Forward Model Checking Techniques Oriented to Buggy Designs

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

Abstract

Forward model checking is an efficient symbolic model checking method for verifying realistic properties of sequential circuits and protocols. In this paper, we present the techniques that modify the order of state traversal in forward model checking, and that dramatically improve average CPU time for finding design errors. A failing property has to be checked again and again to analyze the bug until it is corrected. The techniques, therefore, can have significant impacts on actual verification tasks. We use a modified regular/\(\omega\)-regular expression to represent a set of illegal state transition sequences of an FSM. It makes the problem clear and gives us a sense of depth-first traversal, not on the state space, but on the property.

1 Introduction

Model checking is now attracting a great deal of industrial attention. It is beginning to be applied to commercial designs of digital circuits. Since many bugs are found in an early stage of design, model checking and correction must be repeated at this stage. A quick response from the model checker is desired, especially when design errors still remain. Ordinary model checkers, however, spend huge CPU time for checking a large design no matter whether it has a bug or not. It is the purpose of this paper to present efficient techniques for checking buggy designs.

Symbolic model checking is a successful technique in verifying sequential circuits and protocols, which uses binary decision diagrams (BDDs) [1] for symbolic representations of state sets and transition relations [3, 4, 5]. Forward model checking is an efficient variant of symbolic model checking, which replaces backward state traversal in ordinary symbolic model checking with forward state traversal. It has been shown that CTL model checking [2, 6, 7] can be done mainly based on symbolic forward state traversal [10]. We often observe that the forward model checking algorithm works much better than the ordinary backward algorithm.

We present practical aspects of forward model checking in this paper. We redefine forward model checking as a problem of finding an error trace of a finite state machine (FSM) using symbolic forward state traversal. It makes the problem clear and gives us a sense of depth-first traversal not on the state space but on the property. By introducing a mixture manner of breadth-first and depth-first traversals on the property, we can expect a model checker to find an error trace rapidly. The techniques keep advantages of symbolic model checking, and are very different from logic simulation. They still guarantee the perfect coverage and can find complicated error traces including cycles.

In Section 2, we rearrange the model checking problems. Section 3 describes basic forward model checking operations. Improved state traversal techniques are presented in Section 4. Experimental results and conclusions are given in Section 5 and Section 6, respectively.

2 Backward and forward model checking

2.1 Path set expressions

We define a class of modified regular/\(\omega\)-regular expressions, called path set expressions (PSEs) in this paper, to represent a set of possible error traces (illegal state transition sequences). A propositional formula represents a set of states at which the formula holds. We make no distinction between such a formula and a set of states in this paper. Let \(p\) and \(q\) be propositional formulas, \(\alpha\) a PSE representing finite sequences, and \(\beta, \gamma\) arbitrary PSEs, then:

- \([p]\) matches every one-step sequence \((s_1)\) such that \(s_1\) satisfies \(p\). We use ‘\(\ast\)’ as an abbreviation of ‘\([true]\)’.
- \([p]\ast [q]\) matches every finite sequence \((s_1, \ldots, s_n)\) for \(n \geq 1\), such that \(s_1, \ldots, s_{n-1}\) satisfy \(p\) if \(n \geq 2\), and \(s_n\) satisfies \(q\).
- \([p]\omega\) matches every infinite sequence \((s_1, s_2, \ldots)\) such that \(s_k\) satisfies \(p\) for all \(k \geq 1\).
- \(\alpha \beta\) matches every sequence \((s_1, \ldots, s_n, s_{n+1}, \ldots)\) such that the finite sequence \((s_1, \ldots, s_n)\) matches \(\alpha\) and the sequence \((s_{n+1}, \ldots)\) matches \(\beta\).
- \(\alpha : \beta\) matches every sequence \((s_1, \ldots, s_n, \ldots)\) such that the finite sequence \((s_1, \ldots, s_n)\) matches \(\alpha\) and the sequence \((s_n, \ldots)\) matches \(\beta\).
• \( \beta + \gamma \) represents the union of \( \beta \) and \( \gamma \).

