CTL Model Checking Based on Forward State Traversal

Hiroaki Iwashita Tsuneo Nakata Fumiyasu Hirose
Fujitsu Laboratories Ltd.
1-1 Kamikodanaka 4-chome, Nakahara-ku, Kawasaki 211-88, Japan
hiroaki@flab.fujitsu.co.jp

Abstract

We present a CTL model checking algorithm based mainly on forward state traversal, which can check many realistic CTL properties without doing backward state traversal. This algorithm is effective in many situations where backward state traversal is more expensive than forward state traversal. We combine it with BDD-based state traversal techniques using partitioned transition relations. Experimental results show that our method can verify actual CTL properties of large industrial models which cannot be handled by conventional model checkers.

1 Introduction

Formal verification techniques are starting to become standard industrial hardware verification methods, since people know that design complexity continues to grow and that they cannot keep up using only conventional verification methods, such as logic simulation. For sequential systems, one of the most popular and feasible approaches to formal verification is CTL model checking [1, 2, 3]. It is a method for verifying that a sequential system satisfies a property where the system is modeled as a finite state machine (FSM) and the property is expressed in a temporal logic called Computation Tree Logic (CTL) [1].

Rapid progress has been made in model checking techniques since binary decision diagrams (BDDs) [4] were incorporated into them [5, 6]. It has been shown that these techniques, called symbolic model checking, can potentially verify large FSMs with more than 10^{20} states. The effectiveness of symbolic model checking, however, heavily depends on the model's structure and BDD variable ordering. In practice, symbolic model checking of a complicated system can only be accomplished by a few experts who can fine tune the model based on their knowledge of both the target system and the symbolic model checker.

Some techniques using partitioned transition relations have been proposed to verify larger FSMs [7, 8]. Memory usage is reduced by making the partition size smaller, in general. We often observe, however, CPU time for backward state traversal becomes much larger when we make the partition size smaller. We cannot always find a good partitioning for backward state traversal that satisfies both memory and time limitations. On the other hand, we also observe that CPU time for forward state traversal is not affected by the partition size so much. We can use partitioned transition relations composed of individual latch transition functions to reduce memories, without great loss of execution speed.

In the symbolic model checking paradigm, CTL formulas have been evaluated with backward state traversal [2]. We describe, in this paper, an algorithm to accomplish CTL model checking in the opposite direction. It is effective in many situations where backward state traversal is more expensive than forward state traversal. After preliminaries in Section 2, Section 3 characterizes forward state traversal and backward state traversal using partitioned transition relations. Section 4 presents our new procedure for evaluating CTL properties with forward state traversal. We report the results in Section 5.

2 Preliminaries

2.1 Transition relations

An FSM is a 6-tuple, $(S, I, O, \delta, \lambda, s_0)$, where S is the set of states, I is the set of input values, O is the set of output values, $\delta : S \times I \rightarrow S$ is the next state function, $\lambda : S \times I \rightarrow O$ is the output function, and $s_0 \in S$ is the initial state. In what follows, let $B = \{0, 1\}$.

The *transition relation* of an FSM is the function $T : S \times I \times S \rightarrow B$; $T(x, i, y) = 1$ iff $y = \delta(x, i)$. In the basic symbolic technique, a single BDD is built for representing the transition relation T. For a large FSM, however, we often fail to construct the BDD for T because of a BDD size explosion. *Partitioned transition relations* [8] are popular representation to reduce the BDD size. When the state is expressed by a vector of n Boolean state variables (latches) and the transition function of the k-th latch is given by $\delta_k(\vec{x}, \vec{i})$, we can make a conjunctive partitioned transition relation as follows:

$$\begin{aligned} T(\vec{x}, \vec{i}, \vec{y}) &= T_1(\vec{x}, \vec{i}, y_1) \wedge \ldots \wedge T_n(\vec{x}, \vec{i}, y_n) \\ T_k(\vec{x}, \vec{i}, y_k) &= \Big(y_k \equiv \delta_k(\vec{x}, \vec{i}) \Big) \end{aligned}$$

ICCAD '96
1063-6757/96 $5.00 © 1996 IEEE

Latch transition relations $T_1, \ldots, T_n$ are represented by n BDDs, which are much smaller than the BDD for T in general.

