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. Our synthesis procedure begins with a deterministic signal transition graph specification to which timing constraints can be added. First, a timing analysis extracts the *timed concurrency relation* and *timed causality relation* between any two signal transitions. Then, a hazard-free implementation under the timing constraints is synthesized by constructing a precedence graph and finding a shortest path in the graph. Our method does not have the state explosion problem while the synthesized circuits have nearly the same area with the previous timed circuits.

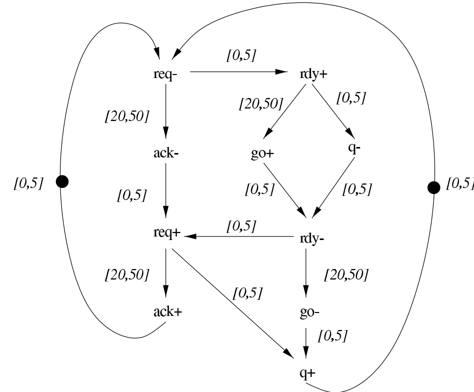

Figure 1: The timed STG for a SCSI controller.

1 Introduction

Speed-independent circuits are very robust since they are guaranteed to work independent of the delays associated with their gates, but they can be overly conservative when timing constraints are available. Methods have been proposed to use timing constraints to synthesize timed circuits which work correctly under the given timing constraints [1, 2]. Timed circuits tend to be more efficient in area and speed than speed-independent circuits [2].

These synthesis techniques in [1, 2] have the state explosion problem because they are based on a state graph. To avoid the problem, this paper presents a direct synthesis method which does not use a state graph. Such direct methods have been proposed for speed-independent circuits [3]. The method in [3] synthesizes speed-independent circuits by using a structural analysis directly on signal transition graphs. In contrast, our method synthesizes timed circuits by using the relations between signal transitions.

In order to synthesize timed circuits directly, timing analysis must be used on the specification to deduce timing information necessary to detect the timed concurrency relation and timed causality relation between any two signal transitions in a circuit specification. For the timing analysis, we use the polynomial-time heuristic algorithm in [2]. After timing analysis, the algorithm synthesizes efficient timed circuits by constructing a precedence graph and finding a shortest path in the graph. In examples with large state spaces, we demonstrate significant reductions in synthesis time as compared to previous methods.

*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.

2 Timed Specifications

Figure 1 shows a timed deterministic signal transition graph (STG) specification for a SCSI protocol controller specification [2]. In the STG specification, a node denotes a rising or falling signal transition, and an arc denotes an ordering relation between two transitions. An arc can have a solid circle which is used to denote 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 signal transition is enabled when each input arc has a token and the timing constraints are satisfied. If there is an arc from $s+$ to $t+$, $s+$ is called an *enabling* transition of $t+$ and s is called an enabling signal of $t+$. If a signal transition is enabled, it can be fired. When a signal transition is fired, all the tokens on the input arc are removed and a token is added to each output arc.

In order to synthesize timed circuits, timing analysis must be used on the specification to deduce timing information. The timing information needed is the minimum and maximum difference in time between any two signal transitions in a circuit specification. The timing analysis algorithm in [2] starts with a cyclic graph specification and unfolds the specification into an infinite acyclic graph. Then it examines only two finite acyclic subgraphs of the infinite graph to determine a sufficient bound on the time difference between two signal transitions.

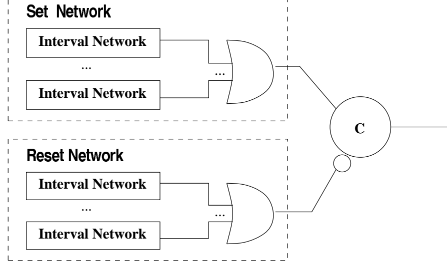

Figure 2: Target circuit model for an output signal.

