Environment Synthesis for Compositional Model Checking

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

Abstract

Modeling the environment of a design module under verification is a known practical problem in compositional verification. In this paper, we propose an approach to translate an **ACTL** *specification into such an environment. Throughout the translation, we construct an efficient tableau for the full range of* **ACTL** *and synthesize the tableau into Verilog HDL behavior level program. The synthesized program can be used to check the properties that the system's components must guarantee. We have used the proposed environment synthesis in the compositional verification of an ATM switch fabric from Nortel Networks. Experiments show that given the theoretical compositional verification intractable limit, we can still manage to verify industry size designs.*

1. Introduction

For the last two decades, verification techniques have been applied in large system engineering projects. The widely used simulation techniques, such as testbench techniques, can handle large scale designs; nevertheless, an additional large simulation environment has to be modeled manually. In a normal case, this environment modeling consumes about 70% of the overall design efforts. As a complementary approach to simulation, formal verification, in particular model checking, have proven to be very useful in verification coverage. However, model checking suffers from the well-known *state space explosion* problem. Compositional verification is one possible way to address this problem.

Compositional verification is to verify each partition in the system separately and then derive the system specification from the partial proofs. In compositional verification, properties are only true under certain environments. One of the problems in compositional verification is to generate these *environment assumptions*. For instance, in the assume-guarantee approach (where $P||Q$ means the parallel composition of modules P and Q):

$$\frac{\begin{array}{r} P \models \varphi_P \\ (\varphi_P, Q) \models \varphi \end{array}}{P \parallel Q \models \varphi}$$

we have to prove that model Q satisfies formula φ under the environment given by φ_P ($(\varphi_P, Q) \models \varphi$). In this paper, we use **ACTL** to describe φ_P and replace (φ_P, Q) by the composition of the synthesized Verilog HDL module of *tableau* $\mathcal{T}(\varphi_P)$ of φ_P and module Q, also written in Verilog HDL. Then, the composed system of $\mathcal{T}(\varphi_P)$ and Q can be fed into a verification tool like VIS [2]. The synthesized program guarantees to cover the behavior of temporal environment specifications. Besides, experiments indicate that synthesized environments enable faster model checking than temporal environment assumptions since generally temporal assumptions lead to larger state spaces to be searched [9].

The complexities for different assume-guarantee reasoning styles with **LTL** (Linear Time Logic) or **CTL** (Computational Tree Logic) are explored in [6] [13]. Generally speaking, using **LTL** and **CTL** as assumptions are both computation hard. The selection is a trade-off depending on the applications. In our approach, we use **ACTL** (A-Computational Tree Logic), a subset of **CTL**, to capture the assumptions because (1) the environment assumptions can be used as guarantees in the next verification stage so that hierarchical verification is possible, since both assumptions and guarantees are in **ACTL**, (2) the expressiveness of **ACTL** is different from **LTL**, as in the case of the well known expression **AFAG**p.

The primary contributions of this paper are (1) implementing an assume-guarantee approach where **ACTL** is used for both the assumption and the guarantee, (2) defining and implementing a tool to synthesize the environment based on an efficient **ACTL** tableau, and (3) applying the approach on an industry ATM switch fabric. Overall, experiments showed that even if the assume-guarantee reasoning is theoretically intractable, however, in practice, we can still manage to verify large industry scale designs using this approach.

The rest of this paper is organized as follows. Section 2 overviews and discusses related work. Section 3 focuses on the tableau construction and Verilog synthesis. Section 4 describes a case study on an ATM switch fabric. Section 5 finally concludes the paper.

2. Related Work

The theoretical analysis of the complexity of assume-guarantee reasoning is discussed in [6] and [13] where it is shown that the complexity of using **ACTL** or **LTL** as the assumptions in the compositional verification is PSPACE-complete and EXPSPACE-complete, respectively. The tableau size is a key factor affecting the verification efficiency. This motivated us to look into the construction of tableau in an optimized way.

We propose a tableau construction which can be seen as an extension of [5] by adding liveness property construction and a rewriting procedure. In [7], M. Maidl also proposes an **ACTL** tableau construction procedure in order to prove the common fragment of **ACTL** and **LTL**. Since the purpose of this tableau is not to be used in the assume-guarantee verification, so the tableau size is not optimal. In [4], D. Long *et al.* proposed a tableau construction approach to connect the simulation relation and the satisfaction of an **ACTL** formula. In [9], C. S. Pasareanu *et al.* proposed the same environment synthesis idea for **LTL** formulas in the context of software model checking using the same tableau construction approach as that in SPIN. We share the similar ideas with the above approaches [5][7][4][9], however, we target the maximum model of full range **ACTL** tableau construction and the size of our tableau is smaller.