2.2 Images and pre-images

Given an FSM $(S, I, O, \delta, \lambda, s_0)$ and a set of states $A \subseteq S$, the *image* of A is defined to be the set of states $\{ y \mid \exists x \in A, \exists i \in I, y = \delta(x, i) \}$, and the *pre-image* of A is defined by the set of states $\{ x \mid \exists y \in A, \exists i \in I, y = \delta(x, i) \}$.

A set of states $A \subseteq S$ can be represented by a *characteristic function* $\chi_A : S \to B; \chi_A(x) = 1$ iff $x \in A$. We use a notation $\mathcal{L}(f)$ to express the set of states represented by function f, in this paper. The image and the pre-image of $\mathcal{L}(f)$ are calculated by following symbolic operations:

$$
\begin{aligned}
&\mathit{Img}(f)(\vec{y}) \\
&\quad = \exists \vec{x}. \exists \vec{i}. \left[T(\vec{x}, \vec{i}, \vec{y}) \wedge f(\vec{x}) \right] \\
&\quad = \exists \vec{x}. \exists \vec{i}. \left[T_1(\vec{x}, \vec{i}, y_1) \wedge \ldots \wedge T_n(\vec{x}, \vec{i}, y_n) \wedge f(\vec{x}) \right] \\
&\mathit{Pre}(f)(\vec{x}) \\
&\quad = \exists \vec{i}. \exists \vec{y}. \left[T(\vec{x}, \vec{i}, \vec{y}) \wedge f(\vec{y}) \right] \\
&\quad = \exists \vec{i}. \exists \vec{y}. \left[T_1(\vec{x}, \vec{i}, y_1) \wedge \ldots \wedge T_n(\vec{x}, \vec{i}, y_n) \wedge f(\vec{y}) \right]
\end{aligned}
$$

The image and the pre-image of $\mathcal{L}(f)$ are $\mathcal{L}(\mathit{Img}(f))$ and $\mathcal{L}(\mathit{Pre}(f))$ respectively. Operations $\mathit{Img}(f)$ and $\mathit{Pre}(f)$ are similar if T is given by a single BDD. When we use conjunctive partitioned transition relations, early existential quantification can be done while calculating the conjunction of all BDDs [7, 8]. Efficiency of the calculation strongly depends on the order in which the BDDs are processed. Strategies to find effective orders for $\mathit{Img}(f)$ and $\mathit{Pre}(f)$ seem to be different, because those expressions have different forms.

2.3 CTL model checking

Model checking is the process of determining whether a model (FSM) satisfies its requirements (properties). A temporal logic *CTL* [1] is commonly used to express properties about an FSM. CTL formulas are composed of atomic propositions with usual logical operators and following temporal operators:

- $\mathbf{EX}\, f$ ($\mathbf{AX}\, f$) which means that f holds at some (every) successor state of the current state.
- $\mathbf{EF}\, f$ ($\mathbf{AF}\, f$) which means that for some (every) state transition path, there exists a state on the path at which f holds.
- $\mathbf{EG}\, f$ ($\mathbf{AG}\, f$) which means that for some (every) state transition path, f keeps holding forever on the path.
- $\mathbf{E}\,[g\,\mathbf{U}\,f]$ ($\mathbf{A}\,[g\,\mathbf{U}\,f]$) which means that for some (every) state transition path, there exists a state on the path at which f holds, and g holds at all the preceding states.

A CTL property is expressed by a notation like "$M, s \models f$." It means that the CTL formula f is true in state s of model M. It is also written simply as "$s \models f$" where the model is not ambiguous.

