Binary Time-Frame Expansion

Farzan Fallah
Fujitsu Labs. of America
Sunnyvale, CA 94085

Abstract- This paper introduces a new method for performing time-frame expansion based on writing the number of time frames in terms of powers of two. In the proposed method, the behavior of a circuit for \( t \) time frames, where \( 0 \leq t < n \) is modeled by unrolling the circuit \( 2^0, 2^1, 2^2, \ldots, 2^{\log_2 n} \) times and combining them. This formulation of the problem makes it possible to prune the search space quickly when the problem is infeasible. To show the advantage of this method, we have used it to model the state justification problem and solve the problem using a SAT-solver. Experimental results show several orders of magnitude speedup for some non-trivial infeasible problems. Furthermore, in most cases the CPU time requirement grows linearly in terms of the number of time frames.

1 Introduction

Many existing algorithms in test and verification can only work on combinational circuits where the output of the circuit is only a function of its inputs, which can be easily controlled. This makes it impossible to use such algorithms on sequential circuits directly. The common method to handle sequential circuits using such algorithms is time-frame expansion which was introduced several decades ago and has been widely used without any improvement since then [1]. In time-frame expansion method an iterative logic array is constructed and the problem is solved on it. An iterative logic array is a combinational circuit that mimics the functionality of the sequential circuit for a given number of time frames or clock cycles. The shortcoming of this approach is that the iterative logic array models the behavior of the sequential circuit for an exact number of cycles. This makes it necessary to perform time-frame expansion for one, two, and more cycles and solve the problem for each of the resulting iterative logic arrays or sub-problems separately. Solving the sub-problems separately does not permit to share the computation of one sub-problem with another one. This paper presents a new method for performing time-frame expansion, namely binary time-frame expansion. In the proposed method, instead of unrolling a sequential circuit for a given number of times, the behavior of the circuit is modeled for all time frames less than a given number. This effectively integrates the number of time frames (i.e., time) as a new variable into the problem. Furthermore, it enables searching the time space (i.e., the number of time frames) in a binary fashion, resulting in significant speed-ups in many cases.

The rest of this paper is organized as follows. In Section 2 we look at previous work addressing this problem. Section 3 explains our binary time-frame expansion method. In Section 4, we describe the formulation of the state justification problem as a Boolean satisfiability problem. Experimental results are presented in Section 5. The conclusions and future work is presented in Section 6.

2 Previous Work

This section explains conventional time-frame expansion and some of its variations. In the conventional method, the sequential circuit is unrolled to construct an iterative logic array.

Figure 1 An iterative logic array.

Figure 1 shows an example of an iterative logic array constructed from a sequential circuit. An iterative logic array mimics the behavior of a sequential circuit for exactly a given number of time frames. Unfortunately, in many cases the number of time frames is not known a priori. As an example, suppose we are interested in discovering if a condition on some signals ever happens in a circuit. Clearly, the number of time frames is not known. In
such a case, an iterative algorithm is used to unroll the sequential circuit 1, 2, 3, ..., t times. Assuming $f^m$ denotes a sequential circuit unrolled m times, in the conventional time-frame expansion $f^1, f^2, f^3, ..., f^t$ are generated, where t is an arbitrary bound.\footnote{There is an exponential upper bound in terms of the number of latches of a circuit for the required number of time frames, but this theoretical bound is too loose to be of any practical use.}

Figure 2 shows the iterative algorithm used for performing time-frame expansion. The algorithm effectively performs a linear search on the number of time frames and runs function FUNC once per iteration for each generated iterative logic array or monitors the last time frame of the iterative logic array. This linear search on the number of time frames makes the algorithm inefficient. Furthermore, depending on the problem, it may be possible to skip some iteration based on the results gathered in the previous iterations, but the above algorithm does not provide such an opportunity. The algorithm presented in Figure 2 has been used in many applications to handle sequential circuits [2,3,4,5,6,7]. It is possible to merge all iterations of the loop in Figure 2 into one by constructing an iterative logic array for t time frames and solving the problem for all time frames less than t together. In other words, instead of monitoring only the last time frame of the iterative logic array, all time frames are monitored.\footnote{If there is a reset line, it will suffice to monitor the last time-frame. Checking all time frames will be still correct, but it will increase the number of solutions if the problem is feasible.}

This formulation will help to decrease the CPU time of setting up the problem. It can also solve the problem faster than other versions of the conventional time-frame expansion, but it still suffers from the fact that it uses a linear search on the number of time frames. We will give an example of this method in Section 4.