Note that we introduced an extended operator \( \cdot* \), which means concatenation of two expressions with one-step overlap. PSEs do not match a null sequence. Repeat operators \( \cdot* \) and \( \cdot\omega \) are only applicable to expressions representing one-step sequences. An \( \cdot\omega \) operator appears only at the end of concatenation.

For example, where \( i \) represents the set of initial states, a set of error traces “Illegal condition \( c \) is satisfied” is written as “\([i] \cdot* [c]\)”, and a set of error traces “A request have been made (\( r \)) but is not acknowledged (\( \neg\alpha \) forever” is written as “\([i] \cdot* [r] \cdot* [\neg\alpha]\)\(\cdot\omega\)”.

### 2.2 Path set expression model checking

When a set of possible error traces on FSM \( M \) is given by PSE \( \alpha \), model checking is done by finding a state transition sequence on \( M \) matched by \( \alpha \), or by proving that no sequence on \( M \) matches \( \alpha \). It is similar to a problem of language containment between two finite automata \([8, 9]\), except that the PSE represents the complement of the property automaton. The language containment algorithm checks the product automaton of \( M \) and the property automaton. For model checking on a PSE, however, no product machine is constructed and the original machine \( M \) is traversed along the PSE.

We define two functions \( bw(\alpha) \) and \( fw(\alpha) \) to carry out model checking. They are functions from a PSE to a formula representing a set of states. \( bw(\alpha) \) represents the set of start states of sequences matched by \( \alpha \). \( fw(\alpha) \) represents the union of the set of final states of finite sequences matched by \( \alpha \) and the set of states visited infinitely often along infinite sequences matched by \( \alpha \). Intuitively, \( bw(\alpha) \) evaluation means backward traversal along \( \alpha \) and \( fw(\alpha) \) evaluation means forward traversal along \( \alpha \). Each result is not a constant false value representing an empty set if and only if one or more sequence on \( M \) is matched by \( \alpha \). Checking \( bw(\alpha) = false \) is called backward model checking on \( \alpha \), and checking \( fw(\alpha) = false \) is called forward model checking on \( \alpha \). Moreover, when \( \alpha \) is in the form of \( \alpha_1 : \alpha_2 \), we can also check it by evaluating \( \alpha \) from both sides: \( fw(\alpha_1) \land bw(\alpha_2) = false \).

### 2.3 Backward evaluation procedure

Backward PSE evaluation is based on traditional CTL model checking techniques \([2, 6, 7]\). The following equations show the procedure to convert a PSE into a CTL formula using operators \( \text{EX} \), \( \text{EU} \), and \( \text{EG} \) (assuming that \( p \) and \( q \) represent sets of states and \( \alpha \) and \( \beta \) are sub-expressions of a PSE):

\[
\begin{align*}
bw([p]) &= p \\
bw([p] \cdot* [q]) &= E(p \cup q)
\end{align*}
\]

\[
\begin{align*}
bw([p]^{\omega}) &= \text{EG} \ p \\
bw([p] \cdot [\alpha]) &= p \land \text{EX} \ bw(\alpha) \\
bw([p] \cdot* [q] \cdot [\alpha]) &= bw([p] \cdot [bw([q] \cdot [\alpha])]) \\
bw([p] \cdot [q] \cdot [\alpha]) &= bw([p] \cdot [bw([q] \cdot [\alpha])]) \\
bw(\alpha + \beta) &= bw(\alpha) \lor bw(\beta).
\end{align*}
\]

### 2.4 Forward evaluation procedure

We regard \( fw([p] \cdot [q]) \), \( fw([p] \cdot [q] \cdot [r]) \), and \( fw([p] \cdot [q]^{\omega}) \) as basic operations of forward evaluation. Their details are shown later in Section 3. Using them, forward evaluation of a PSE is given as follows (assuming that \( p \) and \( q \) represent sets of states and \( \alpha \) and \( \beta \) are sub-expressions of a PSE):

\[
\begin{align*}
fw([p]) &= p \\
fw([p] \cdot [q]) &= q \\
fw([p]^{\omega}) &= fw([p] \cdot [p]^{\omega}) \\
fw(\alpha) &= fw([fw(\alpha)] \cdot [p]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha) \cdot q] \cdot [p]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha) \cdot q] \cdot [p]) \\
fw(\alpha) \cdot [p] &= fw(\alpha) \land p \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha) \cdot q] \cdot [p]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q]) \\
fw(\alpha) \cdot [q] &= fw([fw(\alpha)] \cdot [p] \cdot [q])
\end{align*}
\]

