Synthesis of Timed Asynchronous Circuits

Chris J. Myers and Teresa H.-Y. Meng

***Abstract*—In this paper we present a systematic procedure to synthesize timed asynchronous circuits using timing constraints dictated by system integration, thereby facilitating natural interaction between synchronous and asynchronous circuits. In addition, our timed circuits also tend to be more efficient, in both speed and area, compared with traditional asynchronous circuits. Our synthesis procedure begins with a cyclic graph specification to which timing constraints can be added. First, the cyclic graph is unfolded into an infinite acyclic graph. Then, an analysis of two finite subgraphs of the infinite acyclic graph detects and removes redundancy in the original specification based on the given timing constraints. From this reduced specification, an implementation that is guaranteed to function correctly under the timing constraints is systematically synthesized. With practical circuit examples, we demonstrate that the resulting timed implementation is significantly reduced in complexity compared with implementations previously derived using other methodologies.**

I. Introduction

THE DESIGN OF *timed asynchronous circuits* has recently gained much attention because of the increasing need for asynchronous circuits in mixed synchronous/asynchronous environments. Inherent in these environments are *timing constraints* (gate, wire, and environment delay information) which circuits must satisfy and can exploit to optimize the implementation. Existing asynchronous design techniques either cannot handle systems with timing constraints, or do not fully utilize the information contained in them. This paper presents a methodology to synthesize asynchronous circuits that utilizes timing constraints throughout the synthesis procedure. As a result, our *timed circuits* retain the same behavior with less circuit complexity than earlier implementations.

Many methodologies have been proposed for the synthesis of *speed-independent circuits* [1]–[4]. Speed-independent circuits are very robust since they are guaranteed to work independent of the delays associated with their gates, but they can be overly conservative when timing constraints are known. Timed circuits, on the other hand, are only guaranteed to work if the delays fall in the range given in the timing constraints of the specification. Utilizing these timing constraints, we trade robustness to variations in delays for significant reductions in circuit complexity.

Speed-independent circuits are restricted to interfaces where their environment only changes inputs in response to changes of outputs. Inputs from a synchronous circuit often do not satisfy this restriction. In order to address this problem, *fundamental mode* synthesis methods have been used [5]–[8], which assume the environment will wait long enough for the circuit to stabilize before inputs are changed. Timing analysis must be performed after synthesis, and appropriate delays may need to be added to guarantee that this requirement is satisfied. Since these methods limit the concurrency within a circuit and do not fully utilize available timing constraints, they may result in circuits that are larger and slower than necessary.

Methods have been proposed to use timing constraints to synthesize timed circuits [9], [10]; however, most techniques apply timing constraints after synthesis only to verify that hazards do not exist. If hazards are detected, delay elements are added to avoid them, degrading the performance of the implementation. It was shown in [4] that the more conservative speed-independent model while resulting in somewhat larger circuits actually produces faster circuits compared with the timed circuits described in [10]. This surprising result can be attributed to the fact that these timed circuits often need to have delay elements added to the critical path to remove hazards.

Our synthesis procedure uses the timing constraints at the outset to enhance performance while minimizing circuit complexity. In several practical examples, we show that significant reductions in circuit complexity (measured in terms of literal count needed for the implementation) as compared to previous designs can be achieved using very conservative timing constraints. In particular, in a memory management unit designed for use with a real asynchronous microprocessor [11], [12], the circuit complexity is reduced by over 50% over the speed-independent implementation. Circuit performance is also enhanced, not only because we have reduced circuit area and do not use delay elements, but also because we are able to synthesize a more concurrent specification without adding state variables. An example of a DRAM controller to be used with a synchronous processor and DRAM array is presented to illustrate a design that cannot be done speed-independently. Circuit complexity is also reduced as compared to previous fundamental mode designs [13], [7].

This paper contains five sections. Section II describes our specification language and timing analysis algorithm. Section III discusses our synthesis procedure. Section IV presents several practical examples. Section V gives our conclusions.

II. Timing Analysis on Timed Specifications

A wide variety of methodologies for specification of asynchronous circuits have been proposed. They can be roughly grouped into three classes: language based, such as *communicating sequential processes* (CSP) [1]; graph based, such as *signal transition graphs* (STG) [2]; and finite-state machine based, such as *burst-mode state machines* (BSM) [6]. At

Manuscript received . This work was supported by an NSF fellowship, ONR Grant N00014-89-J-3036, and research grants from the Center for Integrated Systems, Stanford University and the Semiconductor Research Corporation under Contract 92-DJ-205.

The authors are with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

IEEE Log Number 9209313.

1063-8210/93$03.00 © 1993 IEEE

a high-level, CSP provides a very concise representation for large designs such as the microprocessor described in [11]. It is well suited for non-deterministic behavior, but it can be difficult to specify concurrency within a process. On the other hand, STG provides a good representation of concurrency within a process, but it is cumbersome to use for large designs and cannot specify arbitrary non-deterministic behavior. Neither of these representations are good for specifying asynchronous circuits in a synchronous environment. BSM has been successfully used for such applications [13], which was made possible by assuming fundamental mode as opposed to the other two specifications which use the speed-independent model. None of these specification methods incorporates timing constraints.