CTL formula f can be interpreted as a set of states $\mathcal{L}(f) = \{ s \mid s \models f \}$. $\mathbf{EX}\, f$ is then the same operation as computing the pre-image of $\mathcal{L}(f)$:

$$\mathbf{EX}\, f \;=\; \mathit{Pre}(f)$$

$\mathbf{E}\,[g\,\mathbf{U}\,f]$ and $\mathbf{EG}\, f$ can be characterized by the least and greatest fix-point computation as follows:

$$
\begin{aligned}
\mathbf{E}\,[g\,\mathbf{U}\,f] &= \mathbf{lfp}\, Z\ [f \vee (g \wedge \mathbf{EX}\, Z)] \\
\mathbf{EG}\, f &= \mathbf{gfp}\, Z\ [f \wedge \mathbf{EX}\, Z]
\end{aligned}
$$

The remaining operators are given by following rules:

$$
\begin{aligned}
\mathbf{EF}\, f &= \mathbf{E}\,[\mathrm{true}\,\mathbf{U}\,f] \\
\mathbf{AX}\, f &= \neg\, \mathbf{EX} \neg f \\
\mathbf{AF}\, f &= \neg\, \mathbf{EG} \neg f \\
\mathbf{AG}\, f &= \neg\, \mathbf{EF} \neg f \\
\mathbf{A}\,[g\,\mathbf{U}\,f] &= \neg\, (\mathbf{E}\,[\neg f\,\mathbf{U}\,\neg g \wedge \neg f] \vee \mathbf{EG} \neg f)
\end{aligned}
$$

A *fairness constraint* is a condition representing fair state transition paths in which we are interested. CTL model checking under fairness constraints is performed by restricting state transition paths along which each fairness constraint holds infinitely often. Fairness constraints are given by a set of CTL formulas C. CTL formula $\mathbf{EG}\, f$ under fairness constraints in C is computed as follows [2]:

$$\mathbf{E}_C \mathbf{G}\, f \;=\; \mathbf{gfp}\, Z \left[f \wedge \mathbf{EX} \bigwedge_{c \in C} \mathbf{E}\,[Z\,\mathbf{U}\,Z \wedge c] \right]$$

The set of states that are the start of some fair path under fairness constraints in C is given by $\mathcal{L}(\mathbf{E}_C \mathbf{G}\,\mathrm{true})$. Once $\mathbf{E}_C \mathbf{G}\,\mathrm{true}$ is evaluated, $\mathbf{EX}\, f$ and $\mathbf{E}\,[g\,\mathbf{U}\,f]$ under fairness constraints in C can be computed simply as follows:

$$
\begin{aligned}
\mathbf{E}_C \mathbf{X}\, f &= \mathbf{EX}\,(f \wedge \mathbf{E}_C \mathbf{G}\,\mathrm{true}) \\
\mathbf{E}_C\,[g\,\mathbf{U}\,f] &= \mathbf{E}\,[g\,\mathbf{U}\,(f \wedge \mathbf{E}_C \mathbf{G}\,\mathrm{true})]
\end{aligned}
$$

3 Forward versus backward traversal

Conventional CTL model checkers evaluate CTL formulas with repeated pre-image computation, *backward state traversal*. Properties $s_0 \models \mathbf{EF}\, f$ and $s_0 \models \mathbf{AG}\, f$ are also

Model	#Latches	#States	Depth	Description
gigamax	16	1.2×10^{2}	8	An example in VIS distribution (originated in SMV).
atm_sw	54	2.0×10^{5}	126	Abstracted processor core for an ATM-switch [9].
dh_1	46	4.0×10^{3}	21	Cache coherency protocol description focused on a multiprocessor BUS.
dh_2	66	7.9×10^{6}	17	Cache coherency protocol description focused on processor interaction.
vpp	101	2.4×10^{11}	18	Abstracted VLIW microprocessor pipeline.
pipe_s	35	2.9×10^{8}	11	Abstracted superscalar microprocessor pipeline; a simple version.
pipe_d	73	5.7×10^{17}	11	Abstracted superscalar microprocessor pipeline; a complicated version.

