

# Direct Synthesis of Timed Circuits from Free-Choice STGs

Sung-Tae Jung and Chris J. Myers

*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 graph specification with timing constraints. A timing analysis extracts the *timed concurrency* and *timed causality* relations directly from the specification. 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, in practice achieves significant reductions in synthesis time for the specifications which have a large state space, 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.

*Keywords*— Timed asynchronous circuits, timed concurrency, timed causality, free-choice STG, state explosion, timing analysis.

## I. 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 require construction of the reachable state space. To overcome the state explosion problem, direct methods have been proposed for speed-independent circuits [7], [8], [9], [10], [11], [12]. The methods in [7], [8] use lock graphs which are a graphical model to represent lock relations among signals. Even though synthesis based on lock graphs is efficient, the specifications that can be handled are restricted because the lock

relation is only a sufficient condition for the existence of a speed-independent circuit implementation. The method in [9] 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 the correctness criteria, then iterative refinement is performed using state machine decompositions. This method is restricted to state machine decomposable specifications. The method in [10] uses an approach similar to [9] but it allows for a wider class of specifications by finding an initial approximation and refining it using STG-unfolding segments. The method in [11] 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 [12] constructs a precedence graph for each transition of each output signal 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 direct 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 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 [12].

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. The benchmark set also includes timed STG examples which have been converted from burst-mode AFSM (Asynchronous Finite State Machine) specifications and to which timing information has been added. These examples are derived from several academic and industrial designs [13], [17], [18], [19], [20]. For some of these examples, even though the timed state space explodes, the direct method could still produce a timed

This research is supported by a grant from Intel Corporation, an NSF CAREER award MIP-9625014, and partially by Wonkwang University in 2000.

C. Myers is with the Dept. of Electrical Engineering, University of Utah, Salt Lake City, UT 84112. E-mail: myers@ee.utah.edu.

S.-T. Jung is with the Dept. of Computer Engineering, Wonkwang University, Iksan City, Jeonbuk 570-749, Korea. E-mail: stjung@wonkwang.ac.kr

circuit implementation. We compared timed circuit implementations found with our new direct method with those produced by ATACS state based method [6]. In all the benchmarks, the area measured as number of literals of the implementations between the two methods is identical or nearly so. For 43 of the 57 examples which completed using both methods, the literal counts are the same. For 5 of the 57 examples, state space based methods produced a smaller literal count. For 9 of the 57 examples, the direct methods produced a smaller literal count. This last result typically came about due to the optimization for combinational circuits described in this paper. 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.

This paper contains six sections. Section 2 describes our specification language. Section 3 describes the synthesis procedure for the specifications which have only a single occurrence of each signal transition per cycle, have the unique state coding property, and have no choice behavior. Section 4 presents the extensions of the synthesis procedure to specifications which have multiple occurrences of each signal transition, have the complete state coding property, and have free-choice behavior. Section 5 shows experimental results. Section 6 gives our conclusions.

## II. TIMED SPECIFICATIONS

This section describes the basic notations and properties of timed free-choice STGs which are used as a specification method in this paper.

### A. Petri nets and Signal Transition Graphs

A *Petri net* is defined as the four tuple  $\langle P, T, F, M_0 \rangle$ , where  $P$  is the set of *places*,  $T$  is the set of *transitions*,  $F \subseteq (P \times T) \cup (T \times P)$  is the *flow relation* and  $M_0$  is the initial marking. A *marking* is an assignment of *tokens* to a subset of the places in the net. If  $(p, t) \in F$ ,  $p$  is an *input place* of  $t$  and  $t$  is an *output transition* of  $p$ . Conversely, if  $(t, p) \in F$ ,  $t$  is an *input transition* of  $p$  and  $p$  is an *output place* of  $t$ . There are three important subclasses of Petri nets called *marked graphs* (MG), *state machines* (SM), *free-choice* Petri nets. A marked graph is a Petri net in which each place has exactly one input transition and one output transition. A state machine is a Petri net in which each transition has exactly one input place and one output place. A free-choice Petri net is a Petri net in which if any two transitions  $t_1$  and  $t_2$  share the same input place  $p$ , then  $p$  is the unique input place of both  $t_1$  and  $t_2$ . Similarly, if two transitions share an output place  $p$ , then  $p$  is the unique output place for both.

A free-choice *signal transition graph* (STG) is a free-choice Petri net such that each transition is interpreted as a rising transition or a falling transition of a signal. A rising transition and a falling transition of a signal  $s$  are denoted by  $s+$  and  $s-$ , respectively. A transition  $s*$  denotes  $s+$  or  $s-$ . A transition is *enabled* when all input places have at least one token. When an enabled transition *fires*, it removes one token from each input place and adds one

token to each output place. If  $(s*, p) \in F$  and  $(p, t*) \in F$  then  $s*$  is called the *enabling* transition and  $t*$  is called the *enabled* transition for the place.

A timed free-choice STG is a free-choice STG such that each place is associated with a timing constraint  $[l, u]$ , where  $l$  denotes the lower bound and  $u$  denotes the upper bound. Let  $TC_l(t*, s*)$  and  $TC_u(t*, s*)$  denote the lower bound and upper bound of the timing constraint between  $s*$  and  $t*$ , respectively. Let  $TC(t*, s*) = [TC_l(t*, s*), TC_u(t*, s*)]$ . A timing constraint is said to be *satisfied* if a token has been on a place longer than the lower bound for that place. It is said to be *expired* if the amount of time that the token has been on the place exceeds the upper bound. In a timed free-choice STG, a signal transition is enabled when all the timing constraints of the input places are satisfied. A transition should be fired before all timing constraints on the input places have 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. A free-choice place is a place which has multiple output transitions. When a free-choice place has a token and the associated timing constraint is satisfied, all the output transitions are enabled. But only one transition is freely chosen to fire and the other transitions are disabled.

Using timing constraints in synthesis creates a chicken-and-egg problem. Timing information must be provided early in the design phase before concrete timing numbers can be known. However, when an enabled transition is a transition of an input signal, the timing constraints may 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 using the algorithm described in this paper, it should be analyzed using a timing verification tool to verify that the timing constraints used are correct. One such tool is *orbits* [14] which we have used to verify our designs. If the circuit violates the timing constraints, it must be resynthesized with more conservative timing constraints.

Figure 1 shows a timed free-choice STG specification. In Figure 1, signal transitions are denoted by their names. Transitions on input signals are underlined. Multiple transitions for a signal are denoted by indexes such as  $b + /1$  and  $b + /2$ . Circles denote places. For simplicity, the places with only one input transition and one output transition are omitted in the figure. Arcs denote the flow relation. A solid circle within a place denotes a token. In the initial marking, either  $a + /1$  or  $b + /2$  may fire 20 to 50 time units after reset. If  $a + /1$  is selected to fire, then the token in its input place is removed, and a token is added in its output place enabling  $d - /1$ . After 20 time units, the timing constraint on this place becomes satisfied, and before 50 time units have elapsed,  $d - /1$  must fire. The firing of  $d - /1$  enables  $c+$  and  $a - /1$ . While ignoring timing these two transitions can fire in either order, when timing

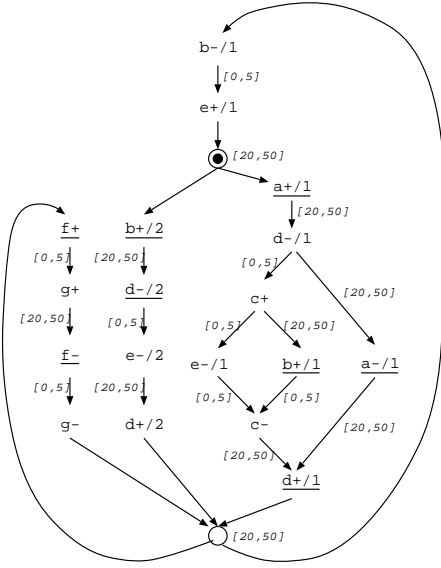


Fig. 1. A timed free-choice STG example.

is considered, the next transition to fire will be  $c+$  since it must do so within 5 time units and  $a- /1$  cannot fire for at least 20 time units. Continuing in this way, all possible reachable states of this timed STG can be found.

### B. Properties

This section describes some important properties of timed free-choice STGs. A timed free-choice STG satisfies the *unique state coding* property if every state in its corresponding state graph has a different binary code. A timed free-choice STG satisfies the *complete state coding* property if when two or more states have an identical binary code, all the output signal transitions enabled in these states are the same. The last property is *persistence*. When  $s^*$  is an enabling transition of an output transition  $t^*$ , if  $t^*$  cannot fire concurrently with the next reverse transition of  $s$ ,  $\bar{s}^*$ , then  $s^*$  is persistent with respect to  $t^*$ . A timed free-choice STG is *persistent* if all enabling transitions are persistent with respect to each output transition they enable. In this paper, the synthesis algorithm requires timed free-choice STGs to satisfy the complete state coding property and persistence. To simplify the explanation, the synthesis algorithm in Section 3 considers only STGs that also satisfy the unique state coding property. Section 4 extends our approach to STGs satisfying the complete state coding property.

### C. Timing Analysis

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 signal transitions in the circuit specification. There are timing analysis methods which can be applied to a timed free-choice STG [15], [16]. However, the method in [15] cannot be used in direct synthesis methods because it is based on the state graph. The method in [16] does timing analysis on timed

free-choice Petri-nets directly, but it can be computationally expensive for large specifications. In this paper, we instead decompose a timed free-choice STG into a set of marked graph (MG) components and apply timing analysis to each MG-component. For the timing analysis, the synthesis procedure uses the polynomial-time heuristic algorithm in [6]. The timing analysis algorithm starts with a cyclic marked 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 the two signal transitions.

Let  $TD(t^*, s^*) = [L, U]$  denote the minimum and maximum time separation between two corresponding instances of the transition  $s^*$  and  $t^*$ , where the base transition is  $s^*$ .  $TD(t^*, s^*) = [10, 30]$  means that the transition  $t^*$  occurs after at least 10 time units after  $s^*$  but before 30 time units have elapsed since  $s^*$ .  $TD(t^*, s^*) = [-50, -20]$  means that the transition  $t^*$  occurs before  $s^*$  by at least 20 time units but at most 50 time units before  $s^*$ . Let  $TD_L(t^*, s^*)$  and  $TD_U(t^*, s^*)$  denote the minimum and maximum value of  $TD(t^*, s^*)$  respectively. Note that by definition  $TD_L(t^*, s^*) = -TD_U(s^*, t^*)$ . Also note that for untimed systems (i.e., all delays are 0 to  $\infty$ ), if  $t^*$  and  $s^*$  are concurrent then  $TD_L(t^*, s^*) = -\infty$  and  $TD_U(t^*, s^*) = \infty$ .

