# Direct Synthesis of Timed Asynchronous Circuits *

Sung Tae Jung and Chris J. Myers
Electrical Engineering Department
University of Utah
Salt Lake City, UT 84112

## Abstract

This paper presents a new method to synthesize timed asynchronous circuits directly from the specification without generating a state graph. The synthesis procedure begins with a deterministic graph specification with timing constraints. A timing analysis extracts the *timed concurrency* and *timed causality* relations between any two signal transitions. Then, a hazard-free implementation of the specification is synthesized by analyzing precedence graphs which are constructed by using the timed concurrency and timed causality relations. The major result of this work is that the method does not suffer from the state explosion problem, achieves significant reductions in synthesis time, and generates synthesized circuits that have nearly the same area as compared to previous timed circuit methods. In particular, this paper shows that a timed circuit — not containing circuit hazards under given timing constraints — can be found by using the relations between signal transitions of the specification. Moreover, the relations can be efficiently found using a heuristic timing analysis algorithm. By allowing significantly larger designs to be synthesized, this work is a step towards the development of high-level synthesis tools for system level asynchronous circuits.

## 1 Introduction

*Speed-independent* asynchronous circuits are very robust since they are guaranteed to work independent of the delays associated with their gates, and many synthesis methods for speed-independent circuits have been proposed [1, 2, 3, 4]. However, speed-independent circuits can be overly conservative when timing constraints are available. Methods have been proposed to use timing constraints to synthesize *timed circuits*. Such circuits work correctly under the given timing constraints [5, 6] and tend to be more efficient in area and speed than speed-independent circuits [6].

The synthesis techniques in [1, 2, 3, 4, 5, 6] have the state explosion problem because they are based on a state graph. To overcome the state explosion problem, direct methods have been proposed for speed-independent circuits [7, 8, 9, 10]. The method in [7] approximates a set of states as a cube by using a concurrency relation between transitions of the specification. It then finds an initial approximation of the implementation using these cubes. If this approximation does not satisfy correctness criteria, then iterative refinement is performed using state machine decompositions. This method is restricted to state machine decomposable specifications. The method in [8] uses an approach similar to [7] but it allows for a wider class of specifications by finding an initial approximation and refining it using STG-unfolding segments. The method in [9] constructs a characteristic graph for the given signal transition graph and generates a hazard-free implementation by finding a strongly connected subgraph. The method in [10] constructs a precedence graph for each transition of output signals and generates a hazard-free implementation by finding paths in the graph. Whereas a characteristic graph encapsulates all feasible solutions of the original STG, a precedence graph encapsulates all feasible solutions for a single transition of an output signal.

Even though several direct methods have been suggested for the synthesis of speed-independent circuits, no method has been suggested for the synthesis of timed circuits. The main goal of this work is to develop a method which generates timed asynchronous circuits for the specifications that cannot be synthesized by the previous techniques due to the large size of the state space. The solution to this problem is found by the use of timing analysis to obtain the necessary timing information directly from the specification. Timing analysis is used to determine the timed concurrency relation and timed causality relation between any two signal transitions in a circuit specification. After timing analysis, the algorithm synthesizes efficient timed circuits by constructing a precedence graph and finding all the paths in the graph in a method similar to that in [10].

This paper compares the new method to the previous methods using many benchmark examples and two parameterizable examples: SCSI and FIFO. Whereas previous methods can only synthesize 8 SCSI controllers and 5 FIFO stages, the new method can synthesize 180 SCSI controllers and 100 FIFO stages. By allowing significantly larger designs to be synthesized, this work is a step towards the development of high-level synthesis tools for system level asynchronous circuits.

## 2 Timed Specifications

Figure 1 shows a timed deterministic signal transition graph (STG) specification for a SCSI protocol controller specification [6]. In Figure 1, a node denotes a rising or falling signal transition. A transition of an input signal is underlined. An arc denotes an ordering relation between two transitions. If there is an arc from $s+$ to $t+$, $s+$ is called the *enabling* transition and $t+$ is called the *enabled* transition of the arc. A solid circle on an arc denotes a token. Each arc is associated with a timing constraint $[L, U]$, where $L$ denotes the lower bound and $U$ denotes the upper bound.

A timing constraint is said to be *satisfied* if a token has been on an arc longer than the lower bound for that arc. It is said

*This research is supported by a grant from Intel Corporation, an NSF CAREER award MIP-9625014, and a post-doctoral fellowship from the Korea Science and Engineering Foundation.