Table 1: Benchmark examples

Model	Complete TR			Partitioned TR		
	#Nodes	Image	Pre-image	#Nodes	Image	Pre-image
gigamax	0.5K	0.0	0.0	1K	0.3	0.0
atm_sw	1302K	8.9	21.8	1K	10.9	202.5
dh_1	3K	0.0	0.8	1K	0.3	1.3
dh_2	100K	13.8	75.4	35K	46.8	>10000
vpp	N/A	N/A	N/A	60K	3.0	4135.1
pipe_s	322K	10.7	14.8	3K	0.6	387.9
pipe_d	N/A	N/A	N/A	348K	200.4	>20000

Table 2: CPU seconds per image or pre-image computation

known to be verified by comparing $\mathcal{L}(f)$ and the reachable states. Reachable states are enumerated with repeated image computation, *forward state traversal*.

Performance of the computation is very sensitive to BDD variable ordering. It is difficult to find a good variable order automatically, and ordinary users cannot always find it manually. When we use conjunctive partitioned transition relations, the performance is also sensitive to the order in which the BDDs are processed. In our experience of industrial hardware verification, however, *image* computation with *partitioned* transition relation works relatively fine even if the FSM is very large and the ordering is not tuned so much.

Table 1 summarizes seven models that we use as benchmark examples in this paper. Excepting gigamax, the models and those properties are actual ones that we have prepared for verifying real hardware products. For each benchmark, we tried conventional model checking procedure that checks reachability with forward state traversal and evaluates CTL formulas with backward state traversal. We used both a complete transition relation represented by a single BDD and a conjunctive partitioned transition relation represented by a set of BDDs for latch transition relations.

Table 2 shows average CPU time per image or pre-image computation during each model checking process. Total numbers of BDD nodes for complete/partitioned transition relations are also shown in the table. Complete transition relations for models vpp and pipe_d could not be made because of BDD size explosions. Pre-image computation with partitioned transition relation for models dh_2 and pipe_d exceeded the CPU time limit of 24 hours. We should not compare image computation time and pre-image computation time directly, because they are solving different problems. The results, however, show that the time ratio of image computation and pre-image computation with a partitioned transition relation is huge, while that with a complete transition relation is relatively small.

4 Forward CTL model checking

As described in Section 3, there is a large CPU time difference between image computation and pre-image computation especially when a conjunctive partitioned transition relation is used. We should traverse state space forward when we can choose the direction. Although CTL model checking has been based on backward state traversal, we propose a new method for checking CTL properties using forward state traversal in this section.

4.1 Rewriting property notations

A CTL property is given as a notation like "$s_0 \models f$." Conventional model checking procedure matches the notation: it evaluates CTL formula f with backward state traversal, and then checks if it holds at state s_0. We rewrite the notation for the purpose of matching it with our method. We translate the CTL property into a problem of comparing a formula with the constant false. Let s_0 be a state of the

FSM, p_0 the characteristic function of $\{s_0\}$, and f an arbitrary CTL formula. Formula p_0 is true only at state s_0, and formula f is true at state s_0 iff $s_0 \models f$ holds. Therefore, the "$\models$" notation can be rewritten as follows:

$$s_0 \models f \iff p_0 \wedge f \neq \text{false} \quad (1)$$
$$s_0 \models f \iff p_0 \wedge \neg f = \text{false} \quad (2)$$

Some model checkers support models with multiple initial states, while "$\models$" represents relation between a single state and a CTL formula. Given a set of initial states S_0, we believe that interpretation of some extended notation like "$S_0 \models f$" is ambiguous. It should be written as "$\exists s \in S_0,\ s \models f$" or "$\forall s \in S_0,\ s \models f$". They can be rewritten as "$p_0 \wedge f \neq \text{false}$" and "$p_0 \wedge \neg f = \text{false}$" respectively where p_0 is the characteristic function of S_0.

