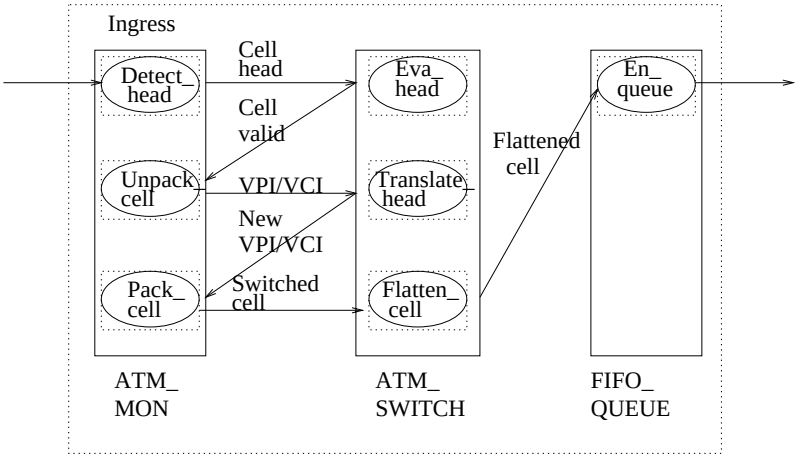

Figure 6 The ingress part

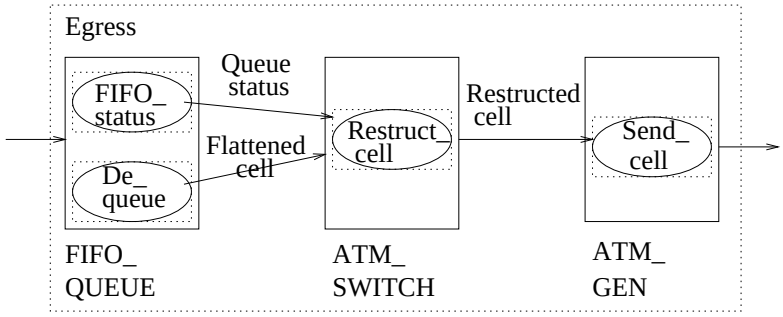

Figure 7 The egress part

where we can see that the system is partitioned into some blocks, namely *Detect_head*, *Unpack_cell*, *Pack_cell*, and so on. Hence, we can check the local properties of these blocks to derive the global property. For example, in order to check block *Translate_head,* we put the local property as

$$Ingress_{\varphi}: \mathbf{AF}\ (((VPI\_VCI\_IN[27:4] = 0)\ \ AND\ \ (MATCH\_FOUND = 1)))$$

where *VPI_VCI_IN* is the VPI/VCI of incoming cells. The incoming cell can find a match VPI/VCI (*MATCH_FOUND = 1*) when *VPI_VCI_IN[27:4] = 0*. In order to verify the egress part, we partition it as shown in Figure 7. For example, in order to check block *Restruct_cell*, we put the local property as

$$Ingress_{\psi}: \mathbf{AG}\ ((RESTRUCTED\_CELL[0] = FLATTENED\_CELL[7:0])\ AND\ (RESTRUCTED\_CELL[53] = FLATTENED\_CELL\ [423:416]))$$

where *FLATTENED_CELL* is the cell from the queue and *RESTRUCTED_CELL* is the restructed cell.The detailed properties of the blocks in the ingress and egress parts are in **Appendix B**.

5.2. Verification of the Switch Fabric

We need to verify that the blocks in the ingress part, i.e., *Detect_head*, *Eva_head*, *Translate_head*, etc., and the blocks in the egress part, i.e., *FIFO_status*, *De_queue*, etc., satisfy their local properties given a cell coming in. Here in this section, we only show how to prove a sample local property Ingress_{φ}. The other properties can be proved in a similar way.