As another variation of Figure 2 algorithm, in each iteration FUNC may be run on time frames m-k to m, where k is an arbitrary number and m is incremented by k after each iteration.

Burch et al. \cite{13} introduced iterative squaring for symbolic model checking. Iterative squaring has some similarities with our method and can exponentially decrease the number of necessary iterations to reach a fixed-point, but as they acknowledge it is impractical for large designs due to the large size of intermediate BDDs.

3 Binary Time-Frame Expansion

Our method is explained in this section. Let t be an n-bit number $[t_n \ldots t_0]$. Then, it can be written as follows,

\[ t = \sum_{i=0}^{n-1} t_i \times 2^i \]

and $f^t$ can be written as,

\[ f^t = f^{0 \leq t \leq n} = f^{t_0 \times 2^0} \times f^{t_1 \times 2^1} \times \cdots \times f^{t_n \times 2^n} \]

where the cross operator is defined as follows,

\[ f^{m+n} = f^m \times f^n \]

We call the above formula the expansion equation. Note that,

\[ f^{t_i \times 2^j} = \begin{cases} f^2 & \text{if } t_i = 1 \\ 1 & \text{if } t_i = 0 \end{cases} \]

which may be modeled using 2 copies of $f$ and a multiplexer as depicted in Figure 3. Hence, the behavior of the circuit for all time frames less than or equal to t may be modeled by every term in the expansion equation as shown in Figure 4. This formulation has two advantages,

1- It enables to search the number of time frames in a binary fashion. In other words, instead of linearly searching a value for $t$ in $f^t$, the search is done on
the bits of \( t \). This makes the computation faster in many cases.

2. The extra clauses it needs to model the problem grows in a logarithmic fashion with \( t \). This is because of the logarithmic growth of the number of MUXes. In comparison, in the fastest version of the conventional time-frame expansion (i.e., when all time frames are monitored), the growth of the extra clauses is linear in \( t \). As a result, the new formulation needs smaller number of clauses, but the difference is not significant.

To illustrate the advantage of using this model consider the problem of finding the number of time frames necessary to set the MSB of an 8-bit counter to 1, assuming the initial value of the counter is 0. The conventional time-frame expansion proceeds by unrolling the counter once. Clearly, it is not possible to set the MSB to 1 in one time frame. The algorithm continues with unrolling the counter 2 and more times. After 128 trials, the algorithm finds a solution to the problem. At each iteration except the last one, the algorithm proves that the problem is infeasible. On the contrast, using binary time-frame expansion, in one step it can be proved that for all values of \( t \) whose eight bit is zero, the problem is infeasible. This enables to prune a large number of time frames rapidly.\(^3\)

Note that in presence of a reset line, our formulation of the problem increases the number of solutions for a feasible problem. The reason is that it is possible to have more than one value for \( t \). In this case, the actual number of time frames can be easily found from the value of \( t \) and the information about the time-frame whose reset line has been activated. As an example, consider a problem that can be solved in 5 time frames. If we use binary time-frame expansion formulation to model all time frames less than 8, one solution of the problem corresponds to \( t=5 \). Another solution corresponds to the case when \( t=7 \) and the reset line of the second time-frame is active. In this case the actual number of time frames (i.e., 5) can be computed by finding how many time frames are present between the activated reset line and the monitor.