### 2.5 Backward versus forward model checking

In general, backward model checking may be better than forward model checking for some examples, and vice versa. In most actual examples of PSE model checking, however, we observe that forward model checking is more efficient than backward model checking. It is remarkable for PSEs ending with an \( \omega \) operator, which are in common use. Let us consider model checking on \([p] \cdot [q]^{\omega}\) as an example. The backward method first looks for all cycles in the entire space of \( q \) and then checks their reachability from \( p \) through \( q \), while the forward method looks for cycles only in the restricted space of \( q \) that are reachable from \( p \) through \( q \). It makes forward model checking more efficient than backward model checking, on the assumption that the size of search space affects the computational cost. Although this assumption is not always true for symbolic state traversal, it is practically true because the computation is usually simplified with don’t-care information of the search space.

### 3 Basic forward operations

#### 3.1 Forward transition evaluation

The first basic forward operation is \( fw([p] \cdot [q]) \), which represents the set of states that satisfy \( q \) and that are reach-
able with just one transition from some state satisfying $p$. We can easily translate this definition into the following rule using the image computation operator $\text{Img}$ as shown in [10]:

$$fw([p] [q]) = \text{Img}(p) \land q.$$  \hfill (1)

### 3.2 Forward reachability evaluation

The second operation is $fw([p] [q]^* [r])$, which represents the set of states that satisfy $r$ and that are reachable from some state satisfying $p$ with zero or more transitions only through states satisfying $q$. The following procedure is essentially the same as $\text{FwdUntil}(p, q) \land r$ proposed in [10]. It is computed by forward state traversal procedure starting from $p$:

$$Q_i = \begin{cases} 
  p & (i = 0), \\
  \text{Img}(Q_{i-1} \land q) \land \neg \bigvee_{j=0}^{i-1} Q_j & (i \geq 1).
\end{cases}$$  \hfill (2)

The image computation is repeated until $Q_i = false$. Assuming $Q_n = false$, the result of the operation is given as follows:

$$fw([p] [q]^* [r]) = r_0 \lor \cdots \lor r_{n-1},$$  \hfill (3)

$$r_i = Q_i \land r.$$  \hfill (4)

When we are checking $fw([p] [q]^* [r]) = false$ rather than exactly evaluating $fw([p] [q]^* [r])$, we do not always need to repeat the image computation $n$ times. The procedure can be terminated as soon as we find some $r_i \neq false$.

### 3.3 Forward cycle evaluation

The final basic operation is $fw([p] [q]^\omega)$, which represents the set of states visited infinitely often along infinite sequences matched by $[p] [q]^\omega$. Actually, we do not need exact computation of $fw([p] [q]^\omega)$ and only need to check $fw([p] [q]^\omega) = false$ (there is no infinite sequence matched by $[p] [q]^\omega$) in forward model checking, because an ‘$\omega$’ operator appears only at the end of concatenation.

Procedure $\text{FwdGlobal}(p, q)$ proposed in [10] can be used to check $fw([p] [q]^\omega) = false$. Here is an alternative procedure, which is more efficient than $\text{FwdGlobal}(p, q)$. We compute $Q_0, Q_1, \ldots$ as follows:

$$Q_i = \begin{cases} 
  p \land q & (i = 0), \\
  \text{Img}(Q_{i-1} \land q) \land q & (i \geq 1).
\end{cases}$$  \hfill (5)

The image computation is repeated until we find either cases listed below:

**Case 1:** $\exists n \geq 0, \ Q_n = false.$  \hfill (6)

**Case 2:** $\exists m \geq 0, \ \exists n \geq m + 1, \ Q_m \neq false,$

$$Q_m \land \neg \bigvee_{k=m+1}^{n} Q_k = false.$$  \hfill (7)