In the verification of Ingress_{φ}, what we want to check is that the correct *VPI_VCI* of the incoming cell can find a match in the routing table, while the corrupted *VPI_VCI* of the incoming cell cannot find a match. Hence, the environment assumption is the value of the *VPI_VCI* of incoming cell, i.e., *VPI_VCI_IN*. Since in the switch fabric, only those *VPI_VCI_IN* with bit 27 to 4 being 0 can find a match, the corresponding environment **ACTL** formula is:

$$\boldsymbol{AF}\ (VPI\_VCI\_IN\ [27{:}4] = 0)$$

This assumption is discharged if the blocks before "Translate_head" can be proved. We construct the reduced tableau of this assumption shown in Figure 8, where "p" menas "*(VPI_VCI_IN [27:4] = 0)*" and "0" mean Buchi states. The states with double circles are initial states and the state without prepositional label (p or ~p) means that "p" can be either true or false in this state. As we proved in [14], this tableau contains less states than a normal tableau, but covers every possible model of the formula.

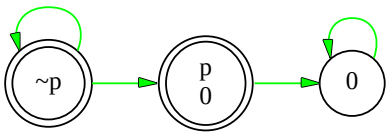

Figure 8 Reduced tableau of the assumption

This above reduced tableau then can be synthesized into Verilog behavior code (see **Appendix A**). This code then can be composed with the block under verification, i.e., *Translate_head*. However, since the routing table is involved in the verification, and the size of the routing table is 1024*58-bit, no model checking tool can accept such a large model. We have to apply syntactic model reduction [17] with respect to the properties.

In order to make the model reduction, we construct the control flow diagram of module *Translate_head* as shown as follows.

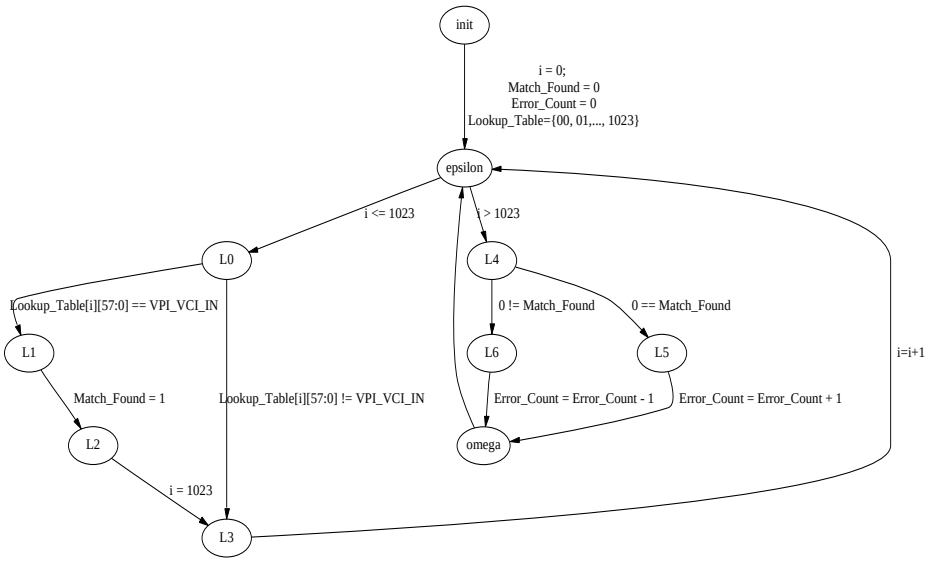

Figure 9 Control flow diagram of Translate_head

By observing property $Ingress_{\varphi}$, we find that we are just verifying the behavior of variable *MATCH_FOUND*. The value of *MATCH_FOUND* is changed in node "L2" in the above diagram, which we call "key node". According to the model reduction approach proposed in [17], we traverse the diagram and find those values that do not affect *MATCH_FOUND*, namely those values from which node "L2" is not reachable. Then those values can be abstracted using one typical value. In the diagram, only the first item in the routing table with bit 27 to 4 equaling to 0 can change the value of *MATCH_FOUND*, so this value is kept as *active values*, while all other values in the routing table, which do not affect the behavior of *MATCH_FOUND* can be removed. So, we can keep only two items in the routing table and remove the other 1022 items. In this way, the model under verification is reduced. Then we can compose the reduced model and its environment, and check it against the local property using VIS.

The verification results of sample properties are shown in the Table 1, where the CPU time reported is the real time; the BDD size in the table represents those states of the system that satisfy the formula.