Besides the tableau construction, we developed a tool which synthesizes Verilog HDL code from the obtained tableau. The synthesis of the tableau into Verilog is easy but not trivial since it depends on what model checking tool is used and what Verilog subset is supported by the tool. There are many similar synthesis approaches available in the open literature, such as [1], which are all application dependent. In our synthesis, we take the advantages of the VIS-Verilog subset which for instance supports non-deterministic transitions to construct simple but efficient programs.

3. Tableau Construction and Synthesis

The tableau construction aims (if possible) to construct the maximal model of an **ACTL**[1] formula φ. Namely, given a formula φ, the tableau construction of φ builds a *Kripke structure* (state transition graph) $\mathcal{K}$ 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 $\mathcal{K}$. We implemented the proposed approach, as depicted in Figure 1.

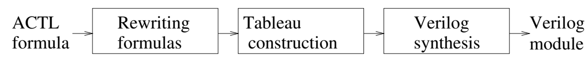

Figure 1. Tableau construction and Verilog synthesis

Rewriting is the first step towards an efficient tableau construction. It is considered an easy and effective way to minimize the tableau [11]. Rewriting consists of a set of rules applied recursively to an **ACTL** formula in positive normal form (i.e., the negation is applied only on atomic propositions), reducing the number of temporal operators and/or connectives. Although there are no dedicated **ACTL** rewriting rules available in the open literature, we can still derive the rules based on well-known identities. One possible way is to deploy similar **LTL** rewriting rules, which can be found, e.g., in [8]. However, we cannot refer to the **LTL** rewriting rules by simply padding the path quantifier by for-all-path quantifier **A**. For example, in **ACTL**, $\mathbf{AGAF}p \vee \mathbf{AGAF}q$ is different from $\mathbf{AGAF}(p \vee q)$, while this is not the case in **LTL**. The application of a rewriting rule consists of replacing in the (sub)formula the left hand side with the right hand side which has in principle less temporal operators and/or connectives.

A set of sample rewriting rules that we have proved sound and implemented in our tool is given below. Our implementation also provides the ability to renew or add new rewriting rules in its internal rewriting database.

Sample Rewriting Rules
$\mathbf{AGAG}\ p \equiv \mathbf{AG}\ p$
$\mathbf{AFAF}\ p \equiv \mathbf{AF}\ p$
$p\ \mathbf{AU}\ (p\ \mathbf{AU}\ q) \equiv (p\ \mathbf{AU}\ q)$
$p\ \mathbf{AV}\ (p\ \mathbf{AV}\ q) \equiv (p\ \mathbf{AV}\ q)$
$(p\ \mathbf{AU}\ q)\ \mathbf{AU}\ q \equiv (p\ \mathbf{AU}\ q)$
$\mathbf{AX}\ p \wedge \mathbf{AX}\ q \equiv \mathbf{AX}\ (p \wedge q)$
$\mathbf{AG}\ p \wedge \mathbf{AG}\ q \equiv \mathbf{AG}\ (p \wedge q)$
$(p\ \mathbf{AU}\ r) \wedge (q\ \mathbf{AU}\ r) \equiv (p \wedge q)\ \mathbf{AU}\ r$

After the rewriting, we calculate the *cover* of a formula meaning that it will cover all the states satisfying the for-

[1] **ACTL** formulas are constructed from a set of atomic propositions $\mathcal{P}$ which represent properties of individual states, the standard boolean operators, the temporal operators **X** (Next),**U** (Until) and **V** (Unless), and the universal path quantifier **A** (for all paths). Additional temporal operators are introduced as abbreviations: $\mathbf{AG}\varphi$ is defined as $\mathbf{A}(\mathbf{falseV}\varphi)$ and $\mathbf{AF}\varphi$ is defined as $\mathbf{A}(\mathbf{trueU}\varphi)$

mula either in the current or in the next state. A cover is a set of items called *particles*. Each particle is considered as a state label in the corresponding Kripke structure of the formula. For example, COVER $(p) = \{\{p\}\}$, where p is a propositional formula in the atomic propositions (APs), meaning that in order to satisfy formula p, the current state label (particle) has to be $\{p\}$. In the calculation of the covers, we decompose formulas till there is at least one elementary formula in each particle, namely, either it is the formula without modalities, or it is the formula beginning with **AX**. The formulas without modalities need to hold in the current state, and the formulas beginning with **AX** modality illustrate what needs to hold in the next state.