0-7803-5832-5/99/ $10.00 © 1999 IEEE

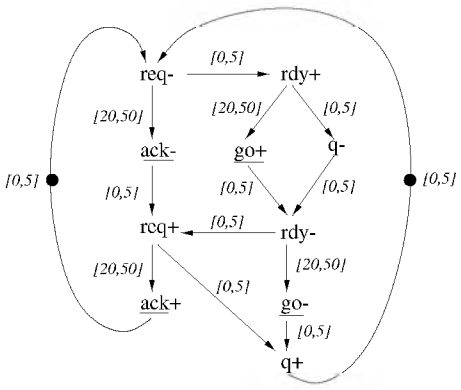


**Figure 1: The timed STG for a SCSI controller.**

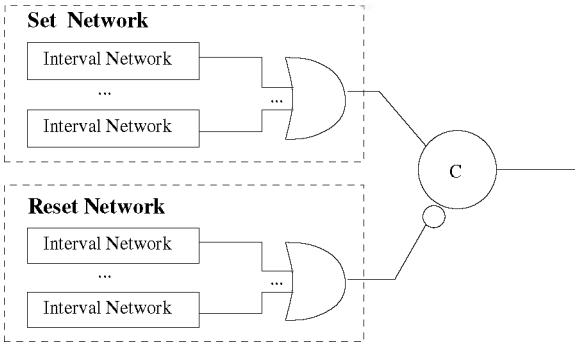


**Figure 2: Target circuit model for an output signal.**

to be *expired* if the amount of time exceeds the upper bound. A signal transition cannot occur until all the timing constraints of the input arcs are satisfied. A transition must always occur before every timing constraint on the input arcs has expired. Since a transition may be enabled by multiple transitions, it is possible that the difference in time between the firings of enabling transitions exceeds the upper bound of their timing constraints, but not for all enabling transitions. When a signal transition is fired, all the tokens on the input arcs are removed and a token is added to each output arc.

When an enabled transition is a transition of an input signal, the timing constraint can be determined from interface specifications or datapath delay estimates. When an enabled transition is a transition of a non-input signal, the timing constraint can be estimated based on the delays for the gates in the library to be used. After a circuit is generated, it should be analyzed using a timing analysis tool to verify that the timing constraints used are correct. If the circuit violates the timing constraints, it must be resynthesized with more conservative timing constraints.

In order to synthesize timed circuits, timing analysis must be applied to the specification to deduce timing information. The timing information needed is the minimum and maximum time separation between any two signal transitions in the circuit specification. For timing analysis, the synthesis procedure uses the polynomial-time heuristic algorithm in [6]. The timing analysis algorithm starts with a cyclic graph specification and unfolds the specification into an infinite acyclic graph. It then examines two finite acyclic subgraphs of the infinite graph to determine a sufficient bound on the time difference between two signal transitions.

# 3 Synthesis Procedure

Figure 2 illustrates the target circuit model of the synthesis algorithm for each output signal. The circuit is implemented as a network of basic gates such as AND gates possibly having inverted input terminals, OR gates, and C-elements. A set and a reset network is synthesized as a sum of *interval* networks as shown in the figure. Each transition of the output signal is activated by exactly one interval network. Two OR gates collect all the outputs of the interval networks to set or reset the memory element.

Let an interval, $u* \mapsto \overline{u*}$, denote the period between the time when $u*$ is enabled and the time when $\overline{u*}$, the next reverse transition of $u*$, is enabled. The interval network for the interval $u* \mapsto \overline{u*}$ must satisfy the following requirements: (i) it is turned on when $u*$ is enabled, (ii) it is turned off before $\overline{u*}$ is enabled, and (iii) once it is turned off, it remains off until $u*$ is enabled again. These requirements are the same as those in [3, 11].

The synthesis algorithm consists of four steps. First, it detects and removes redundant arcs from the specification. Second, it finds the timing relations between any two signal transitions. Third, it constructs a precedence graph for each output transition, finds all the paths in the graph, and derives a single cube circuit implementation. Fourth, it removes memory elements when possible by finding a multi-cube interval network.

## 3.1 Removing Redundant Triggers

If there are multiple enabling transitions for a signal transition, then it is possible that some of them are redundant. Each enabling transition (or *trigger signal*) results in a literal in the implementation of the signal. If a trigger signal is redundant, the corresponding literal can be removed from the implementation resulting in a smaller circuit. For the SCSI protocol controller example in Figure 1, the arc from $q-$ to $rdy-$ is found to be redundant. The worst-case time difference between the two signal transitions rdy- and q- is [15, 55]. The lower bound of this time difference, 15, is greater than the upper bound of the timing constraint on the arc, 5. Therefore, the arc (i.e., the trigger signal) is found to be redundant.