$fw([p] [q]^\omega) = false$ in the first case and $fw([p] [q]^\omega) \neq false$ in the second case. Assuming the first case, every path from $p \land q$ through $q$ has finite length that is $n$ state transitions or less. Thus, there is no sequence matched by $[p] [q]^\omega$. Assuming the second case, every state in $Q_m$ is also included in one of $Q_{m+1}, \ldots, Q_n$. It means that every state in $Q_m$ has a backward path to some state in $Q_m$. We can repeat retraction such paths and visit $Q_m$ infinitely often. Since the state space is finite, we eventually visit some state $s$ in $Q_m$ twice. Thus, there is a cycle from $s$ to $s$ along which $q$ keeps holding, and some infinite sequence including the cycle is matched by $[p] [q]^\omega$.

### 4 Modifying the order of state traversal

#### 4.1 Partitioned state traversal

Forward model checking is basically achieved by a series of image computations, which means breadth-first state traversal. When the set of start states in a PSE is partitioned into subsets, we can use a mixture manner of breadth-first and depth-first traversals.

Let $\beta$ be the PSE that includes $\alpha$ as a prefix. Assuming that $fw([\alpha]) = a_1 \lor \cdots \lor a_n, \ fw([\beta])$ can be evaluated with $n$ state traversals starting from $a_1, \ldots, a_n$. When $\alpha = \beta = [p] [q]^* [r]$, equation (3) gives a reasonable partitioning of $fw([\alpha])$ since it represents grouping under similarities based on distance from $p$. We can use it for evaluating the PSE that includes $[p] [q]^* [r]$ as a prefix. For example:

$$fw([p] [q]^* [r] [s]) = fw([r_0 \lor \cdots \lor r_{n-1}) [s])$$

We generalize it for each basic forward operations. Assuming that $p = p_1 \lor \cdots \lor p_m, \ fw([p] [q])$ is divided simply as follows:

$$fw([p] [q]) = \bigvee_{i=1}^{n} fw([p_i] [q]).$$  \hfill (8)

For $fw([p] [q]^* [r])$ and $fw([p] [q]^\omega)$, the first subsets are given by $fw([p_1] [q]^* [r])$ and $fw([p_1] [q]^\omega)$ respectively. Subsequently, we do not need to traverse the paths that have been traversed once in the previous computations:

$$fw([p] [q]^* [r]) = \bigvee_{i=1}^{n} fw([p_i] [q \land \neg t_{i-1}]^* [r]).$$  \hfill (9)

$$fw([p] [q]^\omega) = \bigvee_{i=1}^{n} fw([p_i] [q \land \neg t_{i-1}]^\omega).$$  \hfill (10)

$t_i$ represents the subset of $q$ composed of the states that have already been visited on and before the $i$-th sub-computation:

$$t_i = \begin{cases} 
  false & (i = 0), \\
  t_{i-1} \lor fw([p_i] [q \land \neg t_{i-1}]^* [q]) & (i \geq 1).
\end{cases}$$  \hfill (11)
Computation of $t_i$ is not a large overhead since we can reuse the $i$-th intermediate result of (8) or (9).

### 4.2 Layered state traversal

Let $\beta$ be the PSE that includes $\alpha$ as a prefix and $fw(\alpha) = a_1 \lor \cdots \lor a_n$. We get a subset of $fw(\beta)$ as a result of the state traversal from each $a_i$, which is possibly partitioned as $b_i = b_{i,1} \lor \cdots \lor b_{i,m_i}$. The order of partitioned state traversal is written as follows:

$$a_1, \ldots, a_n, (b_{1,1}, \ldots, b_{1,m_1}), \ldots, (b_{n,1}, \ldots, b_{n,m_n}).$$

If we are checking $fw(\beta) = false$, the computation can be terminated as soon as some $b_{i,j} \neq false$ is found. Thus, we optimize the order of state traversal to get $b_{i,j}$ earlier, by interleaving $fw(\alpha)$ and $fw(\beta)$ evaluations:

$$a_1, (b_{1,1}, \ldots, b_{1,m_1}), \ldots, a_n, (b_{n,1}, \ldots, b_{n,m_n}).$$

We also modify the order of state traversal recursively on every prefix. As a result, model checking is achieved in layered manner, which is mixture of breadth-first and depth-first traversal over the PSE.