During the formula decomposition, we merge two covers by the union operator "$\cup$" (see Algorithm 1). As a matter of fact, it is not always the case that the union on two particles makes sense because they are not compatible[2], for example, $\{p\} \cup \{\neg p\}$ has no semantic meaning since a state satisfying both p and $\neg p$ does not exist. In this case, we use a "JOIN" operator to exclude this case, namely, COVER_1 JOIN $\text{COVER}_2 = \{\forall_{\rho \in \text{COVER}_1, \tau \in \text{COVER}_2, \rho\ and\ \tau\ compatible} \rho \cup \tau\}$. By the calculation of cover, we analyze the formulas and find out what has to hold at the current state and what has to hold at the next state.

Algorithm 1 cover(ϕ)

if $\phi = p$ **then**
  COVER $(\phi) = \{\{p\}\}$
else if $\phi = \neg p$ **then**
  COVER $(\phi) = \{\{\neg p\}\}$
else if $\phi = \mu \wedge \nu$ **then**
  COVER (ϕ) =COVER (μ) JOIN COVER (ν)
else if $\phi = \mu \vee \nu$ **then**
  COVER (ϕ) =COVER $(\mu) \cup$ COVER (ν)
else if $\phi = \mathbf{AX}\mu$ **then**
  COVER $(\phi) = \{\{\mathbf{AX}\mu\}\}$
else if $\phi = \mathbf{A}(\mu \mathbf{U} \nu)$ **then**
  COVER $(\phi) = [\{\{\mathbf{A}(\mu\mathbf{U}\nu)\}\}$ JOIN COVER $(\{\nu\})] \cup [\{\{\mathbf{A}(\mu\mathbf{U}\nu)\}\}$ JOIN COVER $(\{\mu, \mathbf{AXA}(\mu\mathbf{U}\nu)\})]$
else if $\phi = \mathbf{A}(\mu \mathbf{V} \nu)$ **then**
  COVER $(\phi) = [\{\{\mathbf{A}(\mu\mathbf{V}\nu)\}\}$ JOIN COVER $(\{\mu, \nu\})] \cup [\{\{\mathbf{A}(\mu\mathbf{V}\nu)\}\}$ JOIN COVER $(\{\nu, \mathbf{AXA}(\mu\mathbf{V}\nu)\})]$
end if

Next, we consider the process of computing the particles that are successors of the given particle P (Algorithm 2). In case the particle has no successors, for example $P = \{\mathbf{AX}(p), \mathbf{AX}(\neg p)\}$, the particle is not *realizable* and the successor is labeled by $\emptyset$ with a self-loop transition.

Acceptance conditions have to be added to the tableau for each formula that promise to fulfill another formula. Namely, we cannot have an eventuality $\mathbf{AXA}(p\mathbf{U}r)$ where r is never fulfilled. A formula ψ is said to promise the formula r if ψ has the form $\mathbf{A}(p\mathbf{U}r)$. Once a particle constraints a formula ψ promising r, it must contain an r to fulfill the eventuality. The states in the Kripke structure satisfying such a condition are called *acceptance states*. Regarding each eventuality in the formula, there is a set of corresponding acceptance states, and we use an integer number to identify these sets of acceptance states.

The above construction procedure guarantees that the tableau satisfies the formula and includes every possible model satisfying the formula. The tableau size is smaller than that in [4]. The formal proof can be found in [10].

Algorithm 2 succ(P)

if a particle P is not realizable or does not include any formula of the form $\mathbf{AX}(\varphi)$ **then**
  SUCC $(P) = \{\emptyset\}$.
else
  SUCC (P) =COVER (IMPS (P)), where IMPS $(P) = \{\varphi : \mathbf{AX}(\varphi) \in P\}$.
end if

Algorithm 3 shows the procedure to construct formula ϕ into a Kripke structure $(I, S, \delta, \lambda, \mathcal{F})$ where I are the initial states; S the state space; δ the transition relation; λ the state label; and $\mathcal{F}$ the set of the acceptance states $\mathcal{F}_i$.