## 3.2 Finding the Relations

To directly synthesize a timed circuit, it is necessary to find the timed concurrency and timed causality relations between any two signal transitions. In order to find timed concurrent transitions, the algorithm first finds untimed concurrent transitions by reachability analysis on the STG (not the state space). Then, the algorithm checks the worst-case time difference between untimed concurrent transitions. If the lower bound is less than or equal to zero and the upper bound is greater than or equal to zero, then the two transitions are timed concurrent. For example, in the specification of the SCSI protocol controller, the two transitions $ack-$ and $go+$ are timed concurrent because they are untimed concurrent and the worst-case time difference is the bound [-35, 30]. This bound indicates that they can fire in either order. The two transitions $go+$ and $q-$ are untimed concurrent, too. However, they are not timed concurrent because

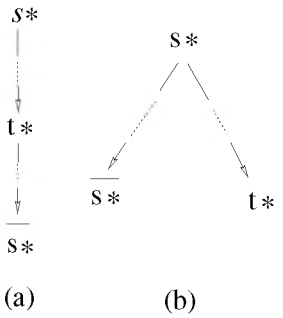


**Figure 3: Causality relation: (a) $t*$ occurs after $s*$ and before $\overline{s*}$. (b) $t*$ occurs after $s*$ and concurrently with $\overline{s*}$.**

the time difference between $go+$ and $q-$ is the bound [15, 50]. This bound means that $go+$ is always fired after $q-$ is fired.

After finding timed concurrent transitions, the algorithm finds the timed causality relations. Let $s*$ and $t*$ be transitions on two signals. If $s*$ and $t*$ have the relation shown in Figure 3 (a) or (b), then we say $s*$ causes $t*$. Here $\overline{s*}$ is the next reverse transition of $s*$. In an untimed STG specification, the causality relations are found by reachability analysis. That is, a transition $s*$ causes a transition $t*$ if $t*$ is reachable from $s*$ without visiting $\overline{s*}$. In a timed STG, the algorithm finds the timed causality relations by analyzing reachability and worst-case time differences. In the specification of the SCSI protocol controller, $go+$ is reachable from $q+$ without visiting $q-$. So, $q+$ is an untimed causal transition for $go+$. However, it is not a timed causal transition because the time difference between $q-$ and $go+$ is [-50, -15]. That is, $q-$ always occurs before $go+$. So, $q-$ timed causes $go+$.

## 3.3 Finding a Single Cube Network

In this step, the synthesis procedure synthesizes each interval network as a single cube. In [3], conditions are developed in which each interval can be implemented as a single cube in a hazard-free manner. In [11], they showed that specifications can be transformed to satisfy these conditions by inserting new signals. The algorithm described in this paper currently only handles specifications which have a single cube implementation. If there is no single cube implementation, new signals are added and the modified specification is resynthesized. For simplicity, the algorithm is presented for specifications which have only one occurrence of each signal transition per cycle. The algorithm, however, can be extended in a straightforward manner to cover the case where there are multiple occurrences of some signal transitions. The current implementation of the algorithm includes this extension.

Let's consider the synthesis procedure for the interval $u* \mapsto \overline{u*}$. The interval network is synthesized to satisfy the requirements of the target circuit model. The synthesis process starts with a minimal interval network which is an AND gate having only the non-redundant trigger signals as inputs. Figure 4 (a) shows the minimal interval networks for the SCSI controller.

All the trigger signals go high when $u*$ is enabled, so requirement (i) is satisfied. However, it might be the case that the trigger signals do not go low before $\overline{u*}$ is enabled or that they do not remain low once they have gone low until $u*$ is enabled again. Therefore, requirements (ii) and (iii) are not yet

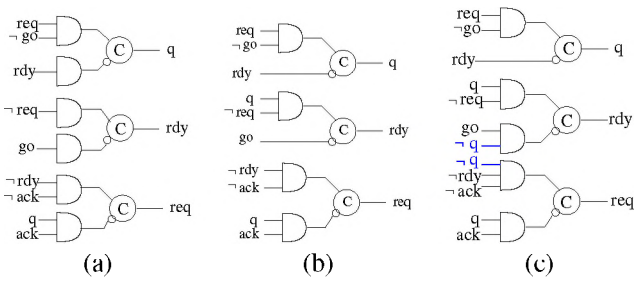


**Figure 4: (a) Minimal interval networks. (b) Timed implementation. (c) Speed-independent implementation.**

guaranteed to be satisfied. For example, the solid thick line in Figure 5 denotes the period in which the minimal interval network for the set interval of the signal *rdy* is turned on. However, the period should be equal to or shorter than the dotted line to satisfy the requirements. Thus, requirements (ii) and (iii) are not satisfied. The synthesis procedure guarantees them to be satisfied by adding some extra context signals to the AND gate. That is, it *shrinks* the period in which the cube yields 1.