## III. SYNTHESIS PROCEDURE

This section describes the synthesis procedure for STGs which have only a single occurrence of each signal transition per cycle, have the unique state coding property, and have no choice behavior. Extensions to the synthesis procedure that lift these restrictions are given in the next section.

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. The memory element is a Muller C-element which takes two inputs and sets its output when both inputs are high and resets its output when both inputs are low. Note the inverting bubble on one input. This means that this circuit sets its output high when the set network evaluates to 1 and the reset network evaluates to 0.

Let an interval,  $u^* \mapsto \bar{u}^*$ , denote the period between the time when  $u^*$  is enabled and the time when  $\bar{u}^*$ , the next reverse transition of  $u^*$ , is enabled. The interval network for the interval  $u^* \mapsto \bar{u}^*$  must satisfy the following hazard-freedom requirements:

1. It is turned on when  $u^*$  is enabled,
2. It is turned off after  $u^*$  is fired but before  $\bar{u}^*$  is enabled, and
3. Once it is turned off, it remains off until  $u^*$  is enabled again.

These requirements are the same as those in [3], [22].

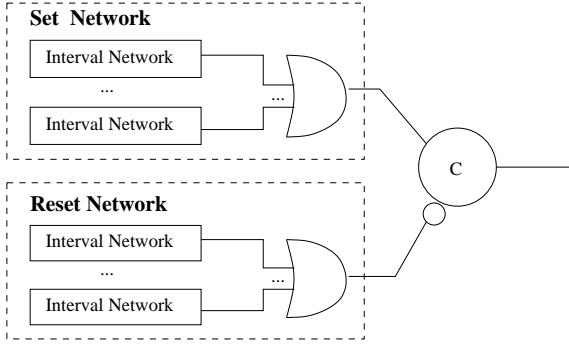


Fig. 2. Target circuit model for an output signal.

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.

#### A. 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.

**Definition 1:** (Redundant Trigger) A trigger is redundant if the following condition is satisfied for the corresponding arc  $s* \rightarrow t*$ :  $TD_L(t*, s*) > TC_u(t*, s*)$ .

The STG for a SCSI protocol controller is shown in Figure 3. In this example, the arc from  $q-$  to  $rdy-$  is found to be redundant. According to the timing analysis, the time difference is  $TD(rdy-, q-) = [15, 55]$  and the given timing constraint is  $TC(rdy-, q-) = [0, 5]$ . So, the condition of the above definition is satisfied as follows:  $TD_L(rdy-, q-) = 15 > TC_u(rdy-, q-) = 5$ , and the arc (i.e., the trigger signal) is found to be redundant.

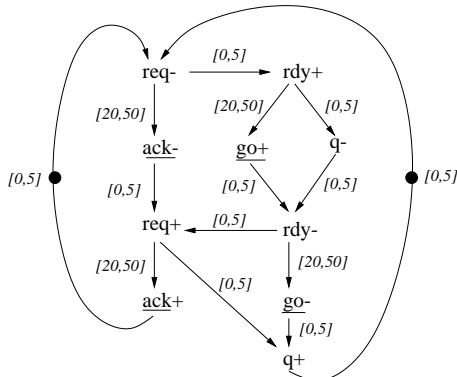


Fig. 3. The timed STG for a SCSI controller.

#### B. Finding the Timed Concurrency and Timed Causality 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 determine if two transitions are *timed concurrent*, the algorithm first finds *untimed concurrent* transitions. A *simple cycle* in an STG is a path from a transition  $u*$  to itself which does not cross an arc more than once. Two transitions  $s*$  and  $t*$  are untimed concurrent in a marked graph if there does not exist a simple cycle that includes both  $s*$  and  $t*$ . Untimed concurrent transitions can be found using reachability analysis on the STG (not the state space). To do this, the STG is cut at the initial marking and unfolded for a couple of cycles. For STGs which have no choice and are strongly connected, this is sufficient [6]. Then, each pair of transitions  $s*$  and  $t*$  can be checked to see if there exists a path from  $s*$  to  $t*$  or vice versa. If no such path exists, then the two transitions are untimed concurrent. Next, the algorithm checks the worst-case time difference between untimed concurrent transitions.

**Definition 2:** (Timed Concurrency) Two transitions  $s*$  and  $t*$  are timed concurrent if the following conditions are satisfied:

1.  $s*$  and  $t*$  are untimed concurrent,
2.  $TD_L(t*, s*) \leq 0$  and  $TD_U(t*, s*) \geq 0$ .

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  $TD(ack-, go+) = [-35, 30]$ . This time separation indicates that they can fire in either order. The two transitions  $go+$  and  $q-$  are untimed concurrent, also. However, they are not timed concurrent because  $TD(go+, q-) = [15, 50]$ . This time separation means that  $go+$  is always fired after  $q-$  is fired.

Next, 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 4 (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  $s*$  untimed causes a transition  $t*$  if  $t*$  is reachable from  $s*$  without visiting  $\overline{s*}$  in all the paths from  $s*$  to  $t*$ . In a timed STG, the algorithm finds the timed causality relations by analyzing reachability and worst-case time differences.

**Definition 3:** (Timed Causality) A transition  $s*$  timed causes  $t*$  if the following conditions are satisfied:

1.  $TD_L(t*, s*) > 0$  or there exists a path from  $s*$  to  $t*$  and  $TD_L(t*, s*) = 0$  (i.e.,  $s*$  fires before  $t*$ ),
2.  $TD_U(\overline{s*}, t*) \geq 0$ .

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  $TD(go+, q+) = [-60, -20]$  and  $TD(q-, go+) = [-50, -15]$ . So, the second condition of the definition is not satisfied. Instead,  $q-$  timed causes  $go+$  because  $TD(go+, q-) = [15, 50]$  and  $TD(q+, go+) = [20, 60]$ . In other words, we know that  $q-$  is always the last

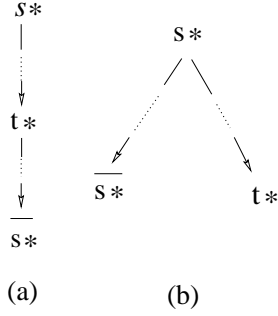


Fig. 4. Causality relation: (a)  $t^*$  occurs after  $s^*$  and before  $\overline{s^*}$ . (b)  $t^*$  occurs after  $s^*$  and concurrently with  $\overline{s^*}$ .

transition before  $go+$ , and thus  $q$  is always low before  $go$  goes high.

### C. Finding a Single Cube Network

In this step, the synthesis procedure synthesizes each interval network as a single cube. As stated above, the algorithm requires that STGs satisfy the complete state coding property and persistency. If STGs satisfy these conditions, each interval can be implemented as a single cube in a hazard-free manner [3]. If an STG does not satisfy these conditions, new signals must be added and the modified specification can be resynthesized. In [22], it is shown that specifications can be transformed to satisfy these conditions by inserting new signals.

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 5(a) shows the minimal interval networks for the SCSI controller.

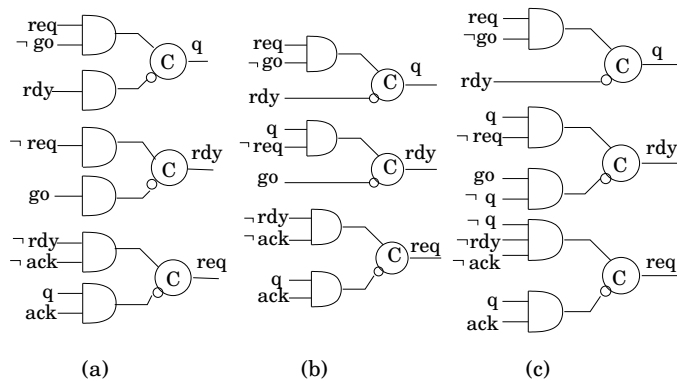


Fig. 5. (a) Minimal interval networks. (b) Timed implementation. (c) Speed-independent implementation.

All the trigger signals go high when  $u^*$  is enabled, and they remain high until  $u^*$  fires because the STG is assumed to be persistent. Therefore, 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 guaranteed

to be satisfied. For example, the solid thick line in Figure 6 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.

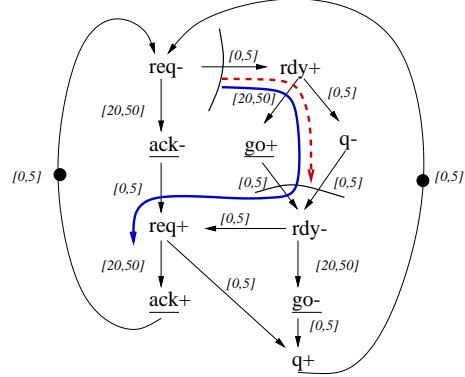


Fig. 6. A violation of the circuit model requirements.

Figure 7 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  $s^*$  timed causes  $t^*$ . 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 8 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.

The extra context signals are necessary to make the interval network satisfy the hazard-freedom requirements. And the *shrink* procedure finds the extra context signals by using the precedence graph. Therefore, the hazard-freedom requirements must be transformed into properties of the precedence graph. To formalize this, the *off-cube* is defined for a path from a source node to a destination node as follows.

*Definition 4:* Let  $P = x_1^* \rightarrow x_2^* \rightarrow \dots \rightarrow x_n^*$  (where  $x_1^*$  is a source node and  $x_n^*$  is a destination node) be a directed path in the precedence graph for the interval  $u^* \mapsto \overline{u^*}$ . The off-cube for the path  $P$  is defined as  $C = c_1 c_2 \dots c_n$ , where  $c_i = x_i$  if  $x_i^*$  is a falling transition or  $c_i = \neg x_i$  if  $x_i^*$  is a rising transition.

In an off-cube  $C = c_1 c_2 \dots c_n$ , the last literal,  $c_n$ , must be a trigger signal while the rest,  $c_1, c_2, \dots, c_{n-1}$ , are con-

```

shrink(STG G, transition u*)
{
  Precedence_graph (V, E) = ⟨∅, ∅⟩

  /* Find source and destination nodes */
  SN = {u*}
  ForEach s* in G
    If (u* ⇒ s* and s* ⇒  $\overline{u^*}$  and  $\overline{s^*}$  ⇒ u*)
      SN = SN ∪ {s*}
  DN = Find_destination_nodes(u*, G)
  V = SN ∪ DN

  /* Expand the precedence graph */
  ForEach unprocessed node s* in V
    ForEach t* in G
      If ((s* || t* or s* ⇒ t*) and t* ⇒  $\overline{s^*}$  and  $\overline{t^*}$  ⇒ u*
          and not (t* || u*))
        V = V ∪ {t*}
        E = E ∪ {(s*, t*)}

  ForEach si ∈ SN
    ForEach dj ∈ DN
      Ei,j = Find_all_possible_context_signals(si, dj)

  Find_minimal_context_signal_set(E);
}

```

Fig. 7. A sketch of the *shrink* procedure.