We use $r_0, \ldots, r_{n-1}$ in equation (3) to make partitions. The first subset $r_0$ can be computed in small CPU time compared with that for computing all subsets. We can dramatically reduce CPU time for completing the first several partial evaluations over the PSE, which gives us a chance to notice a design error quickly.

### 5 Experimental results

Table 1 summarizes benchmark examples. They are based on industrial circuits and have been used for verification of the circuits. We checked one actual property for each model except $\text{dh}_2$. For $\text{dh}_2$, we checked one failing property and one passing property. $\text{pipe}_s2$ ($\text{pipe}_d2$) is a revised version of $\text{pipe}_s1$ ($\text{pipe}_d1$). We used the same property for each version. The property passes on the later version, while it fails on the earlier version.

<table>
<thead>
<tr>
<th>Model</th>
<th>#FFs</th>
<th>#States</th>
<th>Depth</th>
<th>Description</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{atm}_s$</td>
<td>54</td>
<td>2.0 x 10^3</td>
<td>126</td>
<td>ATM-switch [11]</td>
<td>$[\beta] \cdot [r_0] \cdot [\neg \alpha] = 0, \text{fail}$</td>
</tr>
<tr>
<td>$\text{dh}_1$</td>
<td>46</td>
<td>4.0 x 10^3</td>
<td>21</td>
<td>BUS protocol</td>
<td>$[\beta] \cdot [r_0] \cdot [\neg \alpha] = 0, \text{fail}$</td>
</tr>
<tr>
<td>$\text{dh}_2$</td>
<td>66</td>
<td>7.9 x 10^3</td>
<td>17</td>
<td>Cache coherence</td>
<td>$[\beta] \cdot [r_0] \cdot [\neg \alpha] = 0, \text{pass}$</td>
</tr>
<tr>
<td>$\text{dh}_3$</td>
<td>96</td>
<td>3.7 x 10^3</td>
<td>40</td>
<td>Cache coherence</td>
<td>$[\beta] \cdot [r_0] \cdot [\neg \alpha] = 0, \text{fail}$</td>
</tr>
<tr>
<td>$\text{vpp}$</td>
<td>101</td>
<td>2.4 x 10^3</td>
<td>18</td>
<td>VLIW pipeline</td>
<td>$[\beta] \cdot [r_0] \cdot [\neg \alpha] = 0, \text{fail}$</td>
</tr>
<tr>
<td>$\text{pipe}_s1$</td>
<td>35</td>
<td>4.4 x 10^3</td>
<td>11</td>
<td>Superscalar pipeline</td>
<td>$[\beta] \cdot [r_0] \cdot [\neg \alpha] = 0, \text{fail}$</td>
</tr>
<tr>
<td>$\text{pipe}_s2$</td>
<td>35</td>
<td>4.5 x 10^3</td>
<td>11</td>
<td>Superscalar pipeline</td>
<td>$[\beta] \cdot [r_0] \cdot [\neg \alpha] = 0, \text{pass}$</td>
</tr>
<tr>
<td>$\text{pipe}_d1$</td>
<td>73</td>
<td>5.7 x 10^3</td>
<td>11</td>
<td>Superscalar pipeline</td>
<td>$[\beta] \cdot [r_0] \cdot [\neg \alpha] = 0, \text{fail}$</td>
</tr>
<tr>
<td>$\text{pipe}_d2$</td>
<td>73</td>
<td>5.7 x 10^3</td>
<td>11</td>
<td>Superscalar pipeline</td>
<td>$[\beta] \cdot [r_0] \cdot [\neg \alpha] = 0, \text{pass}$</td>
</tr>
</tbody>
</table>