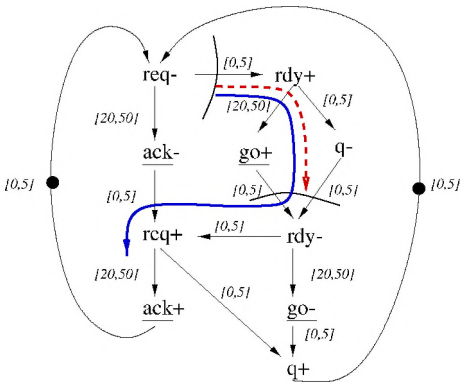


**Figure 5: A violation of the circuit model requirements.**

Figure 6 shows a sketch of the *shrink* procedure. In the algorithm, $s* \parallel t*$ denotes that $s*$ and $t*$ are timed concurrent and $s* \Rightarrow t*$ denotes that $s*$ causes $t*$ under the given timing constraints. To satisfy requirements (ii) and (iii), it is necessary to add signals which turn off the cube before $\overline{u*}$ is enabled and remain off until $u*$ is enabled again. To find such signals, the algorithm constructs a precedence graph. At first, the transitions which occur between the transition $u*$ and the transition $\overline{u*}$ are added as source nodes. Also, the transition $u*$ is added as a source node. The destination nodes for the precedence graph are found next. Here, the destination nodes are the reverse transitions of the non-redundant enabling transitions of $u*$. After finding source and destination nodes, the graph is expanded using the conditions in the algorithm. Figure 7 shows the precedence graph for the set interval of signal *rdy*. A node with a circle denotes a source node and a node with a rectangle denotes a destination node.

One meaning of the precedence graph is as follows: if there is an arc from $s*$ to $t*$, then the signal $s$ cannot return to the original state (change twice) without the reaction of the signal $t$. Extended to a set of signals, a path in the precedence graph guarantees that no nonempty subset can return to the original

```
shrink(STG G,transition u*)
{
  /* Construct a precedence graph */
  Precedence_graph ⟨V, E⟩ = ⟨∅, ∅⟩
  /* Find source and destination nodes */
  S_N = {u*}
  Foreach s* in G
    If (u* ⇒ s* and s* ⇒ ū* and s̄* ⇒ u*)
      S_N = S_N ∪ {s*}
  D_N = Find_destination_nodes(u*, G)
  V = S_N ∪ D_N

  /* Expand the precedence graph */
  Foreach unprocessed node s* in V
    Foreach t* in G
      If ((s* || t* or s* ⇒ t*) and t* ⇒ s̄* and t̄* ⇒ u*)
        V = V ∪ t*
        E = E ∪ (s*, t*)

  Foreach s_i ∈ S_N
    Foreach d_j ∈ D_N
      E_{i,j} = Find_all_possible_context_signals (s_i, d_j)

  Find_minimal_context_signal_set(E);
}
```

**Figure 6: A sketch of the *shrink* procedure.**

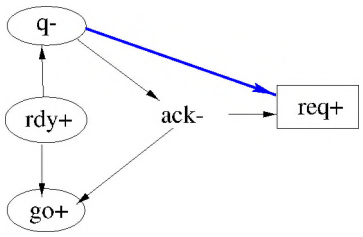


**Figure 7: Precedence graph for the interval $rdy+ \mapsto rdy-$.**

state while the remaining subset has no action. This property is used to guarantee the interval network turns off at the correct time and remains off for a proper period.

A set of transitions $T$ of a set of signals $X$ is called a Complementary Transition Set (CTS) [1] with respect to a state $S$ if it contains an equal number of falling and rising transitions for each signal of the set and each transition can be fired exactly once from the state without firing any other transition of $X$ not in $T$. A CTS has a corresponding vertex set in the precedence graph. For example, a CTS $T = \{q-, q+, req+, req-\}$ can be fired from the state when the $rdy+$ transition is enabled, and its corresponding vertex set in the precedence graph is $\{q-, req+\}$.