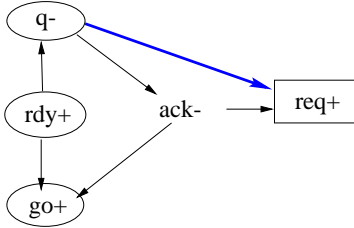


Fig. 8. Precedence graph for the interval  $rdy+ \mapsto rdy-$ .

text signals or possibly other trigger signals. The off-cube satisfies the conditions defined in the following lemma.

*Lemma 1:* Each off-cube  $C = c_1 c_2 \dots c_n$  defined in Definition 4 satisfies the following conditions:

1. It is on when  $u^*$  is enabled.
2. It is turned off after  $u^*$  is fired but before  $\overline{u^*}$  is enabled.
3. Once it is turned off, it remains off until the  $c_n$  is turned on.

*Proof:* (1) Each  $c_i$  is turned on when  $\overline{x_i^*}$  is fired and it is turned off when  $x_i^*$  is fired. According to the *shrink* procedure,  $x_1^*$  can be  $u^*$ . If  $x_1^*$  is  $u^*$  then  $c_1$  is on when  $u^*$  is enabled because  $c_1$  is turned on by firing the transition  $\overline{u^*}$  and it is turned off by firing the transition  $u^*$ . If  $x_1^*$  is not  $u^*$ , it satisfies the condition ( $u^* \Rightarrow x_1^*$  **and**  $x_1^* \Rightarrow \overline{u^*}$ ). This condition means that  $u^*$  occurs between  $\overline{x_1^*}$  and  $x_1^*$ . For each  $x_i^*$ , where  $2 \leq i \leq n$ , the condition ( $\overline{x_i^*} \Rightarrow u^*$  **and not** ( $x_i^* \parallel u^*$ )) is satisfied. This condition means that  $u^*$  occurs between  $\overline{x_i^*}$  and  $x_i^*$  for each  $x_i^*$ , where  $2 \leq i \leq n$ . Because  $u^*$  occurs between  $\overline{x_i^*}$  and  $x_i^*$  for each  $x_i^*$  which is not  $u^*$  and  $c_i$  is turned on by firing  $\overline{x_i^*}$ ,  $c_i$  is on when  $u^*$  is enabled. Therefore,  $C$  is on when  $u^*$  is enabled.

(2) Each  $c_i$  is turned off when the transition  $x_i^*$  is fired. If  $x_i^*$  is not  $u^*$ , it is fired after  $u^*$  is fired because the

condition ( $\overline{x_i^*} \Rightarrow u^*$  **and not** ( $x_i^* \parallel u^*$ )) is satisfied for each  $x_i^*$ . Therefore, each  $c_i$  is turned off after  $u^*$  is fired or at the same time when  $u^*$  is fired. For the source node  $x_1^*$ , the condition  $x_1^* \Rightarrow \overline{u^*}$  is satisfied. From the definition of timed causality,  $x_1^*$  fires before  $\overline{u^*}$ . Therefore,  $c_1$  is turned off before  $\overline{u^*}$ . Because each  $c_i$  is turned off after  $u^*$  is fired and  $c_1$  is turned off before  $\overline{u^*}$  is enabled, the cube  $C$  is turned off after  $u^*$  is fired but before  $\overline{u^*}$  is enabled.

(3) According to the *shrink* procedure, the condition  $x_{i+1}^* \Rightarrow \overline{x_i^*}$  is satisfied for each  $x_i^*$ , where  $1 \leq i \leq n-1$ . From the definition of timed causality,  $x_{i+1}^*$  fires before  $\overline{x_i^*}$ . This implies that  $c_i$  is turned on only after  $c_{i+1}$  is turned off. Because  $c_i$  is turned on only after  $c_{i+1}$  is turned off, once the cube  $C$  is turned off, it remains off until the signal  $c_n$  is turned on. ■

*Theorem 1:* Let  $D = \{d_1^*, d_2^*, \dots, d_n^*\}$  be the set of destination nodes of the precedence graph for an interval  $u^* \mapsto \overline{u^*}$ . Let  $C_i$  be the off-cube for the path from any source node to the destination node  $d_i^*$ . The cube  $C = C_1 C_2 \dots C_n$  satisfies all the hazard-freedom requirements for the interval  $u^* \mapsto \overline{u^*}$ .

*Proof:* (1) Let  $C_i = c_{i_1} c_{i_2} \dots c_{i_m}$ , where  $c_{i_m}$  is the signal for the destination node  $d_i^*$ . From Lemma 1,  $C_i$  is turned on when  $c_{i_m}$  is turned on. Because  $C$  is a conjunction of all  $C_i$ , it is turned on when all  $c_{i_m}$  are turned on, for  $1 \leq i \leq n$ . Each  $c_{i_m}$  is a trigger signal because each  $d_i^*$  is the reverse transition of a non-redundant enabling transition of  $u^*$ . Because trigger signals are turned on when  $u^*$  is enabled,  $C$  is turned on when  $u^*$  is enabled.

(2) From Lemma 1, each  $C_i$  is turned off after  $u^*$  is fired but before  $\overline{u^*}$  is enabled. Therefore  $C$  is turned off after  $u^*$  is fired but before  $\overline{u^*}$  is enabled.

(3) From Lemma 1, once each  $C_i$  is turned off, it remains off until  $c_{i_m}$  is turned on. Because  $C$  is a conjunction of all  $C_i$ , once  $C$  is turned off, it remains off until all  $c_{i_m}$  are turned on. All  $c_{i_m}$  are turned on when  $u^*$  is enabled again. Therefore, once  $C$  is turned off, it remains off until  $u^*$  is enabled again. ■

If there is more than one path for destination nodes, there are many sets of extra context signals which can make the interval network satisfy the hazard-freedom requirements. To minimize the number of extra context signals, 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. 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. While our method does not suffer from state explosion, it can suffer from path explosion during this step. Fortunately, it appears that the number of paths in the precedence graphs do not grow as

fast as the number of states in the state graph.

In the precedence graph for the interval  $rdy+ \mapsto rdy-$ , shown in Figure 8, 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 5(b). Figure 9(a) shows the precedence graph for the interval  $req+ \mapsto req-$ . The shortest path in this precedence graph from a source node to both destination nodes is  $ack+ \rightarrow rdy+$ . Since both of these transitions already correspond to trigger signals, no additional context signals are necessary for the logic to set  $req$ .

If instead, we use the original untimed concurrency and untimed causality relations, the precedence graph for  $req+ \mapsto req-$  would be as shown in figure 9(b). In this graph, the shortest path is  $ack+ \rightarrow q+ \rightarrow rdy+$  which implies that  $\neg q$  must be added as an additional context signal. Figure 5(c) shows the complete speed-independent implementation derived by the direct synthesis algorithm for the untimed case (note that the explicit state approach in ATACS[6] produces the same circuit). 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. The set network of the signal  $req$  also has one more literal because the path,  $ack+ \rightarrow q+ \rightarrow rdy+$ , in the precedence graph for a speed-independent circuit is longer than the path,  $ack+ \rightarrow rdy+$ , in the precedence graph for a timed circuit.

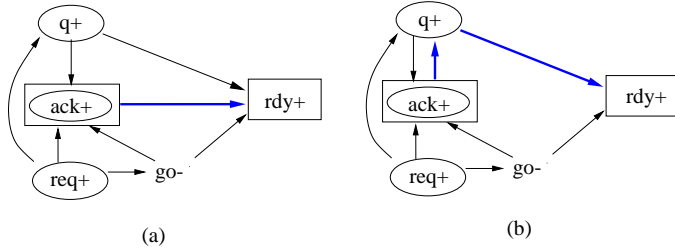


Fig. 9. Precedence graph for the interval  $req+ \mapsto req-$ , (a) for a timed circuit and (b) a speed-independent circuit.

#### D. 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 10 shows a sketch of the *expand* procedure. It finds the extra signals by constructing a precedence graph and finding paths. It is similar to the *shrink* procedure

except the conditions for constructing a precedence graph. One of the extra signals should be turned on before any signal of the single cube is turned off. So, the transition which occurs before any falling transitions of the signals of the single cube is added as a source node. The extra signals should be turned off when  $\overline{u^*}$  is enabled and they should remain off until  $u^*$  is enabled again. So, the previous reverse transition of the non-redundant enabling transition of  $\overline{u^*}$  is added as a destination node. After finding source and destination nodes, the graph is expanded using the conditions in the algorithm. After constructing the precedence graph, the extra signals are found by searching paths from each source node to each destination node. A candidate set for the extra signals of the OR gate are found by selecting one path for each destination node and including the corresponding signals for each node of the path. Because there can be more than one candidate set for the extra signals of the OR gate, the algorithm finds, in a similar manner to the *shrink* procedure, a minimal number of extra signals by searching the entire solution space. Below we prove that a multi-cube interval network constructed by the *expand* procedure satisfies the requirements for safely removing a memory element.

expand(STG  $G$ , transition  $u^*$ , cube  $C$ )

```

{
  Precedence_graph  $\langle V, E \rangle = \langle \emptyset, \emptyset \rangle$ 

  Let  $C = c_1 c_2 \dots c_n$ , where  $c_i = t_i$  or  $c_i = \neg t_i$ 
  Let us assume that  $t_i^* \Rightarrow u^*$  (if  $c_i = t_i$  then  $t_i^*$ 
  is a rising transition else it is a falling transition)
  /* Find source and destination nodes */
  Foreach  $s^*$  in  $G$ 
    If ( $u^* \Rightarrow s^*$  and  $\overline{s^*} \Rightarrow \overline{u^*}$  and ( $t_i^* \Rightarrow s^*$  and not ( $\overline{t_i^*} \parallel s^*$ )
      for all  $i$ ))
       $S_N = S_N \cup \{s^*\}$ 
    If (Is_a_non_redundant_enabling_transition( $s^*, \overline{u^*}$ ))
       $D_N = D_N \cup \{\overline{s^*}\}$ 
   $V = S_N \cup D_N$ 

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

  Foreach  $s_i \in S_N$ 
    Foreach  $d_j \in D_N$ 
       $E_{i,j} = \text{Find\_all\_possible\_context\_signals}(s_i, d_j)$ 

  Find_a_minimal_context_signal_set( $E$ );
}

```

Fig. 10. A sketch of the *expand* procedure.