Properties	Status	Model Checking		
		CPU(S)	Memory(MB)	BDD nodes
$Ingress\_P_1$	Verified	19.5	0.908	42722
$Ingress\_P_2$	Verified	272.4	1.908	18446
$Ingress\_P_3$	Verified	1.4	1.308	15073
$Ingress\_P_4$	Verified	3.8	9.9	7130
$Ingress\_P_5$	Verified	11.3	8.54	164033
$Ingress\_P_6$	Verified	11.6	8.54	383969
$Ingress\_P_7$	Verified	3.7	9.918	7104

Ingress_P8	Failed	-	-	-
Ingress_P9	Verified	15.1	100.6	490923
Egress_P1	Verified	2.5	1.44	15764
Egress_P2	Verified	16.5	112.3	632434
Egress_P3	Failed	-	-	-
Egress_P4	Verified	6.7	12.2	137724

Table 1 Verification Results of Sample Properties in VIS

The verification is performed using the VIS model checker on a SUN Enterprise server with 6GB memory. Through out the model checking, we set VIS with the options: implicit clocking and advanced ordering. In the Table, "-"means that VIS does not accept the model because of VIS internal bugs. In this case, we conducted the particular property verification in another tool (here FormalCheck) to make sure that it is really sound. Also, for the purpose of comparison, we verified the same models in FormalCheck on the same machine. However, this time, we do not do the reduction using our model reduction approach. The verification results are shown in Table 2. The reduction algorithm selected in FormalCheck is iterated with empty reduction seed because there are no constraints on the primary inputs, and the run option is symbolic BDD because it allows a more efficient model checking. The CPU time in the table is the real time and "States" are the states reachable.

Properties	Status	CPU(S)	Memory(MB)	States
Ingress_P1	Failed	-	-	-
Ingress_P2	Verified	1036	29.64	2.02e+03
Ingress_P3	Verified	4	3.121	4
Ingress_P4	Verified	22	6.71	1.02e+03
Ingress_P5	Non-terminated	-	-	-
Ingress_P6	Non-terminated	-	-	-
Ingress_P7	Verified	32	13.75	3.36e+07
Ingress_P8	Verified	8	3.69	1.31e+05
Ingress_P9	Non-terminated	-	-	-
Egress_P1	Verified	365	0.55	2.62e+05
Egress_P2	Non-terminated	-	-	
Egress_P3	Verified	605	115.07	6.67e+02
Egress_P4	Failed	-	-	-

Table 2 Verification Results of Sample Properties in FormalCheck

In the Table, "*Non-terminated"* means that the verification failed due to state space explosion. The reason for this is either that the property under verification involves so many variables in the program that the reduction algorithms in FormalCheck are of no help (in this case, FormalCheck gives an internal bug report), or the model under verification is too large to be even complied by the tool (in this case, the tool will stay in a dead lock state until all the memory is consumed).

The *"Failed"* in the table means that the property cannot be verified by this tool because the environment assumptions could not be specified. We can translate the environment assumption into FormalCheck format by dropping '**A**' operator.

Overall, since the verification in VIS is based on the reduced model while the verification in FormalCheck is based on the concrete model, the former is efficient with respect to CPU time and memory because the latter has to do the reduction work by itself.

Through out the verification, we also found some bugs in the design.

For example, a statement in the *Translate_head* block

```
while (!MATCH_FOUND && i <= MAX_CONNECTIONS)
   if (LOOKUP_TABLE[i].VPI_VCI_IN == VPI_VCI_IN ) begin
     MATCH_FOUND = 1;
……
```

is mistaken as

```
while (!MATCH_FOUND && i <= MAX_CONNECTIONS)
   if  (LOOKUP_TABLE[i].VPI_VCI_IN  ==  VPI_VCI_IN  &
28'hFF7FFFF) begin
      MATCH_FOUND = 1;
……
```

where *MAX_CONNECTIONS* is the number of items in the *LOOKUP_TABLE* and *28* is the length of *VPI_VCI_IN*. In this case, cells with *VPI_VCI* equaling to *008000* are matched, but should not, since according to the specification, only the cells with *VPI_VCI* equaling to *000000* can be matched. This bug actually is difficult to be found using simulation because one has to simulate that all the cells with *VPI_VCI* not equaling to *000000* cannot be matched. With formal verification, one can easily detect this bug using property *Ingress_P*$_6$. According to this property, every state in the state space should be *(VPI_VCI_IN != 000000) AND (MATCH_FOUND = 0),* provided that the incoming *VPI_VCI_IN* does not equal to *000000*. This bug is also corrected by simply removing *28'hFF7FFFF* in the while loop.