4.2 Forward EX evaluation

Let p and f be formulas. We can replace an outermost **EX** evaluation with image computation as follows:

$$p \wedge \mathbf{EX}\, f \neq \text{false} \iff \mathit{Img}\,(p) \wedge f \neq \text{false} \quad (3)$$

Proof: Assume $p \wedge \mathbf{EX}\, f$ holds at state s. Then p holds at s and f holds at some successor state of s, say t. *Img* (p) holds at t since *Img* (p) holds at any successor state of $s \in \mathcal{L}(p)$. Thus, *Img* $(p) \wedge f$ holds at t. Conversely, assume *Img* $(p) \wedge f$ holds at state t. Then f holds at t and t is a successor state of some state $s \in \mathcal{L}(p)$. $\mathbf{EX}\, f$ holds at s since $t \in \mathcal{L}(f)$ is a successor state of s. Thus, $p \wedge \mathbf{EX}\, f$ holds at state s. □

Notice that we have removed an operator **EX** from f. Using equation (3) again or using one of the equations described later, it is possible to continue conversion of a backward traversal operator in f into a forward traversal operator.

4.3 Forward EU evaluation

We define a state enumeration procedure under constraints given by two formulas p and q:

$$\mathit{FwdUntil}\,(p, q) = \mathbf{lfp}\, Z\ [p \vee \mathit{Img}\,(Z \wedge q)]$$

An element of $\mathcal{L}(\mathit{FwdUntil}\,(p, q))$ is a state t such that there exists a path through t from some state at which p holds, and q holds at all states before t on the path.

Using the *FwdUntil* () operator, we can replace an outermost **EU** evaluation as follows:

$$p \wedge \mathbf{E}\,[q\,\mathbf{U}\,f] \neq \text{false} \iff \mathit{FwdUntil}\,(p, q) \wedge f \neq \text{false} \quad (4)$$

Proof: Assume $p \wedge \mathbf{E}\,[q\,\mathbf{U}\,f]$ holds at state s. Then both p and $\mathbf{E}\,[q\,\mathbf{U}\,f]$ hold at s. It means that there exists a path from s through some state $t \in \mathcal{L}(f)$, and q holds at all states before t. Thus, *FwdUntil* $(p, q) \wedge f$ holds at t. Conversely, assume *FwdUntil* $(p, q) \wedge f$ holds at state t. Then both *FwdUntil* (p, q) and f holds at t. There exists a path through t from some state $s \in \mathcal{L}(p)$, and q holds at all states before t. Thus, $p \wedge \mathbf{E}\,[q\,\mathbf{U}\,f]$ holds at s. □

Now the operator **EU** have been removed from f. Thus, we have a chance again to convert a backward traversal operator in f into a forward traversal operator, as in equation (3).

4.4 Forward EG evaluation

We define an operator like **EG**, except that pre-image computation is replaced by image computation:

$$\mathit{EH}\,(p) = \mathbf{gfp}\, Z\ [p \wedge \mathit{Img}\,(Z)]$$

EH (p) is used to check whether there exists a state transition cycle in $\mathcal{L}(p)$. $\mathcal{L}(\mathit{EH}\,(p))$ is the subset of $\mathcal{L}(p)$ such that every state is reachable from a cycle through states only in $\mathcal{L}(p)$. We also define simple composite operators:

$$\mathit{Reachable}\,(p, q) = \mathit{FwdUntil}\,(p, q) \wedge q$$
$$\mathit{FwdGlobal}\,(p, q) = \mathit{EH}\,(\mathit{Reachable}\,(p, q))$$

Reachable (p, q) computes the subset of $\mathcal{L}(q)$ whose elements can be reached from $\mathcal{L}(p \wedge q)$ through states only in $\mathcal{L}(q)$. *FwdGlobal* (p, q) checks whether there exists a state transition cycle in $\mathcal{L}(q)$ that is reachable from $\mathcal{L}(p \wedge q)$ through states only in $\mathcal{L}(q)$.