A CTS is said to be *complete* if no nonempty proper subset of it is also a CTS. Clearly, for a CTS $T = \{x_1+, x_1-, x_2+, x_2-, ..., x_n+, x_n-\}$, if a cube $C = c_1c_2...c_n$ (where $c_i$ is either $x_i$ or $\neg x_i$) is on in the state $S$, it is also on after each member of the CTS is fired exactly once. If a CTS $T = \{x_1+, x_1-, x_2+, x_2-, ..., x_n+, x_n-\}$ is complete and a cube $C = c_1c_2...c_n$ (where $c_i$ is either $x_i$ or $\neg x_i$) is on in the state $S$, then once the cube is set off it remains off until each member of the CTS is fired.

A CTS is complete if its corresponding vertex set in the precedence graph is connected by a directed path. This is because if a set of vertices is connected then any proper subset of it is connected to the remaining subset by at least one arc. This means that the transitions of the proper subset cannot be fired without firing other transitions in the CTS. This fact is used to find context signals which remain off for a proper time.

After constructing the precedence graph, the algorithm finds all possible sets of extra context signals for each destination node by finding all the paths from each source node to the destination node in the graph. By including a source node, requirement (ii) is satisfied. And by including all the signals in the path from a source node to a destination node, requirement (iii) is satisfied because the corresponding transitions become a complete CTS and $u*$ is enabled again when each member of the CTS fired once.

After finding all the possible sets of extra context signals for each destination node, the algorithm finds a minimal set of extra context signals for the interval by set multiplication operations. If there are many solutions with the same number of context signals, the algorithm selects the one which turns off the cube as late as possible. This optimizes the circuit area by allowing the elimination of memory elements. That is, if an interval network is turned on when $u*$ is enabled and turned off when $\overline{u*}$ is enabled, then the memory element can be removed.

In the precedence graph for the interval $rdy+ \mapsto rdy-$, shown in Figure 7, there is one destination node and the shortest path from a source node to the destination is $q- \rightarrow req+$, so the minimal context signal is $q$. The interval networks generated by the *shrink* procedure for the SCSI controller are shown in Figure 4 (b). Figure 4 (c) shows a speed-independent implementation. In the speed-independent implementation, the reset network of the signal $rdy$ has one more literal because the trigger signal $\neg q$ is not redundant in the speed-independent circuit. Moreover, the set network of the signal $req$ has one more literal because the paths in the precedence graph for a speed-independent circuit are longer than those in the precedence graph for a timed circuit as shown in the Figure 8.

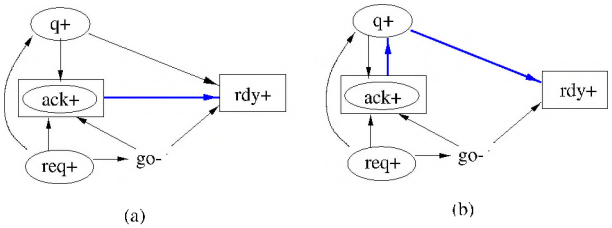


**Figure 8: Precedence graph for the interval $req+ \mapsto req-$, (a) for a timed circuit and (b) a speed-independent circuit.**

## 3.4 Removing Memory Elements

The algorithm improves the performance of the circuits by removing memory elements by finding a multi-cube interval network. It first checks to see if each interval network is turned on during the entire interval. If each set interval network of an output signal is turned on during its entire interval then the C-element and the reset network can be removed. By a similar analysis, the set network and C-element can be eliminated. If an interval network is not turned on during the entire interval, it is off before the end transition of the interval is enabled. So, the algorithm *expands* the period by combining the interval network and some other signals with an OR gate. Figure 9 shows a sketch of the *expand* procedure. It finds the extra inputs by

constructing a precedence graph and finding paths. It is similar to the *shrink* procedure. For the SCSI controller, no memory elements can be eliminated.

```
expand(STG G,transition u*, cube C)
{
  /* Construct a precedence graph */
  Precedence_graph ⟨V, E⟩ = ⟨∅, ∅⟩
  /* Find source and destination nodes */
  Foreach s* in G
    If (u* ⇒ s* and s̄* ⇒ ū* and (t* ⇒ s* and not (t̄* || s*)
        for all t*, where t* triggers the rising
        transition of the single cube C))
      S_N = S_N ∪ {s*}
    If (Is_a_non_redundant_enabling_transition(s*, ū*))
      D_N = D_N ∪ {s̄*}
  V = S_N ∪ D_N

  /* Expand the precedence graph */
  Foreach unprocessed node s* in V
    Foreach t* in G
      If ((s* || t* or s* ⇒ t*) and t* ⇒ s̄* and t̄* ⇒ ū*
          and u* ⇒ t*)
        V = V ∪ {t*}
        E = E ∪ {(s*, t*)}

  Foreach s_i ∈ S_N
    Foreach d_j ∈ D_N
      E_{i,j} = Find_all_possible_context_signals (s_i, d_j)

  Find_a_minimal_context_signal_set(E);
}
```