*Lemma 2:* Let  $C$  be the single cube interval network found by the *shrink* procedure for the interval  $u^* \mapsto \overline{u^*}$ . Let  $P = x_1^* \rightarrow x_2^* \rightarrow \dots \rightarrow x_n^*$  (where  $x_1^*$  is a source node and  $x_n^*$  is a destination node) be a directed path in the precedence graph derived by the *expand* procedure for the interval. The on-cube for the path  $P$  is defined as  $D = d_1 + d_2 + \dots + d_n$ , where  $d_i = x_i$  if  $x_i^*$  is a rising transition or  $d_i = \neg x_i$  if  $x_i^*$  is a falling transition. The

on-cube  $D$  satisfies the following conditions:

1. It is turned on after  $u^*$  is fired but before  $C$  is turned off.
2. Once it is turned on, it remains on until  $d_n$  is turned off.

*Proof:* (1) Let  $C = c_1c_2\dots c_n$ , where  $c_i = t_i$  or  $c_i = \bar{t}_i$ . Let us assume that  $t_i^* \Rightarrow u^*$ . If  $c_i = t_i$  then  $t_i^*$  is a rising transition else it is a falling transition. The cube  $C$  is turned on when all  $t_i^*$  transitions occur and it is turned off when any transition  $\bar{t}_i^*$  occurs. According to the *expand* procedure, the source node  $x_1^*$  satisfies the condition ( $u^* \Rightarrow x_1^*$  and  $\bar{x}_1^* \Rightarrow \bar{u}^*$ ). This condition means that  $x_1^*$  occurs after  $u^*$ . Because  $d_i$  is turned on by  $x_i^*$  and is turned off by  $\bar{x}_i^*$ ,  $d_1$  is turned on after  $u^*$  is fired. Also, it satisfies the condition ( $t_i^* \Rightarrow x_1^*$  and not ( $\bar{t}_i^* \parallel x_1^*$ ) for all  $i$ ). This condition means that  $x_1^*$  occurs after each  $t_i^*$  but before each  $\bar{t}_i^*$ . This implies that  $d_1$  is turned on before each  $c_i$  is turned off. Therefore,  $D$  is turned on after  $u^*$  is fired but before  $C$  is turned off

(2) According to the *expand* procedure, the condition  $x_{i+1}^* \Rightarrow \bar{x}_i^*$  is satisfied for  $1 \leq i \leq n-1$ . From the definition of timed causality,  $x_{i+1}^*$  fires before  $\bar{x}_i^*$  is enabled. This implies that  $d_{i+1}$  is turned on before  $d_i$  is turned off. Because  $d_i$  is turned off only after  $d_{i+1}$  is turned on, once the cube  $D$  is turned on, it remains on until the signal  $d_n$  is turned off. ■

*Theorem 2:* Let  $N = \{d_1^*, d_2^*, \dots, d_n^*\}$  be the set of destination nodes of the precedence graph for an interval  $u^* \mapsto \bar{u}^*$ . Let  $D_i$  be the on-cube for the path from any source node to the destination node  $d_i$ . Let  $C$  be the single cube interval network found by the *shrink* procedure. The cube  $D = C + D_1 + D_2 + \dots + D_n$  is turned on when  $u^*$  is enabled and is turned off when  $\bar{u}^*$  is enabled.

*Proof:* Let  $D_i = d_{i_1}d_{i_2}\dots d_{i_m}$ , where  $d_{i_m}$  is the signal for the destination node  $d_i^*$ . From Lemma 2, each  $D_i$  is turned on after  $u^*$  but before  $C$  is turned off. Therefore  $D$  is turned on when  $C$  is turned on, that is, when  $u^*$  is enabled. Once each  $D_i$  is turned on, it remains on until  $d_{i_m}$  is turned off. Therefore, once  $D$  is turned on, it remains on until all  $d_{i_m}$  are turned off. Because each  $\bar{d}_{i_m}$  is a trigger signal for the interval  $\bar{u}^* \mapsto u^*$ ,  $\bar{u}^*$  is enabled when all  $\bar{d}_{i_m}$  are turned off. Therefore,  $D$  is turned off when  $\bar{u}^*$  is enabled. ■

For the SCSI controller, no memory elements can be eliminated. So, the circuit in Figure 5(b) is the final synthesis result. Figure 11 shows an STG for the packet forwarding controller which is included in the Post Office communication chip [13]. In this specification, the interval  $ack- \mapsto ack+$  can be implemented as multi-cube interval networks. Figure 12 shows the precedence graph derived by the *expand* procedure for the interval  $ack- \mapsto ack+$ . In this graph, the source node  $y0-$  is also a destination node. So, the minimal number of extra signals is  $\neg y0$ . Figure 13(a) shows the final implementation for the packet forward controller. Figure 13(b) shows the implementation using the timed state space exploration method in [6]. Figure 13(c) and (d) show speed-independent implementations by the direct method and the state space exploration

method, respectively. These implementations show that the area of timed circuits tends to be smaller than that of speed-independent circuits. Also, it is notable that the area of the timed circuit synthesized by the direct method is the smallest, since this method can find a multi-cube combinational network *ack*.

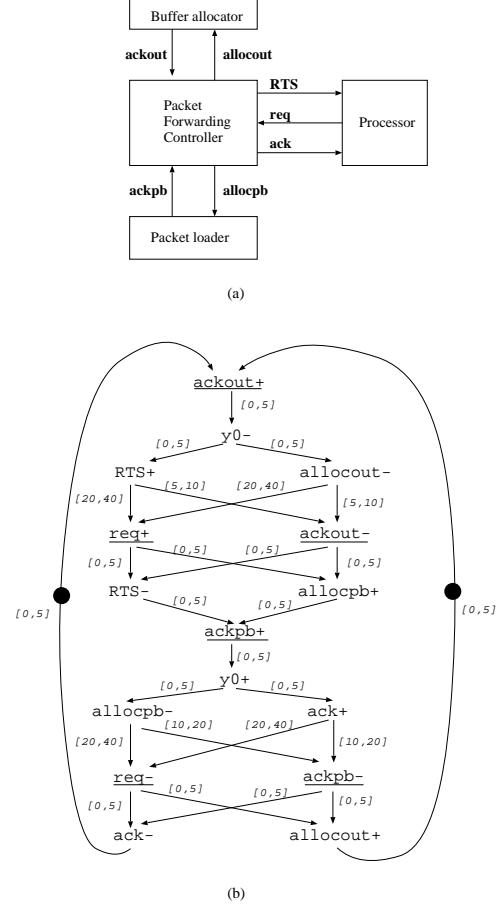


Fig. 11. (a) The block diagram for the packet forward controller. (b) An STG for the packet forward controller.

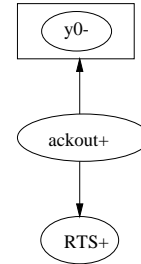


Fig. 12. Precedence graph for the interval  $ack- \mapsto ack+$ .

#### IV. EXTENSIONS OF THE SYNTHESIS PROCEDURE

In the previous section, the synthesis procedure for only a subset of the supported specifications has been discussed to simplify the explanation. This section extends the synthesis procedure for specifications which have multiple occurrences of a signal transition per single cycle. Also, we

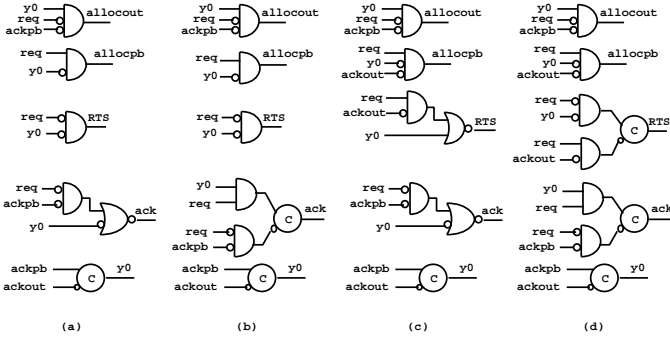


Fig. 13. Synthesized circuits for the packet forwarding controller: (a) timed direct method, (b) timed state space exploration method, (c) untimed direct method, (d) untimed state space exploration method.

extend it to specifications which do not satisfy the unique state coding property but satisfy the complete state coding property. In such a specification, there may be some intervals which should be implemented by one interval network. Finally, we extend the synthesis procedure to specifications which have free-choice behavior.

#### A. Multiple Occurrences of a Signal Transition

The *shrink* and *expand* procedure need to be changed for specifications which have multiple occurrences of each signal transition per single cycle. The extended *expand* procedure is similar to that of the previous section except that only single occurrence transitions are added in the precedence graph. If a signal whose transition occurs many times per single cycle is used as an extra signal, the output of the interval network may turn on outside the interval. So, only the single occurrence transitions are considered.

For the *shrink* procedure, more changes are necessary. Figure 14 shows the extended *shrink* procedure. Let's consider an interval network for the interval  $u*_i \mapsto \bar{u}*_i$ . Let the subscript  $i$  of a transition denote the  $i$ -th occurrence of the transition. Let us assume that the order of transitions of a signal  $s$  is as follows:  $s_{+1}, s_{-1}, s_{+2}, s_{-2}, \dots, s_{+n}, s_{-n}$ . A difference with the *shrink* procedure of the previous section is that source nodes are found separately for each destination node. Let  $e*_k$  be a non-redundant enabling transition of  $u*_i$ . Then, the transition  $\bar{e}*_k$  becomes a destination node in the precedence graph. The corresponding trigger signal for  $e*_k$  is turned off by the transition  $\bar{e}*_k$  and is turned on again by the transition  $e*_k$ . To satisfy the hazard-freedom requirements of the target circuit model, one of the context signals or other trigger signals should be turned off before the trigger signal is turned on again. If there is no such signal, the trigger signal can be turned off and be turned on again while other trigger and context signals remain on. This violates the requirement that the output of an interval network should remain off once it is turned off until  $u*$  is enabled again. This problem can be solved by selecting appropriate source nodes for each destination node. To ensure that one of the context signals or other trigger signals is turned off before the trigger is turned on again, the transition which occurs before

$e*_k$  should be added as a source node for the destination  $\bar{e}*_k$ . In addition, to satisfy the hazard-freedom requirements, the transition which occurs between  $u*_i$  and  $\bar{u}*_i$  should be added as a source node. In the extended *shrink* procedure, these two conditions are checked to find source nodes. Note that source nodes are found differently depending on whether  $e*_k$  and  $\bar{u}*_i$  are timed concurrent or not. If they are timed concurrent, there may not be any transition which satisfies both conditions. In such a case, it is necessary to include two source nodes which satisfy each of the two conditions respectively when finding paths to the destination node. So, source nodes for each destination node are classified into three groups,  $SA$ ,  $SB$ , and  $SC$ . The transitions which satisfy both conditions are included in the  $SC$  group. The transitions which occur between  $u*_i$  and  $\bar{u}*_i$  are included in the  $SA$  group. The transitions which occur between  $e*_k$  and  $e*_k$  are included in the  $SB$  group. If  $e*_k$  and  $\bar{u}*_i$  are not timed concurrent, only the  $SC$  group needs to be found. Note that the condition ( $u*_i \Rightarrow s*_j$  and not ( $s*_j \parallel \bar{u}*_i$ )) is used to check if  $s*_j$  occurs between  $u*_i$  and  $\bar{u}*_i$ . In the *shrink* procedure of the previous section, the condition ( $u* \Rightarrow s*$  and  $s* \Rightarrow \bar{u}*$ ) is used to check if  $s*$  occurs between  $u*$  and  $\bar{u}*$ .