Algorithm 3 tableau(ϕ)

I :=COVER (ϕ);
$S := I$;
$\delta := \emptyset$;
Mark all particles in S as unprocessed
for each unprocessed particle P such that $P \in S$ **do**
  S' :=SUCC (P).
  for each unprocessed particle Q such that $Q \in S'$ **do**
    if Q is not a subset of any particle in S **then**
      Add Q to S.
      Mark Q as unprocessed.
    end if
    Add (P, Q) to δ.
    $\lambda(P) := P \cap \{p : p \in AP\} \cup P \cap \{\neg p : p \in AP\}$.
  end for
  Mark P as processed.
  if P promise r as $\mathbf{A}(p\mathbf{U}r)$ **then**
    Add $(S-$ COVER $(\mathbf{AXA}(p\mathbf{U}r))) \cup$ COVER (p) to accept state $\mathcal{F}_i$.
  end if
end for

Example. The cover of the formula φ : $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$ is $\{P_1, P_2, P_3\}$, where P_1, P_2, P_3 are particles and

[2]Two particles ρ and τ are compatible if $\forall_{p \in \rho, q \in \tau} p \wedge q \neq \mathbf{false}$

$$P_1 = \{\mathbf{A}(\mathbf{true}\mathbf{U}p), p\}$$
$$P_2 = \{\mathbf{A}(\mathbf{true}\mathbf{U}p), \mathbf{AXA}(\mathbf{true}\mathbf{U}p)\}$$
$$P_3 = \{\mathbf{A}(\mathbf{false}\mathbf{V}\neg p), \mathbf{AXA}(\mathbf{false}\mathbf{V}\neg p), \neg p\}$$

P_1, P_2, and P_3 have enclosed all the possibilities to satisfy φ, namely formula φ is covered by P_1, P_2, and P_3. The generated tableau of $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$ is shown in Figure 2, which has four states. States S_0, S_1, and S_2 are initial states. S_3 has no labeling meaning that the atomic formulas can be either true or false in this state. S_1, S_2, and S_3 are acceptance states meaning that these states have to be visited infinitely often to satisfy the formula.

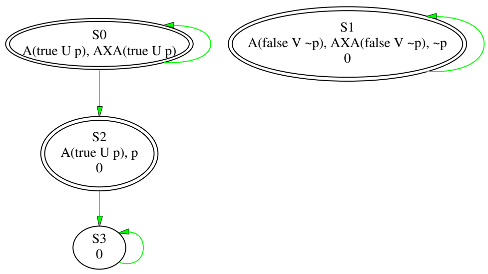

Figure 2. Tableau of $\mathbf{A}(\mathbf{true}\mathbf{U}p) \vee \mathbf{A}(\mathbf{false}\mathbf{V}\neg p)$

The generated tableau now can be easily synthesized into Verilog behavioral level HDL code through a conventional encoding of finite state machine (FSM) and by taking the advantages of VIS-Verilog specific features such as non-deterministic assignments. The synthesis procedure is shown in Figure 3, where state variables, initial blocks, fairness constraints, and sequential behaviors are synthesized. An example is available in the next section.

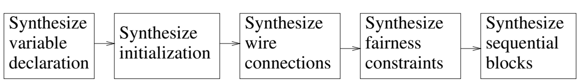

Figure 3. Verilog synthesis procedures

4. Case Study: A Nortel ATM Switch Fabric

In this section, we describe the application of the proposed environment synthesis in the compositional verification of an ATM (Asynchronous Transfer Mode) switch fabric from Nortel Networks [12]. The basic purpose of an ATM switch fabric is to transport valid (i.e., uncorrupted) ATM cells arriving at its ingress ports to the designated egress ports. Invalid ATM cells are to be discarded.

4.1. Modeling the Switch Fabric

There are mainly four modules in the ATM switch fabric at hand: *ATM_SWITCH*, *ATM_MON*, *FIFO_QUEUE*, and *ATM_GEN* as shown in Figure 4. *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 reconstruction functions.

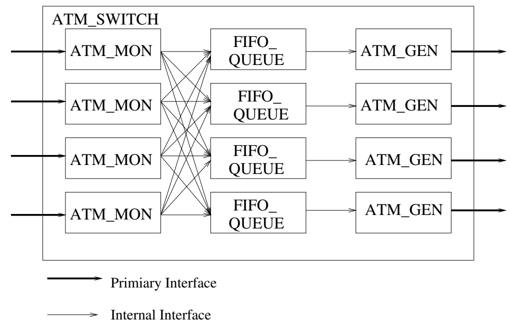

Figure 4. ATM switch fabric

The major *functional* property of such an ATM switch fabric is that "Valid cells (with good HEC and matching VPI/VCI) are switched correctly". Here, since all the cells are queued in the *FIFO_QUEUE* module, we specify the ingress part, the queue, and the egress part separately and extract the local properties respectively (properties shown in Table 1). Namely, in the ingress part, valid cells are switched into the queue, and in the egress part, cells in the queue are reconstructed and sent.