Using the *FwdGlobal* () operator, we can replace an outermost **EG** evaluation as follows:

$$p \wedge \mathbf{EG}\, q \neq \text{false} \iff \mathit{FwdGlobal}\,(p, q) \neq \text{false} \quad (5)$$

Proof: Assume $p \wedge \mathbf{EG}\, q$ holds at state s. Then both p and $\mathbf{EG}\, q$ holds at s. It means that for some path from s, q keeps holding forever on the path. In other words, there exists a cycle in $\mathcal{L}(q)$ and it is reachable from s through states only in $\mathcal{L}(q)$. $\mathcal{L}(\mathit{Reachable}\,(p, q))$ includes the cycle, since it includes all the states reachable from $s \in \mathcal{L}(p \wedge q)$ through states only in $\mathcal{L}(q)$. Thus, $\mathit{EH}\,(\mathit{Reachable}\,(p, q)) \neq \text{false}$. Conversely, assume $\mathit{EH}\,(\mathit{Reachable}\,(p, q)) \neq \text{false}$. There exists a cycle in $\mathcal{L}(\mathit{Reachable}\,(p, q))$. It means that the cycle is in $\mathcal{L}(q)$ and is reachable from some state $s \in \mathcal{L}(p)$ through states only in $\mathcal{L}(q)$. Thus, $p \wedge \mathbf{EG}\, q$ holds at s. □

4.5 Forward fair EG evaluation

We also introduce fairness constraints into forward CTL evaluation. The key is exactly like ordinary fair CTL evaluation, a procedure to find fair cycles. The procedure that

compute $EH(p)$ under fairness constraints C is given as follows:

$$FairEH(p) = \mathbf{gfp}\, Z \left[p \wedge Img\left(\bigwedge_{c \in C} Reachable(c, Z) \right) \right]$$

We then modify the *FwdGlobal*() operator to handle fairness constraints using *FairEH*():

$$FwdFairGlobal(p, q) = FairEH(Reachable(p, q))$$

Using the *FwdFairGlobal*() operator, we can replace an outermost **EG** evaluation under fairness constraints as follows:

$$\begin{aligned} & p \wedge \mathbf{E}_C \mathbf{G}\, q \neq \text{false} \\ & \quad \Longleftrightarrow \quad FwdFairGlobal(p, q) \neq \text{false} \end{aligned} \tag{6}$$

It is clear from the fact that both sides are the modified version of equation (5) that restrict paths under the same constraints.

4.6 The conversion procedure

Using conversion rules (3), (4), (5), and (6), we can replace **EX**, **EU**, **EG**, and $\mathbf{E}_C\mathbf{G}$ with forward traversal operators. An original property notation should be rewritten using either positive form (1) or negative form (2) so that the formula matches one of the rules. The problem of comparing a disjunctive expression with the constant false, such as "$f \vee g \neq$ false", can be divided into sub-problems, such as "$f \neq$ false" and "$g \neq$ false". We can check each term separately, and if one or more terms are not the constant false, the entire expression is not the constant false. We do not need to convert all CTL temporal operators into forward traversal operators. Remaining operators can be evaluated in usual manner, with backward state traversal. Hence, all CTL formulas can be handled with our method. The conversion procedure is shown below:

1. Rewrite the CTL formula only in temporal operators **EX**, **EU**, **EG**, and $\mathbf{E}_C\mathbf{G}$.

2. Translate "$\models$" notation into an expression comparing a formula with the constant false, using equation (1) or (2).

3. Arrange outermost logical operations in disjunctive form, and divide the problem into a set of sub-problems comparing each product term with the constant false.

4. For each sub-problem, convert a backward operator to a forward operator using one of equations (3), (4), (5), and (6), if applicable.