**Table 1: Benchmark examples**

<table>
<thead>
<tr>
<th>Model</th>
<th>VIS1.2</th>
<th>BINGO1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CTL</td>
<td>FW</td>
</tr>
<tr>
<td>$\text{atm}_s$</td>
<td>420 sec</td>
<td>191 img</td>
</tr>
<tr>
<td>$\text{dh}_1$</td>
<td>8 sec</td>
<td>32 img</td>
</tr>
<tr>
<td>$\text{dh}_2$</td>
<td>&gt;500 MB</td>
<td>&gt;500 MB</td>
</tr>
<tr>
<td>$\text{vpp}$</td>
<td>&gt;500 MB</td>
<td>29 img</td>
</tr>
<tr>
<td>$\text{pipe}_s1$</td>
<td>10 sec</td>
<td>27 img</td>
</tr>
<tr>
<td>$\text{pipe}_d1$</td>
<td>&gt;500 MB</td>
<td>25 img</td>
</tr>
</tbody>
</table>

**Table 2: Results for failing properties**

<table>
<thead>
<tr>
<th>Model</th>
<th>VIS1.2</th>
<th>BINGO1.4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CTL</td>
<td>FW</td>
</tr>
<tr>
<td>$\text{atm}_s$</td>
<td>420 sec</td>
<td>191 img</td>
</tr>
<tr>
<td>$\text{dh}_1$</td>
<td>8 sec</td>
<td>32 img</td>
</tr>
<tr>
<td>$\text{dh}_2$</td>
<td>&gt;500 MB</td>
<td>&gt;500 MB</td>
</tr>
<tr>
<td>$\text{vpp}$</td>
<td>&gt;500 MB</td>
<td>29 img</td>
</tr>
<tr>
<td>$\text{pipe}_s1$</td>
<td>10 sec</td>
<td>27 img</td>
</tr>
<tr>
<td>$\text{pipe}_d1$</td>
<td>&gt;500 MB</td>
<td>25 img</td>
</tr>
</tbody>
</table>

**Table 3: Results for passing properties**
The results are shown in Table 2 and Table 3. Standard CTL model checking (labeled ‘CTL’), standard forward model checking (labeled ‘FW’), and forward model checking with layered state traversal algorithm (labeled ‘FW+L’) were tested on BINGO [10]. Rows in each box show the total number of image computations (including pre-image computations in ‘CTL’), CPU time, and memory usage. We also experimented with VIS [12] version 1.2 for reference. Execution parameters for VIS were set based on one of its standard script, ‘script_model_check.robust’. We provided a good initial BDD variable order for each benchmark, and turned off dynamic variable reordering of BINGO and VIS to measure basic performance of the algorithms. All benchmarks were run on a 167MHz Sun Ultra 1 workstation under a memory limit of 500 megabytes.

Table 2 indicates that layered traversal is very efficient in average for failing properties. We obtained one-figure improvements in CPU time and memory usage for large examples. Although the efficiency of layered traversal should depend on the design error, actual design errors have been found effectively in our all examples. Table 3 indicates that layered traversal is not far inferior to the ordinary breadth-first state traversal for passing properties. CPU time was somewhat increased; however, memory usage was comparable. It means that we can safely use the layered traversal even if we cannot expect whether the property fails.

It is generally recognized that design errors sometimes lead the model to random behavior and make symbolic state traversal expensive. The ‘FW’ results of pipe\textsubscript{d1} and pipe\textsubscript{d2} show such a situation. pipe\textsubscript{d1} has a design error and it is more expensive to check than pipe\textsubscript{d2}. The problem was avoided in ‘FW+L’ because the error was found without searching the entire space.

6 Conclusions
We have presented model checking techniques for early design stages at which design errors make many properties fail. They are based on symbolic forward state traversal along path set expressions in a layered manner. Although the techniques do not bring about speedups for finding all theoretical bugs, they dramatically improve average CPU time for finding actual bugs. The failing property has to be checked again and again to analyze the bug until it is corrected. The techniques, therefore, can have significant impacts on actual verification cycles. Since they do not increase memory usage even if the property passes, we can always use them safely. We have a chance to find design error without searching the entire space. It sometimes avoids a memory explosion problem on model checking of buggy design.

While the expressive power of path set expressions is now limited to keep consistency with CTL model checking, syntactic extension would be possible in future. Layered state traversal algorithm repeats partial state traversal over a path set expression incrementally. It would be also interesting to consider layered state traversal algorithm in the context of incremental refinement or automatic abstraction.

Acknowledgments
We would like to thank Randal E. Bryant, Edmund M. Clarke, and Fumiyasu Hirose for valuable discussions.

References