4.2. 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 Translate_head local property $Ingress\_\varphi$: "if the VPI/VCI is correct, the ATM cell will eventually be routed". The other properties can be proved in a similar way.

In the verification of $Ingress\_\varphi$, we want to check if 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 this switch fabric, only those *VPI_VCI_IN* with bits 27 to 4 being 0 can find a match, the corresponding environment specification is[3]: "$\mathbf{AF}(VPI\_VCI\_IN[27:4] = 0)$". A tableau can be constructed from this assumption and then the tableau is synthesized into Verilog behavior program. This program then can

[3]Actually, our tool supports arbitrary length formulas. We choose a small example in this paper because of space limitation.

be composed with the block under verification, i.e., *Translate_head* and the composed system then can be fed into VIS.

The synthesized Verilog of the above environment specification is shown below, where line numbers are added manually for illustration purposes. Interested readers can refer to [10] for the full length program. Lines 14 to 18 do set the initial states. Lines 50 to 70 are the behaviors of this environment, which is a small FSM.

```
...
L6: module tableau(VPI_VCI_IN);
L7: output[27:4] VPI_VCI_IN;
...
L13: //Initialiazation
L14: assign S_INIT_W_TMP = $ND(0, 1, 2, 3);
L15: assign S_INIT_W = ((S_INIT_W_TMP == 3))
            ? 2 : S_INIT_W_TMP;
L16: initial begin
L17:     STATE = S_INIT_W;
L18: end // Initial
...
L50: //Sequential part
L51: always begin
L52:    case (STATE)
L53:    0: begin
L54:        VPI_VCI_IN[27:4] = 1;
L55:        STATE = S0_NEXT_W;
L56:       end
...
L65:    3: begin
L66:        VPI_VCI_IN = VPI_VCI_INND_W;
L67:        STATE = S3_NEXT_W;
L68:       end
L69:    endcase // case (STATE)
L70: end // always begin
L71: endmodule // tableau$
```