**Figure 9: A sketch of the *expand* procedure.**

## 3.5 Complexity and Performance

The algorithms for removing redundant arcs and finding the relations between any two signal transitions have a polynomial-time complexity. The algorithms for finding a single cube interval network and a multi-cube interval network are composed of two steps. The first step is to construct a precedence graph. This step has a polynomial-time complexity. The second step is to find all paths from each source node to each destination node in the graph. The complexity of this step depends on the number of cycles in the graph. In a directed graph which does not have any constraints, the number of cycles can be exponential with respect to the number of nodes. But in the precedence graph, a cycle is made if the transitions of two nodes are concurrent and each transition causes the next reverse transition of the other transition. So, the complexity of the algorithm for finding paths is exponential with respect to the number of concurrent transitions in the specification. However, the number of concurrent transitions seems to increase slowly with respect to the size of the STG specification.

Because the precedence graph represents all the possible candidates for the extra context signals in an interval, the algorithm finds a minimal single cube for each interval. However, the algorithm does not globally consider all the intervals of an output signal. If interval networks of the same output signal are not disjoint, they can be shared, resulting in less area. State graph based algorithms can handle this problem globally, but our algorithm produces only disjoint interval networks for output signals. On the other hand, our algorithm may find a multi-cube interval network to remove memory elements resulting in less area. Finally, since our algorithm uses a heuristic timing analysis, it may not determine the redundant arcs and other timing relations exactly. As a result, the synthesized circuits may not be optimal.

# 4 Experimental Results

Table 1 shows the experimental results. We compared timed circuit implementations found with our new direct method with those produced by ATACS state based method [6]. We compared area (using literal count) and CPU time. Note that the performance of the circuits is quite similar given that the two methods usually produced the same circuit. In the column $STG$, $S$ is the number of signals, $N$ is the number of nodes, and $A$ is the number of arcs. In the column $PG$, $G$ is the number of precedence graphs, $N$ is the average number of nodes per precedence graph, and $A$ is the average number of arcs per precedence graph. To generate examples with large state spaces, we connected the SCSI controller specification in parallel. Also, we synthesized a multi-stage, series connected FIFO [12]. The experimental results show that our synthesis method does not have the state explosion problem and achieves significant reductions in synthesis time as compared to previous methods in examples with large state spaces. For the specifications with small state spaces, the direct synthesis method may be slower than the previous method. In addition, because the direct method searches the precedence graph exhaustively to find a minimal single cube network, it may be slow for specifications whose precedence graphs are very large. However, the size of the precedence graph does not seem to grow as fast as the state space. For multi-stage FIFO circuits, the size of the precedence graphs remain almost constant because they are connected serially. For SCSI controllers, the size of the precedence graphs increase linearly with the size of STG.