After finding source and destination nodes, the graph is expanded in the same way as with the *shrink* procedure in the previous section. After constructing a precedence graph, the algorithm finds all possible sets of extra signals for each destination node finding all the paths from each source node to the destination node in the graph. If a source node is included in the group  $SC$ , then a path from the source node to the destination node represents a possible set of extra signals for the destination node. Otherwise, two paths should be considered where one source node is in the  $SA$  group and the other source node is in the  $SB$  group.

Figure 15 shows an STG example with multiple transitions of a signal per single cycle. In the figure, multiple occurrences of the transition  $a+$  are denoted as  $a+/1$  and  $a+/2$ . For the interval  $u+ \mapsto u-$ , there is one destination node,  $a-/2$ . The source nodes for the destination node are as follows:  $SA_{a-/2} = \{c+\}$ ,  $SB_{a-/2} = \{b+, a-/1\}$ ,  $SC_{a-/2} = \{u+\}$ . Figure 16 shows the precedence graph for the interval. In this graph, the two paths,  $c+ \rightarrow a-/2$  and  $b+ \rightarrow a-/2$ , result in a minimal number of extra signals  $\{-b-c\}$ .

#### B. Gate Sharing Between Intervals

If a specification satisfies the unique state coding property, each interval is implemented by one distinct interval network. If a specification does not satisfy the unique state coding property but satisfies the complete state coding property, some intervals may need to be implemented by the same interval network. That is, gates may need to be shared between the intervals.

To handle such a specification, the hazard-freedom requirements of the target circuit model are modified as follows. The interval network for a set of output transitions  $U = \{u*_1, u*_2, \dots, u*_n\}$  must satisfy the following require-

```

shrink(STG G, transition  $u*_i$ )
{
  Precedence_graph  $(V, E) = \langle \emptyset, \emptyset \rangle$ 

  /* Find source and destination nodes */
  ForEach  $e*_k$  (a non-redundant enabling transition of  $u*_i$ )
     $D_N = D_N \cup \{\overline{e*_k-1}\}$ 
     $SC_{\overline{e*_k-1}} = \{u*_i\}$ 
    If ( $e*_k+1 \parallel \overline{u*_i}$ )
      ForEach  $s*_j$  in  $G$ 
        If ( $u*_i \Rightarrow s*_j$  and not ( $s*_j \parallel \overline{u*_i}$ ) and  $\overline{s*_l} \Rightarrow u*_i$ 
          (for some  $l$ ) and (( $e*_k \Rightarrow s*_j$  or  $\overline{e*_k} \Rightarrow s*_j$ )
          and not ( $e*_k+1 \parallel s*_j$ )))
           $SC_{\overline{e*_k-1}} = SC_{\overline{e*_k-1}} \cup \{s*_j\}$ 
        Else if ( $u*_i \Rightarrow s*_j$  and not ( $s*_j \parallel \overline{u*_i}$ ) and  $\overline{s*_l} \Rightarrow u*_i$ 
          (for some  $l$ ))
           $SA_{\overline{e*_k-1}} = SA_{\overline{e*_k-1}} \cup \{s*_j\}$ 
        Else if ( $u*_i \Rightarrow s*_j$  and  $\overline{s*_l} \Rightarrow u*_i$  (for some  $l$ ) and
          (( $e*_k \Rightarrow s*_j$  or  $\overline{e*_k} \Rightarrow s*_j$ ) and not ( $e*_k+1 \parallel s*_j$ )))
           $SB_{\overline{e*_k-1}} = SB_{\overline{e*_k-1}} \cup \{s*_j\}$ 
      Else
        ForEach  $s*_j$  in  $G$ 
          If ( $u*_i \Rightarrow s*_j$  and not ( $s*_j \parallel \overline{u*_i}$ ) and  $\overline{s*_l} \Rightarrow u*_i$ 
            (for some  $l$ ) and (( $e*_k \Rightarrow s*_j$  or  $\overline{e*_k} \Rightarrow s*_j$ ) and
            not ( $e*_k+1 \parallel s*_j$ )))
             $SC_{\overline{e*_k-1}} = SC_{\overline{e*_k-1}} \cup \{s*_j\}$ 

           $SN_{\overline{e*_k-1}} = SA_{\overline{e*_k-1}} \cup SB_{\overline{e*_k-1}} \cup SC_{\overline{e*_k-1}}$ 
           $V = V \cup SN_{\overline{e*_k-1}}$ 

   $V = V \cup D_N$ 

  /* Expand the precedence graph */
  ForEach unprocessed node  $s*_i$  in  $V$ 
    ForEach  $t*_j$  in  $G$ 
      If (( $s*_i \parallel t*_j$  or  $s*_i \Rightarrow t*_j$ ) and  $t*_j \Rightarrow \overline{s*_i}$  and  $\overline{t*_k} \Rightarrow u*_i$ 
        (for some  $k$ ) and not ( $t*_j \parallel u*_i$ ))
         $V = V \cup \{t*_j\}$ 
         $E = E \cup \{(s*_i, t*_j)\}$ 

  ForEach  $d_i \in D_N$ 
    ForEach  $s_j \in SN_{d_i}$ 
       $E_{i,j} = \text{Find\_all\_possible\_context\_signals}(s_i, d_j)$ 

  Find\_a\_minimal\_context\_signal\_set( $E$ )
}

```

Fig. 14. A sketch of the *shrink* function for the specifications with multiple occurrences.

ments:

1. It is turned on when  $u*_i$  is enabled for each  $i = 1, 2, \dots, n$ .
2. Whenever it is turned on, it is turned off after  $u*_i$  is fired but before the next reverse transition  $\overline{u*_i}$  is enabled.
3. Once it is turned off, it remains off until  $u*_{i+1}$  is enabled.

Here, it is assumed that the order of transitions is  $u*_1, u*_2, \dots, u*_n$ .

In this paper, the transitions which should be implemented by the same interval network are found by using the *enabling cube refinement*. First, the smallest enabling cube is found for each transition of the specification. The smallest enabling cube is defined as follows.

*Definition 5:* (Smallest Enabling Cube) Let  $S = \{s_1, s_2, \dots, s_n\}$  be all the signals of the specification. For an output transition  $u*$ , the smallest enabling cube,  $SEC_{u*} =$

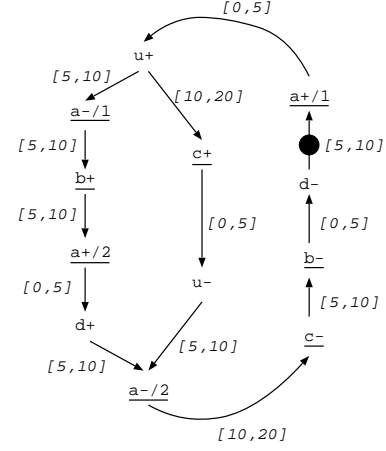


Fig. 15. An STG specification with multiple transitions of a signal per single cycle.

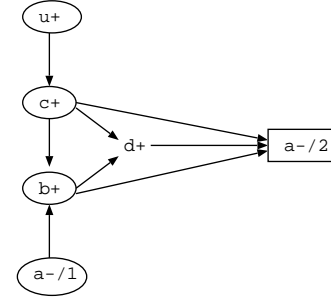


Fig. 16. The precedence graph for the interval  $u+ \mapsto u-$ .

$c_1 c_2 \dots c_n$ , is defined as follows:

1. If one of the transitions of the signal  $s_i$  is timed concurrent with  $u*$ , then  $c_i = X$ .
2. Otherwise, let  $s_i*$  be the transition which timed causes the transition  $u*$ . If  $s_i*$  is a rising transition then  $c_i = 1$ . If  $s_i*$  is a falling transition then  $c_i = 0$ .

Let  $u*_i$  and  $u*_j$  be two transitions of the signal  $u$  which are in the same direction. That is, both are rising or falling transitions. If  $SEC_{u*_i}$  and  $SEC_{u*_j}$  intersect in reachable states, then they should be implemented by one interval network. Two cubes  $SEC_{u*_i} = c_1 c_2 \dots c_n$  and  $SEC_{u*_j} = d_1 d_2 \dots d_n$  intersect if  $c_i = d_i$  for all  $i$ , where both  $c_i$  and  $d_i$  are not  $X$ . Figure 17 shows two STG segments, where  $u$  is an output signal to be synthesized. In Figure 17,  $SEC_{u+/1} = 1XX00$  and  $SEC_{u+/2} = 1XX00$  intersect with each other, where the set of signals is  $S = \{a, b, c, d, u\}$ . Also,  $SEC_{u+/3} = 1XX00$  and  $SEC_{u+/4} = 10100$  intersect with each other. Because  $SEC_{u+/1} = 1XX00$  does not include any unreachable state,  $SEC_{u+/1}$  and  $SEC_{u+/2}$  intersect in reachable states. However, we can easily detect that the cube  $SEC_{u+/3} = 1XX00$  is an overestimation of the enabling states for  $u+/3$  because the state 10100 which is unreachable is also covered. Therefore,  $SEC_{u+/3} = 1XX00$  and  $SEC_{u+/4} = 10100$  do not intersect in reachable states. Therefore, whereas  $u+/1$  and  $u+/2$  should be implemented by one interval network,  $u+/3$  and  $u+/4$  should be implemented separately.