The fairness constraint file we also synthesized 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
);
```

The verification results of sample properties are shown in Table 1. The verification was made on a SUN Enterprise server with 6GB memory. Through out the model checking, we set VIS model checker with the options: implicit clocking and advanced ordering.

After the above verification, we actually proved that, given certain environment assumptions, every block satisfies its local properties. 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_φ means the synthesized Verilog module of formula φ.

Property	VIS + **ACTL** Tableau	
	CPU Time (s)	Memory (MB)
$Ingress\_P_1$	19.5	0.908
$Ingress\_P_2$	2722.4	1.908
$Ingress\_P_3$	1.4	1.308
$Ingress\_P_4$	3.8	9.90
$Ingress\_P_5$	11.3	8.54
$Ingress\_P_6$	11.6	88.51
$Ingress\_P_7$	3.7	9.918
$Ingress\_P_8$	15.1	100.6
$Egress\_P_1$	25	1.44
$Egress\_P_2$	16.5	112.3
$Egress\_P_3$	6.7	12.2

Table 1. Verification statistics of sample properties

$$
\begin{array}{r}
Detect\_head \models Ingress\_P_1 \\
Eva\_head \| T_{Ingress\_P_1} \models Ingress\_P_2 \\
Eva\_head \| T_{Ingress\_P_1} \models Ingress\_P_3 \\
Unpack\_cell \| T_{Ingress\_P_2} \| T_{Ingress\_P_3} \models Ingress\_P_4 \\
Translate\_head \| T_{Ingress\_P_4} \models Ingress\_P_5 \\
Translate\_head \| T_{Ingress\_P_4} \models Ingress\_P_6 \\
Pack\_cell \| T_{Ingress\_P_5} \| T_{Ingress\_P_6} \models Ingress\_P_7 \\
Flattened\_cell \| T_{Ingress\_P_7} \models Ingress\_P_8 \\
FIFO\_Status \models Egress\_P_1 \\
De\_queue \models Egress\_P_2 \\
Send\_cell \| T_{Egress\_P_1} \| T_{Egress\_P_2} \models Egress\_P_3 \\
\hline
T_{Ingress\_P_1} \| Detect\_head \| Eva\_head \| Unpack\_cell \\
\| Translate\_head \| Pack\_cell \| Flattened\_cell \\
\| En\_queue \| FIFO\_Status \| De\_queue \\
\| Restruct\_cell \| Send\_cell \models Egress\_P_3
\end{array}
$$

Actually, the global property: "Valid cells (with good HEC and matching VPI/VCI) are switched correctly" is given by assuming $Ingress\_P_1$ (valid cells) and deducing $Egress\_P_3$ (correct switch). This way, we are checking the satisfaction of the global property against the whole design. Through out the verification, we also found some bugs in the design. For example, the outgoing real cell may be covered by an idle cell because of timing problems where it can be detected by local property $Egress\_P_3$.

4.3. Comparison with FormalCheck

As a further experiment, we tried the verification of the same switch fabric in FormalCheck [3] because of its strong model reduction capabilities and Verilog supports. We set FormalCheck using the options: reduction iterated with empty reduction seeds since there are no constraints on the primary inputs, and the symbolic BDD runtime option since it allows an efficient model checking. Due to the fact that the design contains complicated logic and big data-path, the attempt to prove the global property directly against the whole model failed because of state space explosion, even after using model reduction options. In order to prove this property, compositional verification is necessary. However, FormalCheck provides environment constraint functions by specifying **LTL** like formulas, namely we can only constrain I/O signals through the dialog box in the tool using **G**(always) and **F**(eventually) operators. In the case of the ATM switch fabric, there exist environment specifications of the modules, which are more complicated than what FormalCheck can support due to the logic expressiveness limitations, for example the case of specifying the environment behaviors in consecutive clock cycles. An alternative approach around this problem is to use our tableau construction procedure and synthesize the Verilog environment. Namely we can try to integrate the synthesized Verilog environment with the design and feed the composed system into the tool. Nevertheless, since FormalCheck is an RTL verification tool, it does not support behavioral Verilog features which we need to establish the environment, for example, initial blocks with specific initial values on signals, etc. It is hence impossible to feed such a model into FormalCheck.

5. Conclusion

In this paper, we have presented a possible implementation of assume-guarantee approach where assume and guarantee formulas are both written in **ACTL**. The approach is based on the environment synthesis which allows us to encode assumptions in the state space and hence make model checking more practical. Besides, our synthesis is based on a new tableau construction which builds a maximal model of the environment. An ATM switch fabric from Nortel Networks is used as a case study. The verification results showed that using our approach, we can manage to verify global properties of the fabric, where the verification is impossible in other existing tools.

References

[1] Y. Abarbanel, I. Beer, L. Gluhovsky, K. Keidar, and Y. Wolfsthal. FoCs: Automatic generation of simulation checkers from formal specifications. In *Proceedings of Computer Aided Verification*, LNCS 1855, pages 538–542, Chicago, IL, USA, July 2000. Springer Verlag.

[2] R. K. Brayton et al. VIS: A system for verification and synthesis. In *Proceedings of Computer Aided Verification*, volume 1102 of *LNCS*, pages 428–432, Rutgers University, NY, USA, July 1996. Springer Verlag.

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

[4] O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[5] S. Katz, D. Geist, and O. Grumberg. Have I written enough properties? - a method of comparison between specification and implementation. In *Proceedings of Conference on Correct Hardware Design and Verification Methods*, volume 1703 of *LNCS*, pages 280–297, Bad Herrenalb, Germany, September 1999. Springer Verlag.

[6] O. Kupferman and M. Y. Vardi. On the complexity of branching modular model checking. In *Proceedings of Concurrent Theory*, pages 408–422, Philadelphia, Pennsylvania, USA, August 1995. Extended abstract.

[7] M. Maidl. The common fragment of ctl and ltl. In *41st Annual Symposium on Foundations of Computer Science*, pages 643–653, Redondo Beach, California, November 2000.

[8] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Safety*. Springer Verlag, New York, 1991.

[9] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *Proceedings of SPIN Workshop 1999*, pages 168–183, Trento, Italy, June 1999.

[10] H. Peng. *Compositional Verification for Large Scale Designs: Environment Synthesis and Model Reduction*. PhD thesis, Electrical and Computer Engineering, Concordia University, September 2002.

[11] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formula. In *Proceedings of Computer Aided Verification*, volume 1855 of *LNCS*, pages 248–263, Chicago, IL, USA, July 2000. Springer Verlag.

[12] Northern Telecom. *Specification of a 4*4 ATM switch*, November 1998.

[13] M. Y. Vardi. On the complexity of modular model checking. In *Proceedings of the 10th IEEE Symposium on Logic in Computer Science*, pages 101–111, San Diego, California,USA, June 1995.