**Table 1: Experimental results.**

| Example | STG (S/N/A) | States | PG (G/N/A) | ATACS | | Direct Method | |
|---|---|---|---|---|---|---|---|
| | | | | Total Literals | CPU time (sec) | Total Literals | CPU time (sec) |
| half | 4/8/11 | 14 | 4/4/6 | 8 | 0.03 | 8 | 0.03 |
| full | 4/8/12 | 16 | 4/4/6 | 8 | 0.03 | 8 | 0.01 |
| converta | 5/14/16 | 19 | 8/5/7 | 20 | 0.04 | 20 | 0.04 |
| sender-done | 4/8/9 | 9 | 5/3/3 | 5 | 0.03 | 5 | 0.03 |
| mp-fwd-pkt | 8/16/26 | 22 | 14/5/7 | 16 | 0.05 | 14 | 0.05 |
| MMU | 8/16/23 | 92 | 8/7/17 | 22 | 0.06 | 22 | 0.05 |
| master-read | 18/28/40 | 2108 | 16/7/17 | 34 | 2.01 | 34 | 0.15 |
| AtoD | 7/14/19 | 24 | 12/5/8 | 12 | 0.04 | 12 | 0.04 |
| counter3 | 6/22/58 | 32 | 14/8/10 | 23 | 0.14 | 40 | 0.14 |
| elatchB | 8/16/35 | 55 | 20/8/27 | 14 | 0.12 | 14 | 0.07 |
| VME | 5/10/26 | 19 | 6/5/8 | 6 | 0.05 | 6 | 0.05 |
| cstat | 3/6/11 | 8 | 2/3/4 | 4 | 0.04 | 4 | 0.03 |
| inv | 2/4/5 | 4 | 2/2/1 | 1 | 0.01 | 1 | 0.01 |
| lapbN | 8/16/28 | 97 | 13/5/8 | 20 | 0.14 | 1 | 0.01 |
| pab_c4 | 4/8/24 | 16 | 4/4/7 | 8 | 0.05 | 8 | 0.05 |
| SCSI Ctrl | 5/10/17 | 16 | 7/4/5 | 10 | 0.02 | 10 | 0.02 |
| 4 SCSI | 14/28/62 | 806 | 28/9/42 | 40 | 1.17 | 40 | 0.22 |
| 8 SCSI | 26/52/122 | 404006 | 56/15/166 | 80 | 4937.36 | 80 | 1.29 |
| 9 SCSI | 29/58/137 | N/A | 63/17/210 | N/A | N/A | 90 | 1.96 |
| 10 SCSI | 32/64/152 | N/A | 70/19/260 | N/A | N/A | 100 | 2.91 |
| 20 SCSI | 62/124/302 | N/A | 140/34/1058 | N/A | N/A | 200 | 24.68 |
| 40 SCSI | 122/244/602 | N/A | 280/65/4284 | N/A | N/A | 400 | 246.51 |
| 60 SCSI | 182/364/902 | N/A | 420/97/9681 | N/A | N/A | 600 | 1019.0 |
| 80 SCSI | 242/484/1202 | N/A | 560/129/17250 | N/A | N/A | 800 | 3505.15 |
| 100 SCSI | 302/604/1502 | N/A | 700/160/26990 | N/A | N/A | 1000 | 8231.93 |
| 120 SCSI | 362/724/1802 | N/A | 840/191/38901 | N/A | N/A | 1200 | 16395.79 |
| 150 SCSI | 452/904/2252 | N/A | 1050/239/60840 | N/A | N/A | 1500 | 38976.31 |
| 180 SCSI | 542/1084/2702 | N/A | 1260/286/87664 | N/A | N/A | 1800 | 82151.49 |
| FIFO 1-stage | 7/14/31 | 29 | 6/12 | 9 | 0.06 | 9 | 0.02 |
| FIFO 4-stgs | 22/44/97 | 10176 | 48/10/40 | 36 | 39.57 | 36 | 0.69 |
| FIFO 5-stgs | 27/54/119 | 67392 | 60/10/40 | 45 | 456.6 | 45 | 1.23 |
| FIFO 6-stgs | 32/64/141 | N/A | 72/10/40 | N/A | N/A | 54 | 2.2 |
| FIFO 7-stgs | 37/74/163 | N/A | 84/10/41 | N/A | N/A | 63 | 3.53 |
| FIFO 10-stgs | 52/104/229 | N/A | 120/10/41 | N/A | N/A | 90 | 16.98 |
| FIFO 20-stgs | 102/204/449 | N/A | 240/10/42 | N/A | N/A | 180 | 139.17 |
| FIFO 40-stgs | 202/404/889 | N/A | 480/10/43 | N/A | N/A | 360 | 1240.98 |
| FIFO 60-stgs | 302/604/1329 | N/A | 720/10/43 | N/A | N/A | 540 | 4558.70 |
| FIFO 80-stgs | 402/804/1769 | N/A | 960/10/43 | N/A | N/A | 720 | 11351.76 |
| FIFO 100-stgs | 502/1004/2209 | N/A | 1200/10/43 | N/A | N/A | 900 | 19079.43 |

We ran the two programs on a 400MHz PentiumII with 384MB main memory and 700MB swap memory. For exam-

ples with state spaces exceeding one million states, the previous method did not finish due to the lack of memory. The area of the synthesized circuits are the same in most cases. In some specifications, such as *mp-forward-pkt*, the direct method produces smaller circuits because it removes memory elements by finding multi-cube interval networks. In some examples, such as $counter3$, the direct method produces a bigger circuit because it does not consider sharing among the interval networks of the same output signal.

If all the timing constraints in the timed STG specification are given as $[0, \infty]$, the synthesized circuit is speed-independent. The top 7 examples in Table 1 are speed-independent and the remaining ones are timed. We also compared our results to the synthesis tool for speed-independent circuits, named Petrify [13]. The CPU time with Petrify was 255.73 seconds for 8 untimed SCSI controllers and 1616.81 seconds for 10 untimed SCSI controllers. It did not finish for 13 controllers after running for one day. It is notable that our synthesis method can synthesize 60 SCSI controllers within 20 minutes. Whereas, the method in [6] can only synthesize 8 SCSI controllers. Also, it is notable that our synthesis method is about 100 times faster than the method in [13] and about 1000 times that of the method in [6] for specifications with large state spaces. In comparing the synthesis results among the various methods, it is important to note that the synthesized circuits are very similar.