In order to check if two cubes intersect in reachable

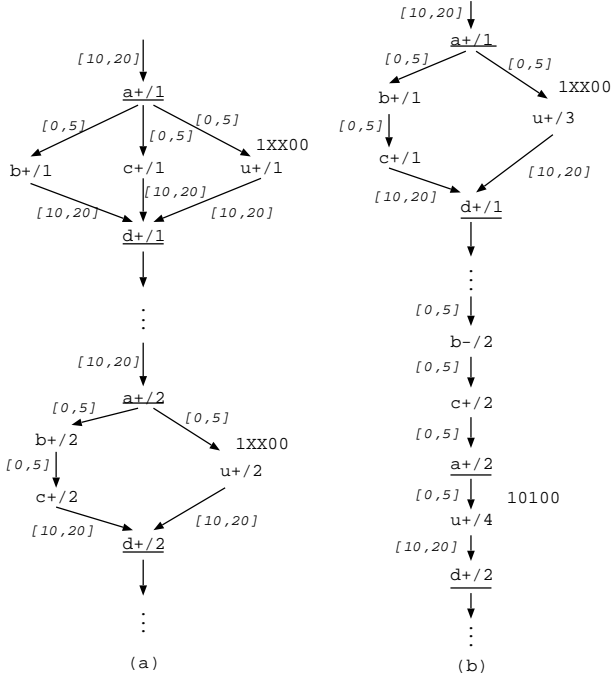


Fig. 17. Example STG segments.

states, the cube including unreachable states needs to be refined until the refined cubes do not include unreachable states. In Figure 18, a circle represents a *place* and the cube attached to a place represents the approximation of the states in which the place has a token. The cube for a place can be computed similarly with the smallest enabling cube for the transition. To obtain refined cubes, we use the fact that  $u+/3$  is timed concurrent with  $p1$ ,  $p2$ , and  $p3$ . The cube  $SEC_{u+/3}$  can be refined by intersecting it with the conjunction of the cubes of  $p1$ ,  $p2$ , and  $p3$ . Then, the refined cubes  $10000 + 11000 + 11100$  correctly covers the enabling states of  $u+/3$ . Repeated refinements may be needed if refined cubes still cover some unreachable states.

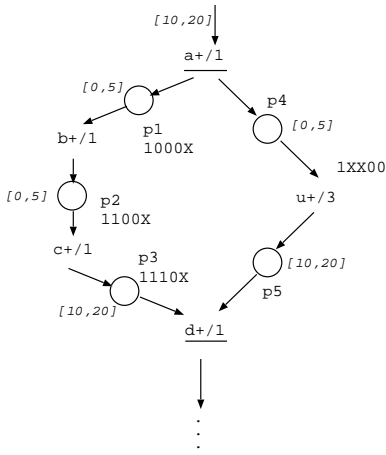


Fig. 18. Cube refinement.

The single cube implementation of the shared interval network is found by searching the trigger and context sig-

nals for each interval and merging them in one AND gate. The *shrink* procedure applied to each interval must be changed to determine destination nodes differently. Let  $u*_1 \mapsto \overline{u*}_1$ ,  $u*_2 \mapsto \overline{u*}_2$ , ...,  $u*_n \mapsto \overline{u*}_n$  be the shared intervals, and let us also assume that the transitions are ordered as  $u*_1, \overline{u*}_1, u*_2, \overline{u*}_2, \dots, u*_n, \overline{u*}_n$ . Then, for the interval  $u*_i \mapsto \overline{u*}_i$ , the destination nodes are the previous reverse transitions of the non-redundant enabling transitions of  $u*_{i+1}$ . After finding trigger signals and all possible candidates of context signals for each interval, it is checked if each candidate can be used in the final single cube implementation. If a context signal in one interval is not stable in other intervals, then it cannot be used as a context signal in the resultant single cube implementation. So, the candidate including such a context signal is removed from the candidate set. Then, all the possible candidates for the single cube implementation can be found using set multiplication, and a minimal solution is selected.

### C. Free-choice STGs

To extend the synthesis algorithm to a timed free-choice STGs, we could find the necessary relations between signal transitions by applying a timing analysis directly on the timed free-choice STG. In this paper, we instead decompose a timed free-choice STG into a set of marked graph (MG) components and apply timing analysis to each MG-component. By decomposing to MG-components, we can use the same efficient, heuristic timing analysis algorithm that we used above for marked graphs [6]. For an interval network, trigger signals and context signals are then found for each marked graph component individually. Then, they are merged into a single cube.

#### C.1 MG-decomposition

The divide-and-conquer approach using MG-decomposition for the synthesis of speed-independent circuits was suggested in [11]. In this approach, a free-choice specification is decomposed into a set of MG-components and each component is solved separately. It is based on the following lemma which was proved in [11].

*Lemma 3:* Let  $S$  be an output signal of the specification  $G$ ,  $U$  be a set of transitions of  $S$  to be implemented by an interval network, and  $R$  be a MG-component set of  $G$ . An interval network  $F$  satisfies the hazard-freedom requirement of the target circuit model if and only if:

1. For each MG-component in  $R$  which covers a subset of  $U$ , the interval network  $F$  satisfies all the requirements of the target circuit model.
2. For each MG-component in  $R$  which does not cover any transitions in  $U$  but still covers some transition of  $S$ ,  $F$  is always off.
3. For each component in  $R$  which does not cover any transition of  $S$ ,  $F$  is either always off or always on.

The synthesis procedure can handle free-choice STGs which run one MG-component at a time. This means that any transition which is not included in a MG-component should not be untimed concurrent with the transitions of

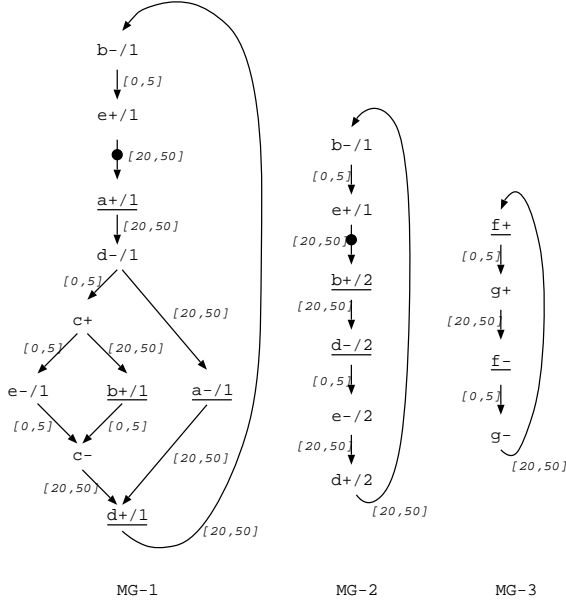


Fig. 19. MG-components for the free-choice STG example in Figure 1.

the component. In other words, all the signals whose transitions are not included in a MG-component should be stable while the component is running. Most benchmarks satisfy this condition. In the original description, the free-choice transitions must be transitions of input signals. In this paper, the synthesis procedure allows dummy transitions as free-choice transitions to allow choices between sets of input transitions.

Because the complexity of the synthesis procedure based on MG-decomposition depends on the size of the MG-component set, it is necessary to minimize the size of the set of MG-components. If any transition is not untimed concurrent with free-choice transitions, then only the set of MG-components in which each transition is covered at least once is sufficient for the synthesis. Most benchmarks satisfy this condition and they require a manageable number of MG-components in general. The STG in Figure 1 has three MG-components as shown in Figure 19.

### C.2 Finding the Timing Relations

The synthesis procedure finds the required timing information for a timed free-choice STG by applying the timing analysis on each MG-component. The timed concurrency relation can be found by only examining each MG-component because any two transitions which are timed concurrent are included in the same MG-component. To find the timed causality relation between any two transitions of a free-choice STG, it is necessary to apply timing analysis to the entire STG. However, the synthesis procedure does not require all timed causality relations. For each MG-component, the synthesis procedure requires the timed causality relations between any two transitions which are included in the MG-component. These relations are found by applying the timing analysis for each MG-component. In addition, it requires the timed causality relations be-

tween any two transitions  $s^*$  and  $t^*$ , where  $t^*$  is a transition of the MG-component,  $s^*$  is a transition of the free-choice STG, and there does not exist any transition on  $s$  in the MG-component. In this case, if  $s^*$  untimed causes  $t^*$  then  $s^*$  also timed causes  $t^*$ . This is due to the fact that the next reverse transition of  $s^*$  needs to be included in the MG-component for the untimed causality relations and timed causality relations to be different. The required untimed causality relations are found by using reachability analysis for the entire free-choice STG. Therefore, required timed causality information can be found without doing timing analysis on the entire timed free-choice STG. Redundant trigger signals can be found in each MG-decomposition because trigger signals should be untimed concurrent for some of them to be redundant and the concurrency occurs only within the MG-component. For the STG example in Figure 1, the arc from  $e-/1$  to  $c-$  is found to be redundant because  $TD_L(c-, e-/1) = 15 > TC_u(c-, e-/1) = 5$  according to the timing analysis on component MG-1.

### C.3 Finding a Single Cube Interval Network

Let  $U$  be a set of transitions to be implemented by an interval network. A single cube interval network for  $U$  is synthesized based on Lemma 3. For each MG-component which covers a subset of  $U$ , the single cube interval network should satisfy all the requirements of the target circuit model. The trigger and context signals which are necessary to satisfy the requirements of the target circuit model are found by the synthesis procedure for a marked graph. For each MG-component which does not cover any transitions in  $U$  but still covers some transition of  $S$ , the single cube network should be always off. Figure 20 shows the procedure which finds the cubes which are always off for a MG-component. In the procedure,  $u^*$  denotes a transition of  $U$ , and  $G$  denotes a MG-component. It first finds the signals which stay zero-stable for the MG-component. If the following two conditions are satisfied, then a signal  $s$  is found to be a zero-stable signal for  $G$ : i) all the transitions of  $G$  are timed caused by some falling transition of  $s$ , and ii) all the transitions of  $U$  are timed caused by some rising transition of  $s$  and any transition of  $s$  is not timed concurrent with the transitions of  $U$ . In a similar way, a signal  $\neg s$  can be found to be a zero-stable signal. Note that if  $s$  or  $\neg s$  is zero-stable for  $G$ , then any transition of the signal  $s$  does not occur in  $G$ .

After finding the zero-stable signals, it finds cubes which are always off for the MG-component by constructing a precedence graph and finding paths in the precedence graph. If any trigger signal is a zero-stable signal for the MG-component, no extra signals are necessary to make the single cube interval network off for the MG-component. Therefore, we don't have to apply the second method for such a case.