3 Synthesis procedure

Figure 2 illustrates the target circuit model of our synthesis algorithm for each output signal. The circuit can be implemented with discrete gates as shown, or it can be built using a single generalized C-element. A set and a reset network is synthesized as a sum of interval networks as shown in the figure. An interval network implements a transition of the output signal and only one interval network yields 1 at a time. Let an interval denote the period between the time when a transition is enabled and the time when the next reverse transition of it is enabled. Here, the former transition is referred to as the start transition of the interval and the latter one is referred to as the end transition of the interval.

Our synthesis algorithm consists of the following four steps. The first step is to detect and remove redundant arcs from the specification. The second step is to get the relations between any two signal transitions. The third step is to construct a precedence graph and find a shortest path in the graph to derive a single cube circuit implementation for each set and reset interval network for each transition of an output signal. The fourth step is to find a multi-cube combinational implementation, if one exists.

3.1 Removing Redundant Arcs

If there are multiple enabling transitions for a signal transition, then it is possible that some of them are redundant. Each enabling transition results in a literal in the implementation of the signal. If an enabling transition is redundant, the corresponding literal can be removed from the implementation resulting in a smaller circuit. In the SCSI protocol controller example depicted 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]. Since the lower bound of this time difference, 15, is greater than the upper bound of the timing constraint on the arc, 5, the arc is found to be redundant.

3.2 Finding the Relations

In order 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, we first find untimed concurrent transitions by reachability analysis on the STG (not the state space). Then, we check 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 $ack-$ are $go+$ 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. But they are not timed concurrent because 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. A transition $e+$ can $cause$ a transition $f+$ if they are ordered and there exists a path from $e+$ to $f+$ and each existing path does not contain the transition $e-$. 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, as mentioned above, $q-$ always fires before $go+$.

3.3 Finding a Single Cube Network

The synthesis procedure synthesizes each interval network as a single cube. In [4], conditions are developed in which each interval can be implemented as a single cube in a hazard-free manner. In [5], they showed that specifications can be transformed to satisfy these conditions by inserting new signals. Our algorithm currently only handles specifications which have a single cube implementation. 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.

In order to illustrate our synthesis process of an interval network, we must first introduce the notion of the *1-interval* and the *0-interval* of an interval network. The period in which an interval network should have the value 1 in order to be hazard-free under the timing constraints is referred to as the 1-interval of the interval network. The 0-interval is similarly defined. The synthesis process starts with a minimal interval network which is an AND gate having only the non-redundant enabling signals as inputs. All the enabling signals go high at the start of the 1-interval, so the minimal cube has the value 1. However, the enabling signals may not go low until after the 1-interval. That is, the interval network may yield 1 within the 0-interval. Our synthesis procedure removes it by adding some extra signals to the AND gate. That is, it shrinks the period in which the interval network yields 1.

Figure 3 shows a sketch of the shrink procedure. In the algorithm, $u*$ denotes the start transition of the current interval, $s* \parallel t*$ denotes that $s*$ and $t*$ are timed concurrent and $s* \Rightarrow t*$ denotes that $s*$ causes $t*$ under the given tim-

ing constraints. To find the needed extra 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 signals of $u*$. After finding source and destination nodes, the graph is expanded so that if a signal transition can continue to yield 0 following the signal transition of the existing node, the transition is added as a new node and an arc is added between them.

shrink(STG G,transition $u*$)
{
  /* Construct a precedence graph */
  Precedence_graph $\langle V, E\rangle = \langle \emptyset, \emptyset \rangle$
  /* Find source and destination nodes */
  $S_N = \{u*\}$
  Foreach $s*$ in G
    If ($u* \Rightarrow s*$ **and** $s* \Rightarrow \overline{u*}$ **and** $\overline{s*} \Rightarrow u*$)
      $S_N = S_N \cup \{s*\}$
    If (Is_a_non_redundant_enabling_transition($s*$, $u*$))
      $D_N = D_N \cup \{\overline{s*}\}$
  $V = S_N \cup D_N$

  /* Expand the precedence graph */
  Foreach unprocessed node $s*$ in V
    Foreach $t*$ in G
      If (($s* \,||\, t*$ **or** $s* \Rightarrow t*$) **and** $t* \Rightarrow \overline{s*}$ **and** $\overline{t*} \Rightarrow u*$)
        $V = V \cup \{t*\}$
        $E = E \cup \{(s*, t*)\}$

  Foreach $s_i \in S_N$
    Foreach $d_j \in D_N$
      $E_{i,j}$ = Find_all_possible_context_signals (s_i, d_j)

  Find_a_minimal_context_signal_set(E);
}

Figure 3: A sketch of the *shrink* function.