4 SAT Formulation of State Justification Problem

Given a sequential circuit, \( C \), the problem of finding a sequence of vectors that takes the state of \( C \) from an initial value to a desired one is called state justification problem. State justification is a fundamental problem in test and validation. As another version of the problem, one might be interested to prove that a state is not reachable within a number of clock cycles or it is not reachable at all. Because state justification is an NP-Complete problem, it is not possible to solve the problem for very large and deep sequential circuits. In this section we explain how state justification can be formulated as a Boolean satisfiability problem. First, we give a brief description on how a circuit can be modeled using Boolean expressions. More complete description may be found in [8]. Consider the circuit of Figure 5. Clauses may be written for each gate in the circuit to model the input-output relationship of the gate. The AND gate can be modeled by,

\[
(x \lor z) \land (y \lor z) \land (\neg x \lor \neg y \lor z)
\]

and the Inverter can be modeled using,

\[
(x \lor y) \land (\neg x \lor \neg y).
\]

The above method was used by Larrabee to generate stuck-at fault test vectors using a SAT solver [8]. We use this method to model a sequential circuit. We set the values of the latches in the first time frame to the initial values and check the values of the latches in the last time frame against the desired values. Instead of intermediate latches, buffers are used. In the conventional method, we use the iterative algorithm depicted in Figure 2 to justify a state. As a variant of this approach, we check the values of the latches of all time frames against the desired values. This approach is not iterative. Yet, as another variant, we increment the number of time frames by \( k \) at each iteration and check the values of the latches of the last \( k \) time frames against the desired values. In the next section, we compare the performance of binary time-frame expansion against the conventional method and its above variants.

5 Experimental Results

This section provides the experimental results of using our method to justify a random state for sequential circuits in ISCAS89 benchmark suite [9]. We unrolled the circuits until the random state was justified or 4096 frames were reached or the number of variables or clauses in the SAT problem was more than the allowed limit by the SAT solver (we cascaded one copy of the circuits with the model in Figure 4 to search \( 1 \leq t \leq n \) space). This gave a series

---

\(^3\) This example has been used just to illustrate the idea behind binary time-frame expansion method. In practice, this example is solved quickly by both methods.
of problems corresponding to different number of time frames for each circuit in the benchmark. We used our binary time-frame expansion method and the conventional one where the latches of all frames are compared against the random state. To make it easy to compare the results, we used equal number of time frames in both cases. We used the CHAFF SAT solver [10] running on a SUN ULTRA80 dual processor 450 MHz workstation with 2GB memory to solve the satisfiability problems. For each problem, the time limit of the solver was set to one hour. We did not provide any variable ordering for the SAT solver.

**a) General Statistics and Information about Performance of Binary Time-Frame Expansion**

Table 1 gives some statistics. Out of 311 problems, 8 problems were satisfiable. Our binary time-frame expansion method failed in 13 problems (corresponding to 4 circuits) given the one-hour time limit. In comparison, the fastest version of the old method failed in 33 cases (corresponding to 17 circuits, some of them having small number of latches, e.g., S289 with 14 latches). Our method was able to handle all problems that the old one could handle. This showed that our method was more robust than the old one. Total CPU time of our approach for all examples excluding the time-out ones were 7,934 seconds. The corresponding number for the old approach was 52,886 seconds. Out of 35 circuits in ISCAS89 benchmark, for 26 circuits the CPU time grew linearly while increasing the number of time frames. The corresponding number for the old method was 7, all of them were trivial examples. We divided the problems into two sets, trivial and non-

<table>
<thead>
<tr>
<th>Method</th>
<th># Problems</th>
<th># Sat.</th>
<th># Time Out</th>
<th>CPU Time (s)</th>
<th># Circuits</th>
<th># Linear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>311</td>
<td>8</td>
<td>13</td>
<td>7,934</td>
<td>35</td>
<td>26</td>
</tr>
<tr>
<td>Conventional</td>
<td>311</td>
<td>8</td>
<td>33</td>
<td>52,886</td>
<td>35</td>
<td>7</td>
</tr>
</tbody>
</table>

**Table 1 Comparison of two approaches.**

![Figure 6 Distribution of ratio of CPU times for non-trivial examples.](image)

![Figure 7 Distribution of ratio of CPU times for trivial examples.](image)