After the above verification, we actually proved that every block satisfies its local properties, given certain environment assumptions. Moreover, because these environment assumptions are the outputs of the blocks in the system, they are discharged in the verification of the local properties. We apply the compositional rule as follows. Where $T\varphi$ means the synthesized Verilog module of formula φ, Actually, the global property: "Valid cells (with good HEC and matching VPI/VCI) are switched correctly" is given by assuming P_{valid_cells} and deducing *Egress_P*$_4$ (correct switch). This way, we are checking the satisfaction of the global property against the whole design.

$$T_{Pvalid\_cell} \,||\, Detect\_head \,|\!=\, Ingress\_p1$$
$$Eva\_head \,||\, T_{Ingress\_p1} \,|\!=\, Ingress\_p2$$
$$Eva\_head \,||\, T_{Ingress\_p1} \,|\!=\, Ingress\_p3$$
$$Unpack\_cell \,||\, T_{Ingress\_p3} \,|\!=\, Ingress\_p4$$
$$Translate\_head \,||\, T_{Ingress\_p4} \,|\!=\, Ingress\_p5$$
$$Translate\_head \,||\, T_{Ingress\_p4} \,|\!=\, Ingress\_p6$$
$$Pack\_cell \,||\, T_{Ingress\_P5} \,||\, T_{Ingress\_p6} \,|\!=\, Ingress\_p7$$
$$Flattened\_cell \,||\, T_{Ingress\_P7} \,|\!=\, Ingress\_p8$$
$$En\_queue \,||\, T_{Ingress\_P8} \,|\!=\, Ingress\_P9$$
$$FIFO\_status \,|\!=\, Egress\_p1$$
$$De\_queue \,|\!=\, Egress\_p2$$
$$\mathrm{Re}\, struct\_cell \,||\, T_{Egress\_p1} \,||\, T_{Egress\_p2} \,|\!=\, Egress\_p3$$
$$Send\_cell \,|\, T_{Egress\_p3} \,|\!=\, Egress\_p4$$

$$\frac{}{T_{Pvalid\_cell} \,||\, Detect\_head \,||\, Eva\_head \,||\, Unpack\_cell \,||\, Translate\_cell \,||\, Pack\_cell \,||\, Flatten\_cell \,||\, En\_queue \,||\, FIFO\_status \,||\, De\_queue \,||\, \mathrm{Re}\_struct\_cell \,||\, Send\_cell \,|\!=\, Egress\_p4}$$

6. Conclusion

In this paper, we proposed a compositional verification framework including environment synthesis and model reduction techniques. Using this framework, we verified an ATM switch fabric from Nortel Networks, which cannot be verified by plain model checking due to state space explosion. Here, we use VIS as target model checker, however, we can still use some other alternatives, such as SMV [13]. Through out the verification, we found bugs in the design, which were not caught through simulation. Because of the advantages in the environment synthesis and the model reduction, this framework is efficient in the verification with respect to the CPU time and memory resources. The framework is implemented in Java running on SUN Solaris OS.

Reference

[1] A.Arora, P.C. Attie, and E.A. Emerson. Synthesis of fault-tolerant concurrent programs. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 173--182, Puerto Vallarta, Mexico, June 1998.

[2] R.K. Brayton et al.VIS: A system for verification and synthesis. In T.Henzinger and R.Alur, editors, *Computer-aided Verification'96*, volume 1102 of *Lecture Notes in Computer Science*, pages 428--432. Springer Verlag, Rutgers University, NY, USA, July 1996.

[3] Cadence Design Systems. *Technical manual of FormalCheck*, v2.3 edition, 1987-1999.