5. For each newly updated sub-problems, call the procedure recursively from step 3.

Although steps 2 and 4 have choice, it is easy to find good conversion for actual CTL properties. Many properties that we examined can be fully converted to forward state traversal problems, as shown in the next section.

Example Here is an example of converting one of the most common properties, "whenever a request is made, acknowledgment will return in the future," where *req* means the request, *ack* means the acknowledgment, s_0 is the initial state, and p_0 is the characteristic function of $\{s_0\}$:

$$\begin{aligned} & s_0 \models \mathbf{AG}(req \rightarrow \mathbf{AF}\, ack) \\ & \Longleftrightarrow \quad s_0 \models \neg\mathbf{E}[\text{true}\,\mathbf{U}\,(req \wedge \mathbf{EG}\,\neg ack)] \\ & \Longleftrightarrow \quad p_0 \wedge \mathbf{E}[\text{true}\,\mathbf{U}\,(req \wedge \mathbf{EG}\,\neg ack)] = \text{false} \\ & \Longleftrightarrow \quad FwdUntil(p_0, \text{true}) \wedge (req \wedge \mathbf{EG}\,\neg ack) = \text{false} \\ & \Longleftrightarrow \quad (FwdUntil(p_0, \text{true}) \wedge req) \wedge \mathbf{EG}\,\neg ack = \text{false} \\ & \Longleftrightarrow \quad FwdGlobal((FwdUntil(p_0, \text{true}) \wedge req), \neg ack) = \text{false}. \end{aligned}$$

5 Experimental results

5.1 Applicability to actual CTL properties

Our method becomes effective when many temporal operators in a CTL formula are converted into our forward traversal operators. We investigated examples in two existing symbolic model checkers SMV [2] and VIS [10], and also examined our own property examples.

We found that 90% of the properties can be rewritten using only the forward traversal operators: 18 properties out of 20 in the SMV examples, 47 properties out of 55 in the VIS examples, and all of our 13 properties. The rest of the properties are classified into three types:

$$\begin{aligned} & s_0 \models \mathbf{AG}\,\mathbf{EF}\, a \\ & \quad \Longleftrightarrow \quad FwdUntil(p_0, \text{true}) \wedge \neg\mathbf{EF}\, a = \text{false} \end{aligned}$$

$$\begin{aligned} & s_0 \models \mathbf{AG}(a \rightarrow \mathbf{EG}\, b) \\ & \quad \Longleftrightarrow \quad FwdUntil(p_0, \text{true}) \wedge a \wedge \neg\mathbf{EG}\, b = \text{false} \end{aligned}$$

$$\begin{aligned} & s_0 \models \mathbf{AG}((a \rightarrow \mathbf{EX}\, b) \wedge (b \rightarrow \mathbf{EX}\, a)) \\ & \quad \Longleftrightarrow \quad \begin{cases} FwdUntil(p_0, \text{true}) \wedge a \wedge \neg\mathbf{EX}\, b = \text{false} \\ FwdUntil(p_0, \text{true}) \wedge b \wedge \neg\mathbf{EX}\, a = \text{false} \end{cases} \end{aligned}$$

where a and b are atomic propositions. Both forward and backward traversal are used to check these properties. They are, in fact, the same as conventional methods that can unfold only outermost **AG** operators.

5.2 Performance of CTL model checking

We measured execution speed and memory usage of our model checker, named BINGO. We also executed SMV

Model	SMV	VIS	BINGO	
			Bwd	Fwd
`gigamax`	3.6sec 1.6MB	21.3sec 4.9MB	9.3sec 3.7MB	9.1sec 3.7MB
`atm_sw`	1734sec 8.3MB	*space*	9447sec 27.1MB	1158sec 23.1MB
`dh_1`	—	138.7sec 6.8MB	29.6sec 3.7MB	20.3sec 3.8MB
`dh_2`	—	*space*	*time*	1357sec 31.5MB
`vpp`	*time**	*space*	45569sec 17.5MB	223.3sec 13.2MB
`pipe_s`	23715sec 60.2MB	51.7sec 6.1MB	12429sec 5.5MB	64.6sec 4.9MB
`pipe_d`	—	*space*	*time*	8979sec 249.7MB

'*time*' exceeded 24 hours in CPU time.
'*space*' exceeded 300 megabytes of memory.
'*' failed while building a transition relation.