The trivial set included all examples that could be solved by just using implications. The non-trivial ones were the ones that could be solved only if branching was used. About 53% of the problems were non-trivial and the rest were trivial. Figure 6 shows the distribution of ratio of CPU times for non-trivial examples. As one can see in most cases binary time-frame expansion outperforms the conventional one. For example, for about 19% of total problems the new method was between 1 to 10 times faster than the old one. In some cases our method was several orders of magnitude faster than the old one. For non-trivial problems, our method was at worst 3x slower than the old approach. Figure 7 shows the distribution for trivial problems. For these problems, the old method is almost always better than our method, but the difference in speed is at most 3x (i.e., there is no exponential difference). Note that for trivial problems the infeasibility can be proved by using implications; there is no need to branch on variables. On the other hand when we use binary time-frame expansion, we need to branch on control variables of MUXes. As a result it takes more time to prove infeasibility, but the required time does not grow exponentially. As one can see by using binary time-frame expansion method, in most cases the infeasibility of problems can be proved very quickly.

**b) Details of Performance for Satisfiable Problems**

There is no significant difference between binary time-frame expansion and the conventional one if the problem is feasible. Table 2 gives CPU time and some information about 16 satisfiable instances. The examples ending with letter ‘b’ and ‘c’ have been ignored instances solved in less than 0.01 second to emphasize hard instances.
Comparison of two methods for satisfiable problems.

<table>
<thead>
<tr>
<th>Example</th>
<th>Frames</th>
<th>Time</th>
<th>Max-Dec-Level</th>
<th>Decisions</th>
<th>Original Clauses</th>
<th>Original Literals</th>
<th># Implications</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1488b</td>
<td>32</td>
<td>6.45</td>
<td>236</td>
<td>11,294</td>
<td>59,132</td>
<td>141,860</td>
<td>3,110,248</td>
</tr>
<tr>
<td>s1488c</td>
<td>32</td>
<td>11.86</td>
<td>227</td>
<td>16,759</td>
<td>59,291</td>
<td>142,254</td>
<td>5,412,321</td>
</tr>
<tr>
<td>s1494b</td>
<td>32</td>
<td>2.01</td>
<td>240</td>
<td>4,370</td>
<td>60,028</td>
<td>144,036</td>
<td>938,220</td>
</tr>
<tr>
<td>s1494c</td>
<td>32</td>
<td>11.48</td>
<td>235</td>
<td>14,706</td>
<td>60,187</td>
<td>144,430</td>
<td>5,591,514</td>
</tr>
<tr>
<td>s208.1b</td>
<td>256</td>
<td>1452.50</td>
<td>2,462</td>
<td>9,785,234</td>
<td>57,728</td>
<td>135,408</td>
<td>28,444,346</td>
</tr>
<tr>
<td>s208.1c</td>
<td>256</td>
<td>1373.98</td>
<td>2,435</td>
<td>9,005,731</td>
<td>59,897</td>
<td>141,544</td>
<td>27,206,958</td>
</tr>
<tr>
<td>s27b</td>
<td>2</td>
<td>0.01</td>
<td>11</td>
<td>12</td>
<td>70</td>
<td>162</td>
<td>33</td>
</tr>
<tr>
<td>s27c</td>
<td>2</td>
<td>0.00</td>
<td>10</td>
<td>11</td>
<td>70</td>
<td>157</td>
<td>34</td>
</tr>
<tr>
<td>s386b</td>
<td>8</td>
<td>0.02</td>
<td>38</td>
<td>132</td>
<td>3,524</td>
<td>8,612</td>
<td>8,996</td>
</tr>
<tr>
<td>s386c</td>
<td>8</td>
<td>0.04</td>
<td>37</td>
<td>213</td>
<td>3,539</td>
<td>8,622</td>
<td>17,533</td>
</tr>
<tr>
<td>s510b</td>
<td>8</td>
<td>0.04</td>
<td>175</td>
<td>249</td>
<td>4,700</td>
<td>11,164</td>
<td>12,805</td>
</tr>
<tr>
<td>s510c</td>
<td>8</td>
<td>0.02</td>
<td>171</td>
<td>188</td>
<td>4,715</td>
<td>11,174</td>
<td>5,983</td>
</tr>
<tr>
<td>s820b</td>
<td>4</td>
<td>0.01</td>
<td>65</td>
<td>79</td>
<td>3,980</td>
<td>9,862</td>
<td>2,829</td>
</tr>
<tr>
<td>s820c</td>
<td>4</td>
<td>0.01</td>
<td>65</td>
<td>80</td>
<td>3,980</td>
<td>9,845</td>
<td>3,636</td>
</tr>
<tr>
<td>s832b</td>
<td>4</td>
<td>0.01</td>
<td>65</td>
<td>79</td>
<td>4,084</td>
<td>10,126</td>
<td>3,094</td>
</tr>
<tr>
<td>s832c</td>
<td>4</td>
<td>0.01</td>
<td>65</td>
<td>80</td>
<td>4,084</td>
<td>10,109</td>
<td>3,692</td>
</tr>
</tbody>
</table>