[4] E.M. Clarke, O.Grumberg, and D.Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, Vol.16(No. 5):1512--1542, Sept 1994.
[5] P.Cousot and R.Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238--252, Los Angeles, California, USA, 1977.
[6] E.A. Emerson and E.M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241--266, 1982.
[7] O.Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843--871, May 1994.
[8] G.J. Holzmann. *Design and validation of computer protocols*. Prentice hall, 1991.
[9] Y.Kesten and A.Pnueli. Modularization and abstraction: the key to practical formal verification. *23rd Int. Symp. Mathematical Foundations of Computer Science*, Brno, Czech Republic, 1998.
[10] R.P. Kurshan. *Computer-aided verification of coordinating processes*. Princeton University Press, 1994.
[11] D.E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, CMU, 1993.
[12] Z.Manna and A.Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety*. Springer-Verlag, New York, 1991.
[13] K.L. McMillan. *Symbolic M¦odel Checking*. Kluwer, 1993.
[14] H.Peng, Y.Mokhtari, and S.Tahar. *Environment synthesis for compositional model checking.* In *Proceeding of IEEE International Conference on Computer Design*, Freiburg, Germany, September 2002. IEEE computer society Press.
[15] K.S. Namjoshi and R.P. Kurshan. Syntactic program transformations for automatic abstraction. In E.Allen Emerson and A.Prasad Sistla, editors, *Computer-aided Verification'00*, volume 1855 of *Lecture Notes in Computer Science*, pages 433--449, Chicago, IL, USA, July 2000. Springer Verlag.
[16] C.S. Pasareanu, M.B. Dwyer, and M.Huth. Assume-guarantee model checking of software: A comparative case study. *SPIN Workshop 1999*, pages 168--183, Trento, Italy, June 1999.
[17] H.Peng, Y.Mokhtari, and S.Tahar. Model reduction based on value dependency. In *Proceeding of IEEE International ASIC/SOC Conference*, Washigton, DC, USA, September 2001.
[18] H.Peng and S.Tahar. Compositional verification of IP based designs. In *Proceedings of IFIP International Workshop on IP Based Synthesis and System Design*, Grenoble, France, December 1999.
[19] A.Pnueli. In transition for global to modular temporal reasoning about programs. In K.R. Kurshan, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI series. Series F*. Springer Verlag, 1984.
[20] Northern Telecom. *Specification of a 4*4 ATM switch*, November 1998.
[21] K.Yorav. *Exploiting syntactic structure for automatic verification*. PhD thesis, Israel institute of technology, June 2000.

A. Synthesized Environment of *Ingress_φ*

The **ACTL** environment assumption of properties *Ingress_φ* is "**AF**(VPI_VCI_IN[27:4] = 0)". The synthesized Verilog code (Verilog subset acceptable in VIS model checker) of this assumption is shown as follows. Lines 0 to 5 are comments. *VPI_VCI_IN[27:4]* is set as an output of the module *tableau*. Lines 9 to 12 are to declare the variables. Lines 14 to 18 are to set the initial states, where