We also compared our results to the direct synthesis method for speed-independent circuits in [7]. Both programs were run on the same SUN Sparc20 with 128MB of main memory. The CPU time for the tool from [7] is 19.23 seconds for 10 untimed SCSI controllers and 3868.85 seconds for 60 untimed SCSI controllers. The CPU time of the method described in this paper was 11.78 seconds for 10 untimed SCSI controllers and 6419.85 seconds for 60 untimed SCSI controllers. Even though the suggested method uses an exhaustive approach and the method in [7] uses a heuristic approach, the CPU times are quite similar. Whereas the method in [7] cannot synthesize 70 untimed SCSI controllers because it runs out of memory, the method in this paper can synthesize 90 untimed SCSI controllers on SUN Sparc20.

# 5 Conclusions

This paper presents a direct synthesis method for timed circuits. It shows that a timed circuit — not containing circuit hazards under given timing constraints — can be found by using the timing relations between signal transitions of the specification. Moreover, these relationships can be efficiently found using a heuristic timing analysis algorithm. The results indicate that by using the direct synthesis approach, we can overcome the state explosion problem. Currently, the synthesis algorithm can handle only deterministic specifications. Future work includes the extension of the algorithm to specifications with free-choice behavior. Also, we plan to extend the target circuit model and synthesis algorithm to apply gate sharing between interval networks.

# Acknowledgements

We would like to thank Eric Mercer and Kip Killpack of the University of Utah for their comments on this paper.

# References

[1] Tam-Anh Chu, "Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications", *PhD thesis, MIT Laboratory for Computer Science,* Jun. 1987.

[2] T.H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt, "Automatic synthesis of asynchronous circuits from high-level specifications", *IEEE Transactions on Computer-Aided Design,* Vol. 8, No. 11, pp. 1185-1205, Nov. 1989.

[3] P. Beerel and T.H.-Y. Meng, "Automatic Gate-Level Synthesis of Speed-independent Circuits", *In Proceedings of International Conference on Computer Aided Design,* pp. 581-586 Nov. 1992.

[4] C. Ykman-Couvreur, B. Lin, and H. de Man, " Assassin: A synthesis system for asynchronous control circuits", *Technical report - User and Tutorial manual, IMEC,* Sep. 1994.

[5] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli, "Algorithms for Synthesis of Hazard-Free Asynchronous Circuits", *Proceedings of the 28th Design Automation Conference,* , 1991.

[6] C.J. Myers, T. H.-Y. Meng, "Synthesis of Timed Asynchronous Circuits", *IEEE Transitions on VLSI Systems,* pp. 106-119 June 1993.

[7] E. Pastor, J. Cortadella, A. Kondratyev, and O. Roig, "Structural Methods for the Synthesis of Speed-Independent Circuits", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* Vol. 17, No. 11, pp. 1108-1129, Nov. 1998.

[8] A. Semenov, A. Yakovlev, E. Pastor, M.A. Peña and J. Cortadella, "Synthesis of Speed-independent circuits from STG-unfolding segment", *Proc. 34th ACM/IEEE Design Automation Conference,* pp. 16-21, June, 1997.

[9] K.J. Lin, C.W. Kuo and C.S. Lin, "Synthesis of Hazard-Free Asynchronous Circuits Based on Characteristic Graph", *IEEE Transactions on Computers,* Vol. 46, No. 11, pp. 1246-1263, Nov. 1997

[10] S.T. Jung and C.S. Jhon, "Direct Synthesis of Efficient Speed-independent Circuits from Deterministic Signal Transition Graphs", *Proceedings of International Symposium on Circuits and Systems,* pp. 307-310, June, 1994

[11] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanvekbergen, and Yakovlev, "Basic Gate Implementation of Speed-independent Circuits", *In Proceedings of Design Automation Conference,* pp. 56-62 June, 1994.

[12] Charles E. Molnar, Ian W. Jones, Bill Coates, and Jon Lexau. "A FIFO ring oscillator performance experiment", In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems.* IEEE Computer Society Press, April 1997.

[13] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers", *IEICE Transactions on Information ans Systems,* Vol. E80-D, No. 3, Mar. 1997, pp. 315-325.