After constructing the precedence graph, the algorithm finds all possible sets of extra signals for each destination node by finding all the paths from each source node to the destination node in the graph. Let $t-$ be a non-redundant enabling transition of $u*$. Then the signal t' is included in the initial cube and $t+$ becomes one of the destination nodes. If there is a path $s_1+ \rightarrow s_2+ \rightarrow ... \rightarrow s_n+ \rightarrow t+$, where s_1 is a source node and $t+$ is a destination node, then the extra signals are $s_1' s_2' ... s_n'$ because the signal s_1' yields 0 before $\overline{u*}$ is enabled and s_{i+1} continues to yield 0 following s_i, where $1 \leq i \leq n-1$. Note that the enabling signal t' yields 0 following s_n'. When the signal t' yields 1, the transition $u*$ is enabled.

After finding all the possible sets of extra context signals for each destination node, the algorithm finds a minimal set of extra signals for the interval by set multiplication operations. If there are many solutions with the same number of extra signals, the algorithm selects the one which makes the cube yield 0 as late as possible. This increases the possibility of finding a combinational network.

For the interval, $req+ \mapsto req-$, the minimal interval network is $f = \neg ack \wedge \neg rdy$. Figure 4(a) shows the precedence graph of the interval. The circled nodes, $q+$, $ack+$, and $req+$ are source nodes and the rectangled nodes, $ack+$ and $rdy+$ are the destination nodes. Because the destination node $ack+$ is a start node itself, no extra signal is necessary for it. And the shortest path for the destination node $rdy+$ is $ack+ \rightarrow rdy+$. So, the necessary extra signal is $\neg ack$ which is already in the minimal interval network. As a result, no extra signal is necessary for the interval. Figure 4(b) shows the precedence graph of the interval in the untimed case for the SCSI controller specification. In this graph, the shortest path for the destination node $rdy+$ is $ack+ \rightarrow q+ \rightarrow rdy+$. So, the extra signal $\neg q$ is necessary resulting in a bigger circuit compared to the timed circuit. The synthesized interval networks by the shrink procedure for the timed SCSI controller specification and the untimed SCSI controller specification are shown in Figure 5(a) and Figure 5(b) respectively.

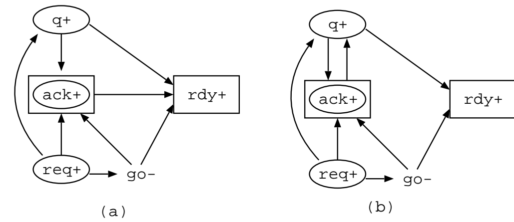

Figure 4: Precedence graph for the interval $req+ \mapsto req-$. (a) For a timed circuit. (b) For a speed-independent circuit.

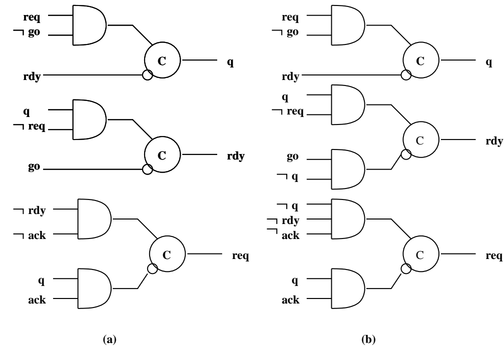

Figure 5: Circuit implementation. (a) Timed circuit (b) Speed-independent circuit

3.4 Finding a Combinational Network

Our algorithm improves the performance of the circuits by finding a combinational implementation. It checks if each interval network can be changed to a combinational one by combining the interval network and some other signals with an OR gate. If an interval network is not combinational, its output becomes 0 before the end transition of the interval is enabled. So, the extra inputs of the OR gate are used to make the interval network yield 1 until the end transition is enabled. We find the extra inputs by constructing a precedence graph and finding shortest paths. It is very similar to the *shrink* procedure. For the SCSI controller specification, there is no combinational network.