Table 2 Comparison of two methods for satisfiable problems.

Formulated using binary and conventional methods, respectively. As one can see the difference between two formulations is negligible. This may be due to the fact that the branching heuristic in SAT solvers usually guides the search to parts of the space that is likely to have a solution. Hence, the portions with no solution will be by passed quickly in both formulations.

c) Effect of Increasing the Number of Time Frames on CPU Time

Figure 8 shows the CPU time growth for several circuits when increasing the number of time frames. For three circuits namely S13207, S35932, and S38417, the CPU time grows linearly in terms of the number of time frames. This resulted in small CPU time for a large circuit like S35932 with 1728 latches. For some circuits like S298, the CPU time growth was not linear. As the last example, we have presented S1423 for which the CPU time requirement grows rapidly and surpasses one-hour allocated time.

d) Detailed Comparison of Three Methods

Finally, Figure 9 shows CPU time of using three different methods for time-frame expansion for circuit S1196. The lower curve corresponds to binary time-frame expansion method. The curve plotted with squares corresponds to the conventional time-frame expansion method when all time frames are monitored. The curve plotted with triangles corresponds to the conventional time-frame expansion method when only the last 100 time frames are monitored. Note that this is an iterative algorithm and Figure 9 shows the CPU time of each iteration separately. To compute the total CPU time using this method, sum of the CPU time of all iterations (i.e., points on the graph) preceding the desired point has to be computed. As one can see, the CPU time of each iteration in this method is almost equal to the CPU time for the case that all time frames are monitored. As a result, the latter is much more efficient. The total CPU time of the former method decreases if the number of time frames monitored in each iteration is increased. As one can see binary time-frame expansion method is about two orders of magnitude faster than other methods. In fact, many points of the two top curves correspond to problems that could not be solved within one hour time limit. In the conventional methods, increasing the number of time frames decreases the CPU time after a certain point. Also, there is a sudden drop in the CPU time when using the iterative algorithm. These phenomena occur due to the heuristic nature of the variable ordering algorithm used in the SAT solver and do not happen for all circuits.

6 Conclusions and Future Work

In this paper we have presented the idea of binary time-frame expansion and used it to prove that a given state is not justifiable within a number of time frames. The experimental results showed that our method outperformed the conventional time-frame expansion by several orders of magnitude in many cases. Our experiments also showed that for many circuits the CPU time requirement grew linearly in terms of the number of time frames. We are currently trying to discover what kinds of circuits are better suited for our method and explain its linear behavior. In our experiments, we used a SAT solver to prove that a state is not justifiable. It is interesting to see the
Figure 8 CPU time growth versus the number of time frames for several circuits when using binary time-frame expansion.

Figure 9 Comparison of CPU time of three different methods.
effect of using binary time-frame expansion while using PODEM [11] or D-algorithm [12]. Finally, it remains to be seen if combining current methods into a hybrid one makes it possible to handle all circuits efficiently.

7 References