S_INIT_W indicates the initial states and *S_INIT_W_TMP* is a temporary variable. In Lines 19 to 25, wire variables *Sx_NEXT_W* describe the transitions of the states, i.e., what is the next state of current state *Sx*. *Sx_NEXT_W_TMP* are the temporary variables. Lines 26 to 49 are the non-deterministic assignment of *VPI_VCI_IN[27:4]*. Lines 50 to 70 are the behaviors of this environment.

```
L0: //`define TRUE 1
L1: //`define FALSE 0
L2: //`define S0 0
L3: //`define S1 1
L4: //`define S2 2
L5: //`define S3 3
L6: module tableau(VPI_VCI_IN);
L7: output[27:4] VPI_VCI_IN;
L8: //Variable declaration
L9: reg [27:4] VPI_VCI_IN;
L10: wire [27:4] VPI_VCI_INND_W;
L11: reg [1:0] STATE;
L12: wire [1:0] S_INIT_W_TMP, S_INIT_W,
                S0_NEXT_W, S1_NEXT_W,
                S2_NEXT_W, S3_NEXT_W;
L13: //Initialiazation
L14: assign S_INIT_W_TMP = $ND(0, 1, 2, 3);//$
L15: assign S_INIT_W = ((S_INIT_W_TMP == 3)) ?
            2 : S_INIT_W_TMP;
L16: initial begin
L17:     STATE = S_INIT_W;
L18: end // Initial

L19: //Combinational part
L20: assign S2_NEXT_W = 3;
L21: assign S3_NEXT_W = 3;
L22: assign S1_NEXT_W = 1;
L23: wire [1:0] S0_NEXT_W_TMP;
L24: assign S0_NEXT_W_TMP = $ND (0,1,2,3);//$
L25: assign S0_NEXT_W = ((S0_NEXT_W_TMP == 1)
     || (S0_NEXT_W_TMP == 3)) ?
        2 : S0_NEXT_W_TMP;
L26: assign VPI_VCI_INND_W[4] = $ND( 0, 1);
```

```
L27: assign VPI_VCI_INND_W[5] = $ND( 0, 1);
L28: assign VPI_VCI_INND_W[6] = $ND( 0, 1);
L29: assign VPI_VCI_INND_W[7] = $ND( 0, 1);
L30: assign VPI_VCI_INND_W[8] = $ND( 0, 1);
L31: assign VPI_VCI_INND_W[9] = $ND( 0, 1);
L32: assign VPI_VCI_INND_W[10] = $ND( 0, 1);
L33: assign VPI_VCI_INND_W[11] = $ND( 0, 1);
L34: assign VPI_VCI_INND_W[12] = $ND( 0, 1);
L35: assign VPI_VCI_INND_W[13] = $ND( 0, 1);
L36: assign VPI_VCI_INND_W[14] = $ND( 0, 1);
L37: assign VPI_VCI_INND_W[15] = $ND( 0, 1);
L38: assign VPI_VCI_INND_W[16] = $ND( 0, 1);
L39: assign VPI_VCI_INND_W[17] = $ND( 0, 1);
L40: assign VPI_VCI_INND_W[18] = $ND( 0, 1);
L41: assign VPI_VCI_INND_W[19] = $ND( 0, 1);
L42: assign VPI_VCI_INND_W[20] = $ND( 0, 1);
L43: assign VPI_VCI_INND_W[21] = $ND( 0, 1);
L44: assign VPI_VCI_INND_W[22] = $ND( 0, 1);
L45: assign VPI_VCI_INND_W[23] = $ND( 0, 1);
L46: assign VPI_VCI_INND_W[24] = $ND( 0, 1);
L47: assign VPI_VCI_INND_W[25] = $ND( 0, 1);
L48: assign VPI_VCI_INND_W[26] = $ND( 0, 1);
L49: assign VPI_VCI_INND_W[27] = $ND( 0, 1);
L50: //Sequential part
L51: always begin
L52:    case (STATE)
L53:    0: begin

L54:        VPI_VCI_IN[27:4] = 1;
L55:        STATE = S0_NEXT_W;
L56:       end
L57:    1: begin
L58:        VPI_VCI_IN[27:4] = 1;
L59:        STATE = S1_NEXT_W;
L60:       end
L61:    2: begin
L62:        VPI_VCI_IN[27:4] = 0;
L63:        STATE = S2_NEXT_W;
```

```
L64:       end
L65:    3: begin
L66:        VPI_VCI_IN = VPI_VCI_INND_W;
L67:        STATE = S3_NEXT_W;
L68:       end
L69:    endcase // case (STATE)
L70: end // always begin
L71: endmodule // tableau
```

The fairness constraint file is shown as follows, namely one of the following states has to be asserted infinitely often.

```
 (tableau.STATE = 1
 || tableau.STATE = 2
 || tableau.STATE = 3
);
```

B. Ingress and Egress Properties

Ingress_P$_1$
In this property, we require that the ingress port will receive a cell if a cell is coming into the port.Formally,

$$\boldsymbol{AF}\ (New\_cell\_recieved = 1)$$

Where *New_cell_recieved* is set when a cell with integral structure is received.
Ingress_P$_2$
In this property, we check the HEC detection mechanism in the ingress part, given there is a cell ready. Namely,

$$\mathbf{AG}\ (\text{HEC\_OK} = 1)$$

where *HEC_OK* is set if the cell under test has a good HEC value.
Ingress_P$_3$
In this property, we check the IDLE detection mechanism in the ingress part, given there is a cell ready. Formally,

$$\boldsymbol{AG}((WORD[0]=0)\ AND\ (WORD[1]=0)\ AND\ (WORD[2]=0)\ AND\ (WORD[3][7:1]=0)\ AND\ (WORD[3][0]=1) \rightarrow (IS\_IDLE=1))$$

meaning that when the byte stream (*WORD*) in a cell satisfying the above format (all 0 except the last bit), then this cell is judged to be an idle cell.
Ingress_P$_4$
In this property, we check that a cell is unpacked correctly. Formally,