4 Experimental results

Table 1 shows the experimental results. We compared our timed implementation with the previous timed implementation [2] by the number of literals and CPU time. We synthesized both generalized-C implementations and standard-C implementations. The CPU time in the table are for generalized-C implementations. The CPU times for standard-C implementations are almost the same. To make examples with a huge number of state, we connected a number of SCSI controller specifications in parallel. Also, we synthesized a multi-stage, series connected FIFO [6]. 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 a small state space, the direct synthesis method may be slower than the previous method. Also, because the direct method searches the precedence graph exhaustively to find a minimal single cube network, it may be slow for the 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 grows.

Table 1: Experimental results.

Example	ATACS				Direct Method		
	States	Total Literals		CPU time (sec)	Total Literals		CPU time (sec)
		gC	sC		gC	sC	
qr42	18	12	14	0.04	12	14	0.04
mp-forward-pkt	22	16	16	0.05	14	14	0.05
master-read	2108	34	34	2.01	34	34	0.15
counter3	32	21	39	2.01	37	37	0.15
AtoD	24	12	12	0.04	12	12	0.05
VME	19	6	6	0.05	6	6	0.05
SCSI Controller	16	10	10	0.02	10	10	0.02
4 SCSI Ctrls	806	40	40	1.17	40	40	0.22
5 SCSI Ctrls	3646	50	50	8.02	50	50	0.32
6 SCSI Ctrls	17150	60	60	58.28	60	60	0.51
7 SCSI Ctrls	82630	70	70	441.21	70	70	0.78
8 SCSI Ctrls	404006	80	80	4937.36	80	80	1.29
9 SCSI Ctrls	N/A	N/A	N/A	N/A	90	90	1.96
10 SCSI Ctrls	N/A	N/A	N/A	N/A	100	100	2.91
20 SCSI Ctrls	N/A	N/A	N/A	N/A	200	200	24.68
FIFO 1-stage	29	9	9	0.06	9	9	0.02
FIFO 3-stages	1533	27	27	3.25	27	27	0.33
FIFO 4-stages	10176	36	36	39.57	36	36	0.69
FIFO 5-stages	67392	45	45	456.6	45	45	1.23
FIFO 6-stages	N/A	N/A	N/A	N/A	54	54	2.2
FIFO 7-stages	N/A	N/A	N/A	N/A	63	63	3.53

We ran the two programs on a 400MHz PentiumII with 384MB main memory and 700MB swap memory. For examples with state spaces exceeding one million states, the previous method didn't finish because of a lack of memory. The area of the synthesized circuits are the same in most cases. In some specifications, the direct method produces smaller circuits because it finds multi-cube combinational networks while the previous method doesn't find them. For the example $counter3$, the direct method produces a bigger circuit for the generalized-C implementation 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 3 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[7]. The CPU time was 255.73 second for 8 untimed SCSI controllers and 1616.81 second for 10 untimed SCSI controllers. It did not finish for 13 controllers after running for one day.

5 Conclusions

We have developed a direct synthesis method for timed circuits. We show that a timed circuit without circuit hazards under given timing constraints can be found by using the relations between signal transitions of the specification and the relations can be efficiently found using a heuristic timing analysis algorithm. Our results indicate that by using the direct synthesis approach, we can overcome the state explosion problem. Currently, our synthesis algorithm can handle only deterministic specifications. We plan to extend our algorithm to specifications with free choice behavior.

References

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

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

[3] 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.

[4] 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.

[5] 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 Jun. 1994.

[6] 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.

[7] 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.