Table 3: Performance of CTL model checking

(version 2.2.4, released in 1994) and VIS (version 1.0, released in 1995) as references. BINGO makes a conjunctive partitioned transition relation composed of individual latch transition functions. Nondeterministic relations are translated into deterministic functions by adding unconstrained pseudo inputs. SMV makes a complete transition relation [2]. VIS makes a conjunctive partitioned transition relation by clustering several transition functions together [10].

We tried all models shown in Table 1 on BINGO and VIS. We also tried `gigamax`, `atm_sw`, `vpp`, and `pipe_s` on SMV. The `gigamax` description for SMV has a slightly different construction from that for others. Excepting `gigamax`, we ran the experiments for all model checkers with the same BDD variable order, which is defined statically according to the original model description. We also tested dynamic variable reordering on VIS; essential improvement could not be observed, however. Transition relation clustering in VIS is left unchanged from its default. The results are shown in Table 3. CPU times and memories were measured on a 50MHz SPARCstation10. We set a CPU time limit of 24 hours and a memory limit of 300 megabytes. BINGO is executed in two modes: the conventional model checking mode that unfolds only an outermost **AG** or **EF** operator (labeled "Bwd"), and the forward traversal mode that uses the algorithm presented in this paper (labeled "Fwd").

The results indicate that a fully partitioned transition relation and forward CTL model checking made a successful combination in BINGO. Some large models cannot be verified without using fully partitioned transition relations. Execution speed of forward CTL model checking is the fastest or nearly the fastest of the methods. It is remarkable that "Fwd" is extremely faster than "Bwd" in the large models that require fully partitioned transition relations.

6 Conclusion

In this paper, we have presented techniques for verifying CTL properties using forward state traversal. They can be mixed with conventional backward CTL evaluation techniques, and are applicable to arbitrary CTL properties. Most realistic properties can be converted to use only forward state traversal, while the class of such properties is still uncertain. When our method is combined with BDD-based state traversal techniques using partitioned transition relations, it becomes a very practical method for verifying large FSMs.

References

[1] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," in *ACM Transitions on Programming Languages and Systems*, 8(2), pp. 244–263, 1986.

[2] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[3] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao, "Efficient Generation of Counterexamples and Witness in Symbolic Model Checking," in *Proc. 32nd DAC*, pp. 427–432, 1995.

[4] R. E. Bryant, "Graph Based Algorithm for Boolean Function Manipulation," in *IEEE Transactions on Computers*, C-35(8), pp. 677–691, 1986.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill, "Sequential Circuit Verification Using Symbolic Model Checking," *Proc. 27th DAC*, pp. 46–51, 1990.

[6] O. Coudert and J. C. Madre, "A Unified Framework for the Formal Verification of Sequential Circuits," in *Proc. ICCAD-90*, pp. 126–129, 1990.

[7] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli, "Implicit State Enumeration of Finite State Machines using BDD's," in *Proc. ICCAD-90*, pp. 130–133, 1990.

[8] J. R. Burch, E. M. Clarke, and D. E. Long, "Representing Circuits More Efficiently in Symbolic Model Checking," in *Proc. 28th DAC*, pp. 403–407, 1991.

[9] B. Chen, M. Yamazaki, and M. Fujita, "Bug Identification of a Real Chip Design by Symbolic Model Checking," in *Proc. European Design and Test Conference*, pp. 132–136, 1994.

[10] The VIS Group, "VIS: A System for Verification and Synthesis," in *Proc. Conference on Computer Aided Verification*, 1996.