The problem of finding the cubes which are always off for a MG-component is very similar to the problem of finding extra signals in the *shrink* procedure in Section III-C. In the *shrink* procedure, the extra signals are used to make the interval network be off from a source node to a des-

```

find_off_cubes(MG-component  $G$ , a set of transition  $U$ )
{
   $C = \text{Find\_zero\_stable\_signals}(G, U)$ 
  If (a trigger signal is included in  $C$ ) return

  Unfold  $G$  into two cycles

  Precedence_graph  $\langle V, E \rangle = \langle \emptyset, \emptyset \rangle$ 

  /* Find source and destination nodes */
  ForEach  $s^*$  in  $G$ 
  If ( $\overline{s^*_k} \Rightarrow u^*$  (for some  $k$ ) and  $s$  is stable with  $U$ )
     $S_{N_{s^*}} = \langle s^*, 0 \rangle$ 
     $D_{N_{s^*}} = \langle s^*, 1 \rangle$ 
     $V = V \cup \{S_{N_{s^*}}\} \cup \{D_{N_{s^*}}\}$ 

  /* Expand the precedence graph */
  ForEach unprocessed node  $\langle s^*, 0 \rangle$  in  $V$ 
  ForEach  $t^*$  in  $G$ 
  If ( $(s^* \parallel t^* \text{ or } s^* \Rightarrow t^*)$  and  $t^* \Rightarrow \overline{s^*}$  and  $\overline{t^*_k} \Rightarrow u^*$ 
    (for some  $k$ ) and not ( $t^* \parallel \overline{s^*}$ ) and  $t$  is stable with  $U$ )
  If (is_in_same_cycle( $s^*, t^*$ ))
     $V = V \cup \{\langle t^*, 0 \rangle\}$ 
     $E = E \cup \{\langle s^*, 0 \rangle, \langle t^*, 0 \rangle\}$ 
  Else
     $V = V \cup \{\langle t^*, 1 \rangle\}$ 
     $E = E \cup \{\langle s^*, 0 \rangle, \langle t^*, 1 \rangle\}$ 

  ForEach unprocessed node  $\langle s^*, 1 \rangle$  in  $V$ 
  ForEach  $t^*$  in  $G$ 
  If ( $(s^* \parallel t^* \text{ or } s^* \Rightarrow t^*)$  and  $t^* \Rightarrow \overline{s^*}$  and  $\overline{t^*_k} \Rightarrow u^*$ 
    (for some  $k$ ) and not ( $t^* \parallel \overline{s^*}$ ) and  $t$  is stable with  $U$ )
  If (is_in_same_cycle( $s^*, t^*$ ))
     $V = V \cup \{\langle t^*, 1 \rangle\}$ 
     $E = E \cup \{\langle s^*, 1 \rangle, \langle t^*, 1 \rangle\}$ 

  ForEach  $s_i \in S_N$ 
   $C = C \cup \text{Find\_all\_possible\_candidates}(s_i, d_i)$ 
}

```

Fig. 20. A sketch of the *find\_off\_cubes* procedure.

transition node. If there is a path from a source node to a destination node, then the off-cube (defined in Definition 4) for the path is always off from the source node to the destination node. Therefore, if the source and destination nodes are modified so that the path between them covers a whole cycle of the MG-component, then the corresponding cube for the path is always off for the MG-component. The procedure *find\_off\_cubes* unfolds a MG-component into two cycles to find such a source node and a destination node. In the procedure,  $\langle s^*, 0 \rangle$  and  $\langle s^*, 1 \rangle$  denotes the transition  $s^*$  in the first cycle and in the second cycle, respectively. If  $\langle s^*, 0 \rangle$  and  $\langle s^*, 1 \rangle$  are selected as a source node and a destination node, respectively, then the path between them cover the whole cycle of the MG-component. The procedure *find\_off\_cubes* selects  $\langle s^+, 0 \rangle$  and  $\langle s^+, 1 \rangle$  as a source node and a destination node respectively if the signal  $\neg s$  can be used as a context signal or a trigger signal of the single cube interval network. The necessary condition for the signal  $\neg s$  to be used as a context or trigger signal is that it should be turned on when the output transition is enabled and it should remain on until the output transition is fired. In a similar way, the transition  $\langle s^-, 0 \rangle$  and  $\langle s^-, 1 \rangle$

can be selected as a source node and a destination node.

After finding source and destination nodes, it expands the precedence graph. The conditions for the expansion are similar to those of the *shrink* procedure except that the condition for checking the appropriate cycle for the nodes. The condition for the expansion, ( $(s^* \parallel t^* \text{ or } s^* \Rightarrow t^*)$  **and**  $t^* \Rightarrow \overline{s^*}$ ), can be represented as in Figure 21(a). This condition can be transformed in three different forms in the unfolded graph depending on how they are unfolded as shown in Figure 21(b), (c), or (d). If they have the relations in Figure 21(b) or (c), then  $s^*$  and  $t^*$  are in the same cycle. So, an arc from  $\langle s^*, 0 \rangle$  to  $\langle t^*, 0 \rangle$  is added in the precedence graph. Also, an arc from  $\langle s^*, 1 \rangle$  to  $\langle t^*, 1 \rangle$  is added. If they have the relations in Figure 21(d), then  $s^*$  and  $t^*$  are in the different cycle. So, an arc from  $\langle s^*, 0 \rangle$  to  $\langle t^*, 1 \rangle$  is added.

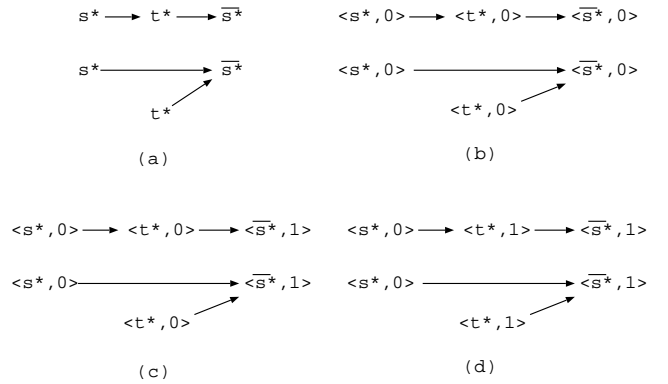


Fig. 21. Relations for the expansion of the precedence graph: (a) in the original MG-component. (b), (c), and (d) in the unfolded graph.

For each MG-component which does not cover any transition of  $S$ , the single cube network should be either always off or always on. The cubes which are always off are found by the procedure *find\_off\_cubes*. A cube  $C = c_1 c_2 \dots c_n$  is always on if every signal  $c_i$  ( $i = 1, 2, \dots, n$ ) stays on for the MG-component. So, the synthesis procedure finds all the signals which stay one-stable for the MG-component. Later, it is necessary to check if all the trigger signals and context signals are one-stable signals for the MG-component. If the following two conditions are satisfied, then a signal  $s$  is found to be a one-stable signal for a MG-component  $G$ : i) all the transitions of  $G$  are timed caused by some rising transition of  $s$ , and ii) all the transitions of  $U$  are timed caused by some rising transition of  $s$  and any transition of  $s$  is not timed concurrent with the transitions of  $U$ . In a similar way, a signal  $\neg s$  can be found to be a one-stable signal. After finding all the possible solutions for each MG-component, the minimal single cube interval network is found by searching the entire solution space.

For the STG example in Figure 1, the interval network for  $c^+ \mapsto c^-$  should satisfy the hazard-free requirements for component MG-1 and should be always off or always on for components MG-2 and MG-3. The set of cubes which satisfy the hazard-freedom requirements for component MG-1 is  $\{\neg de, \neg d\neg b\}$ . For component MG-2, the one-stable sig-

nals are  $\neg g$ ,  $\neg c$ , and  $\neg f$ . And the zero-stable signal is  $\neg a$ . Because  $\neg a$  is not a trigger signal for the interval network, the cubes of two or more signals which always stay off for the MG-component need to be found. Figure 22 shows the precedence graph for the unfolded graph of component MG-2. There are three paths  $\langle b+ /2, 0 \rangle \rightarrow \langle d+ /2, 0 \rangle \rightarrow \langle b+ /2, 1 \rangle$ ,  $\langle d+ /2, 0 \rangle \rightarrow \langle b+ /2, 1 \rangle \rightarrow \langle d+ /2, 1 \rangle$ , and  $\langle e- /2, 0 \rangle \rightarrow \langle d+ /2, 0 \rangle \rightarrow \langle b+ /2, 1 \rangle \rightarrow \langle e- /2, 1 \rangle$ . And the off cubes for the paths are  $\neg d \neg b$  and  $\neg d \neg b e$ . For component MG-3, the one-stable signal is  $\neg c$  and the zero-stable signals are  $\neg d$ ,  $\neg b$ ,  $a$ , and  $e$ . Because the trigger signal  $\neg d$  is zero-stable signal for this MG-component, it is not necessary to find other off cubes. From the above results, the minimal single cube is found to be  $\neg d \neg b$ . Figure 23 (a) shows the timed implementation and Figure 23 (b) shows the speed-independent implementation for the STG in Figure 1.

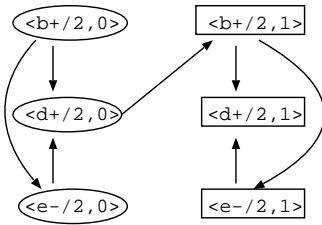


Fig. 22. A precedence graph for finding off cubes.

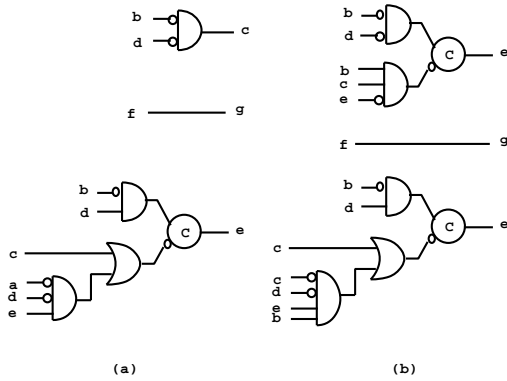


Fig. 23. Implementations for the STG example in Figure 1: (a) Timed implementation. (b) Speed-independent implementation.

## V. COMPLEXITY

The algorithms for MG-decomposition have a polynomial-time complexity. 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 paths in the graph. The number of paths may be determined by several factors. One main factor is

the size of solution space, that is, the number of possible candidates for context signals. It may increase exponentially with respect to the number of signals which can be used as context signals. So, the direct method suggested in this paper does not remove the exponential complexity of the state graph based method but moves the complexity of the state space into the solution space. However, the size of the solution space seems to increase slowly with respect to the size of STG specification. It seems to highly depend on the structure of the STG and the relations between the signal transitions. For most benchmarks, the solution space does not increase exponentially.

## VI. EXPERIMENTAL RESULTS

Table 1 shows the experimental results for marked graph benchmarks. 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 many SCSI controller specifications in parallel. Since they are independent, the state of one SCSI controller can coexist with many combinations of states of the others leading quickly to state explosion. Also, we synthesized a multi-stage, series connected FIFO [23]. Table 2 shows the experimental results for free-choice STG benchmarks. Many timed STG examples are converted from burst-mode AFMSM specifications which were derived from academic and industrial design examples such as *postoffice* [13], *cache-ctrl* [18], *stetson* [19], *hp-ir* [19], *pscsi* [21], *sscsi* [17], and *dram-ctrl* [17]. Unfortunately, the benchmarks in both tables come from speed-independent and burst-mode benchmark suites, so most do not have delay numbers provided. One notable exception is the FIFO benchmark where delay numbers were derived from information in [23]. In cases in which no timing information is available, we assumed circuit delays of  $[0, 5]$  and environment delays of  $[20, 50]$ .