AG *((VPI[11:4] = WORD[0]) AND (VCI[11:4] = WORD[2]) AND (VPI[3:0] = WORD[1][7:4]) AND (VCI[15:12] = WORD[1][3:0]) AND (VCI[3:0] = WORD[3][7:4]) AND (PTI[2:0]=WORD[3][3:1]) AND (CLP = WORD[3][1]))*

where *WORD* is the input byte stream and *VPI*, *VCI*, *CLP*, *PTI* are the formatted cell headers.

Ingress_P$_5$

In this property, we check that if the incoming *VPI_VCI* satisfies our specification (bit 27 to 4 are 0), then it will find a match in the routing table. Formally,

AF *(((VPI_VCI_IN[27:4] = 0) AND (MATCH_FOUND = 1)))*

where *VPI_VCI_IN* is the *VPI_VCI* value of the input cell. *MATCH_FOUND* is set when *VPI_VCI_IN* can find a match in the routing table.

Ingress_P$_6$

In this property, we check that all incoming *VPI_VCI* that do not satisfy our specification cannot find a match in the routing table. Formally,

AG *((NOT(VPI_VCI_IN[27:4] = 0)) → (MATCH_FOUND = 0)))*

This is a safety property of the routing table, which has the similar form as *Ingress_P$_5$*.

Ingress_P$_7$

In this property, we check that the cell is packed correctly. Formally,

AG *((VPI[11:4] = WORD[0]) AND (VCI[11:4] = WORD[2]) AND (VPI[3:0] = WORD[1][7:4]) AND (VCI[15:12] = WORD[1][3:0]) AND (VCI[3:0] = WORD[3][7:4]) AND (PTI[2:0]=WORD[3][3:1]) AND (CLP = WORD[3][1]))*

This property is similar with *Ingress_P$_4$*.

Ingress_P$_8$

In this property, we check that the cell is flattened correctly, namely the word structure of a cell can be correctly flattened into a bit stream. Formally,

AG*((FLATTENED_CELL[7:0] = WORD[0]) AND (FLATTENED_CELL[15:8] = WORD[1]))*

where *FLATTENED_CELL* is the corresponding bit stream of the cell.

Ingress_P$_9$

In this property, we check that the flattened cell can be enqueued correctly, namely the flattened cell is put into the queue and the pointer of the queue is changed accordingly. Formally,

AG*(NOT IS_FULL →* ***AF*** *((Queue.HEAD = FLATTENED_CELL) AND (HEAD = HEAD + 1)))*

Where *IS_FULL* is set when the queue is full; the property means that if the queue is not full, then the cell will find a place in the queue.

Egress_P$_1$

In this property, we check that the status of the queue is empty if the head pointer equals to the tail pointer. Formally,

AG *((HEAD = TAIL) → (EMPTY = 1))*

where *EMPTY* is set when the queue is empty.

Egress_P$_2$

In this property, we check that the flattened bit stream cell can be restructured into a word format cell. Formally,

AG *((RESTRUCTED_CELL[0] = FLATTENED_CELL[7:0]) AND (RESTRUCTED_CELL[53] = FLATTENED_CELL [423:416]))*

meaning that the dequeued cell (*FLATTENED_CELL*) can be restructured into a formatted cell (*RESTRUCTED_CELL)*;

Egress_P_3

In this property, we check that the flattened bit stream cell can be restructured into a word format cell. Formally,

$$\textbf{\textit{AG}}\ \textit{((RESTRUCTED\_CELL[0] = FLATTENED\_CELL[7:0]) AND (RESTRUCTED\_CELL[53] = FLATTENED\_CELL [423:416]))}$$

meaning that the dequeued cell (*FLATTENED_CELL*) can be restructured into a formatted cell (*RESTRUCTED_CELL*);

Egress_P_4

In this property, we check that the de-queued cell can be sent out to the egress port. Formally,

$$\textbf{\textit{AF}}\ \textit{(NEWCELL\_READY = 1)}$$

where *NEWCELL_READY* is set when a cell has been sent out successfully.