The experimental results show that our synthesis method does not exhibit the state explosion problem and in practice for large examples achieves significant reductions in synthesis time as compared to previous methods. 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 remains almost constant because they are connected serially. For SCSI controllers, the size of the precedence graphs in-

TABLE I  
EXPERIMENTAL RESULTS.

Example	STG (S/N/A)	States	PG (G/N/A)	ATACS		Direct Method	
				Total Lit.	CPU time (sec)	Total Lit.	CPU time (sec)
atod	7/14/17	24	10/5/9	13	0.04	13	0.06
converta	5/14/16	19	8/6/8	20	0.06	20	0.04
full	4/8/12	16	4/4/8	8	0.03	8	0.03
half	4/8/11	14	4/4/6	8	0.04	8	0.03
hybridf	8/16/26	80	10/8/34	14	0.07	14	0.07
master-read	18/28/40	2108	16/8/25	34	3.38	34	0.15
mp-fwd-pkt	8/16/26	22	14/5/7	16	0.06	14	0.05
nak-pa	10/20/24	58	17/6/16	20	0.06	17	0.07
nowick	6/16/21	20	12/6/7	18	0.04	12	0.05
rpdft	5/22/22	22	6/8/9	12	0.07	12	0.10
cstat	3/6/8	8	2/3/4	4	0.02	4	0.03
elatch	8/16/26	44	10/8/24	12	0.09	12	0.07
elatchX	26/52/103	9475	40/14/61	44	105.5	45	1.33
elatchP	8/16/30	88	10/7/23	14	0.11	16	0.06
fib	8/16/37	64	9/6/14	17	0.12	20	0.08
ifreq1	5/10/16	28	4/5/8	7	0.03	6	0.03
lapbN	8/16/24	97	13/6/12	19	0.12	19	0.08
mul2c	7/14/38	65	6/6/20	6	0.09	7	0.05
slatch	8/16/26	37	10/7/22	10	0.06	10	0.06
vmeP	5/10/21	19	6/5/8	6	0.06	6	0.04
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

creases linearly with the size of STG.

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 did not finish due to the lack of memory. The area of the synthesized circuits is 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 *elatchP*, the direct method produces a bigger circuit because the heuristic timing analysis does not find some redundant triggers. In all examples where the circuits differed, the results produced by the direct method were successfully verified using the timed circuit verifier *orbits* [14].

If all the timing constraints in the timed STG specification are given as  $[0, \infty]$ , the synthesized circuit is speed-independent. The top 10 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 [24]. The CPU time with Petrify is 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 [24] 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 [9]. Both programs were run on the same SUN Sparc20 with 128MB of main memory. The CPU time for the tool from [9] 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 [9] uses a heuristic approach, the CPU times are quite similar. Whereas the method in [9] cannot synthesize 70 untimed SCSI controllers because it runs out of memory, the method in this paper can synthesize 90 untimed SCSI controllers on the same SUN Sparc20.

## VII. 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 in practice. Currently, the synthesis algorithm can handle only timed free-choice STG specifications which have a single cube implementation. Future work includes the extension of the algorithm to a wider class of specifications such as specifications with a unique choice or that require multi-cube implementations.

## REFERENCES

- [1] T.-A. 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 Transactions on VLSI Systems*, pp. 106-119 June 1993.
- [7] K.J. Lin, J.W. Kuo and C.S. Lin, "Direct synthesis of hazard-free asynchronous circuits from STGs based on lock relation and MG-decomposition approach", *In Proceedings of European Design and Test Conference*, pp. 178-183, Nov. 1994.
- [8] C. Ykman-Couvreur, B. Lin, G. Goossens and H. De Man, "Synthesis and optimization of asynchronous controllers based on extended lock graph theory", *In Proceedings of European Conference on Design Automation (EDAC)*, pp. 512-517, Feb. 1993.
- [9] 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.
- [10] 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.
- [11] 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
- [12] 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
- [13] K.S. Stevens, S.V. Robinson, and A.L. Davis, "The Post Office - Communication Support for Distributed Ensemble Architectures", *In Proceeding of 6th International Conference on Distributed Computing Systems*, pp. 567-571, 1986
- [14] T. G. Rokicki, *Representing and Modeling Circuits*, Ph.D. thesis, Stanford University, 1993.
- [15] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng, "POSET: Timing and Its Application to the Synthesis and Verification of Gate-Level Timed Circuits", *IEEE Transactions on Computer-Aided Design*, Vol. 18, No. 6, pp. 769-786, Jun. 1999.
- [16] H. Hulgaard and S.M. Burns, "Bounded Delay Timing Analysis of a Class of CSP Programs with Choice", *In Proceedings of Advanced Research in Asynchronous Circuits and Systems*, pp. 2-11, Nov. 1994.
- [17] S.M. Nowick, K.Y. Yun and D.L. Dill, "Practical Asynchronous Controller Design", *International Conference on Computer Design (ICCD)*, pp. 341-345, Oct. 1992.
- [18] S.M. Nowick, M.E. Dean, D.L. Dill and M. Horowitz, "The Design of a High-Performance Cache Controller: a Case Study in Asynchronous Synthesis", *Integration, the VLSI journal*, vol.15, no. 3, pp.241-262, Oct. 1993.
- [19] A. Marshall, B. Coates and Polly Siegel, "Designing an Asynchronous Communications Chip", *IEEE Design & Test of Computers* vol. 11, no. 2, pp. 8-21, 1994.
- [20] K.Y. Yun, "Synthesis of Asynchronous Controllers for Heterogeneous Systems", *Ph.D thesis*, Dept. of Elec. Eng., Stanford University, Aug. 1994.
- [21] K.Y. Yun and D.L. Dill, "Automatic Synthesis of 3D Asynchronous State Machines", *International Conference on Computer Aided Design (ICCAD)*, pp. 576-580, Nov. 1992
- [22] A. Kondratyev, M. Kishinevsky, B. Lin, P. Vanvekbergen, and A. Yakovlev, "Basic Gate Implementation of Speed-independent Circuits", *In Proceedings of Design Automation Conference*, pp. 56-62 June, 1994.
- [23] C.E. Molnar, I.W. Jones, B. Coates, and J. 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.
- [24] 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 and Systems*, Vol. E80-D, No. 3, Mar. 1997, pp. 315-325.

TABLE II  
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)
DME	9/30/34	28	43/8/10	39	0.14	34	0.26
cache-ctrl	36/369/1098	N/A	1240/35/186	N/A	N/A	811	390.30
cache-ctrl-s2io1	52/738/2196	N/A	2364/68/344	N/A	N/A	1601	4944.34
cache-ctrl-s2io2	72/738/2196	N/A	2480/68/268	N/A	N/A	1622	3910.09
dff	4/18/24	18	14/5/4	16	0.05	16	0.09
dlatch	3/12/16	12	6/3/1	6	0.04	6	0.04
dram-ctrl	13/50/97	107	45/11/31	46	4.86	46	0.50
factorial	17/48/91	72	26/9/14	45	0.67	45	0.49
hp-ir-it-ctrl	13/47/75	71	60/14/25	60	0.45	60	0.66
hp-ir-two-ticks	6/23/33	26	16/5/3	8	0.10	7	0.09
po-alloc-outbound	9/22/23	21	19/7/7	17	0.12	17	0.15
po-rcv-setup	5/15/20	14	8/3/2	8	0.05	8	0.04
po-sbuf-read-ctl	8/24/29	20	28/9/20	13	0.07	13	0.10
po-sbuf-send-ctl	8/24/29	33	28/9/20	36	0.14	36	0.16
po-sbuf-send-pkt2	9/27/35	28	35/6/6	23	0.11	23	0.21
p SCSI-ircv	9/28/48	44	17/6/12	27	0.19	27	0.12
p SCSI-isend	10/49/81	74	48/11/19	59	0.41	59	1.02
p SCSI-trcv	8/26/39	35	15/5/8	23	0.12	23	0.09
p SCSI-trcv-bm	10/44/67	70	41/9/15	37	0.26	37	0.81
p SCSI-tsend	10/45/72	65	50/11/17	55	0.34	55	0.81
p SCSI-tsend-bm	10/49/81	80	57/12/21	63	0.82	63	1.05
rlm	6/13/14	12	10/3/3	7	0.04	7	0.05
sm	3/10/12	8	9/3/3	7	0.06	7	0.03
sscsi-isend-bm	11/48/82	76	56/13/27	63	44.18	63	1.20
sscsi-isend-csm	11/37/69	64	32/10/21	51	38.27	51	0.26
sscsi-trcv-bm	11/49/81	75	56/13/26	63	45.49	64	1.24
sscsi-trcv-csm	11/38/68	63	30/10/19	47	38.67	47	0.24
sscsi-tsend-bm	11/46/72	65	53/13/22	60	1.82	60	0.99
sscsi-tsend-csm	11/34/52	53	30/10/17	41	0.28	41	0.22
stetson-p3	6/22/31	24	16/5/3	8	0.09	7	0.11
vme-master	12/52/75	162	22/14/46	31	0.92	31	0.52
vme-read	13/34/46	124	17/11/40	25	0.48	24	0.28
vme-write	15/36/53	243	19/12/56	29	1.91	28	0.37
vmebus-arb	5/30/41	42	11/5/4	10	0.08	10	0.09

**Sung-Tae Jung** received the M.S. and Ph.D. degrees in computer engineering from Seoul National University, Seoul, Korea, in 1989 and 1994, respectively. He has been an Assistant Professor in the Department of Computer Engineering, Wonkwang University, Iksan City, Jeonbuk, Korea, since 1995. His current research interests are asynchronous circuit design, logic synthesis of asynchronous circuits, and high-level synthesis of asynchronous systems.

**Chris J. Myers** received the B.S. degree in electrical engineering and Chinese history in 1991 from the California Institute of Technology, Pasadena, CA, and the M.S.E.E. and Ph.D. degrees from Stanford University, Stanford, CA, in 1993 and 1995, respectively. He is an Associate Professor in the Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT. His current research interests are innovative architectures for high performance and low power, algorithms for the computer-aided analysis and design of real-time concurrent systems, analog error control decoders, formal verification, and asynchronous circuit design. Dr. Myers received an NSF Fellowship in 1991, an NSF CAREER award in 1996, and a best paper award at Async99.

algorithms for the computer-aided analysis and design of real-time concurrent systems, analog error control decoders, formal verification, and asynchronous circuit design. Dr. Myers received an NSF Fellowship in 1991, an NSF CAREER award in 1996, and a best paper award at Async99.