We chose to use a specification language, the *event-rule (ER) system* [14], which is easily derivable from CSP, STG, and BSM and incorporates timing constraints. It is shown in [14] that specifications that are not *disjunctive* or *non-deterministic* can be directly transformed into an ER system. A specification is disjunctive if there exists a transition in the specification that is specified to occur after either one transition or another, but it does not have to be preceded by both. A specification is non-deterministic if the circuit behavior is determined by a choice made by either the environment or the circuit. Derivation of ER systems from each specification method described above (i.e., CSP, STG, and BSM) is illustrated through an example. While our synthesis procedure does not presently allow non-deterministic specifications, it is shown, by way of an example, that some non-deterministic specifications can be transformed into deterministic specifications which can then be synthesized.

In order to synthesize timed circuits, timing analysis must be used on the ER system specification to deduce timing information necessary to detect redundancy in the specification from the given timing constraints. More specifically, in timed circuits, the timing information needed is the minimum and maximum difference in time between any two events (i.e., signal transitions) in a circuit specification. Polynomial-time algorithms have been developed [15], [16] to determine the difference in time between any two events in an acyclic graph. Circuit specifications, however, are normally cyclic. Therefore, to apply these algorithms to circuit synthesis, these results must be expanded to handle cyclic specifications. Recently, an algorithm has been proposed that finds these time differences in cyclic graphs in exponential-time [17]. In this paper, we propose instead a polynomial-time heuristic algorithm which is sufficient for our purposes. Our algorithm unfolds the cyclic graph into an infinite acyclic graph and then examines only two finite acyclic subgraphs of the infinite graph to determine a sufficient bound on the time difference between two events.

2.1. Event-Rule System

The ER system was introduced in [14] for performance analysis of asynchronous circuits. It was modified to incorporate bounds on the timing constraints and introduced as a specification language for timed circuits in [18]. An ER system is composed of a set of atomic actions, *events*, and the causal dependencies between them, *rules*, and it can be compactly represented using an *event-rule (ER) schema*.

1) Events: An event is defined as " . . . an action which one can choose to regard as indivisible—it either has happened or has not . . . " [19]. In circuits, events are *transitions* of signals from one value to another. There are two transitions associated with each signal s in a specification, namely, $s \uparrow$ where $\uparrow$ denotes that the signal s is changing from a low to high value, and $s \downarrow$ where $\downarrow$ denotes that the signal s is changing from a high to low value.

2) Rules A rule is a causal dependency between two events. Each rule is composed of an *enabling event*, an *enabled event*, and a *bounded timing constraint*. Informally, a rule states that the enabled event cannot occur until the enabling event has occurred. If two rules enable the same event then that event cannot occur until *both* enabling events have occurred. This causality requirement is termed *conjunctive*.

The bounded timing constraint places a lower and upper bound on the timing of a rule. A rule is said to be *satisfied* if the amount of time which has passed since the enabling event has exceeded the lower-bound of the rule. A rule is said to be *expired* if the amount of time which has passed since the enabling event has exceeded the upper bound of the rule. An event cannot occur until *all* rules enabling it are satisfied. An event must always occur before *every* rule enabling it has expired. Since an event may be enabled by multiple rules, it is possible that the difference in time between the enabled event and some enabling events exceeds the upper-bound of their timing constraints, but not for all enabling events. These timing constraints are the same as the *max constraints* described in [15] and the *type 2 arcs* described in [16].

Finding timing constraints for a specification is not a trivial task. Rules can be categorized into *environment rules* (i.e., the enabled event is a transition of an input signal) and *internal rules* (i.e., the enabled event is a transition of a state variable or output signal). Timing constraints for environment rules can be determined from interface specifications or datapath delay estimates. For internal rules, the problem is much more difficult since the timing constraints cannot be known until the circuit is synthesized, but the circuit cannot be synthesized without given timing constraints. To solve this problem, the designer should estimate the maximum delay for the gates in the library to be used and set the upper-bound of the timing constraint in each internal rule to this value. The lower-bound of the timing constraint should usually be set to 0 since optimizations could potentially reduce the gate to nothing more than a wire. After a circuit is generated, it should be analyzed using a timing analysis tool to verify that the timing constraints used are correct. If the circuit violates the timing constraints, it must be resynthesized with more conservative timing constraints (larger upper bounds in this case). In order to avoid resynthesis, conservative values should be used for timing constraints on internal rules at the outset. In the design of interface circuits and other controllers, inputs often are from off-chip or from a datapath. In these cases, the lower bound of the timing constraint on environment rules is large compared with the upper bound of the timing constraints on internal rules. Therefore, a conservative estimate for internal gate delays does not significantly affect the complexity of the timed implementation.

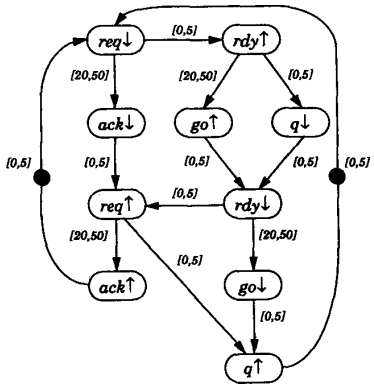


Fig. 1. The cyclic constraint graph for an SCSI protocol controller (courtesy of [20]).

3) Event-Rule Schema: An ER system can be specified using an ER schema and initialization information described in the next subsection. An ER schema defines a *cyclic constraint graph* which is a weighted marked graph in which the vertices are the events, the arcs are the rules, the weights are the bounded timing constraints, and the initial marking is given by the value of ε. Each rule of the form $\langle e, f, \varepsilon, \tau \rangle$ is represented in the graph with an arc connecting the enabling event e to the enabled event f. The arc is weighted with the bounded timing constraint τ. In other words, each rule corresponds to a graph segment, $e \xrightarrow{\tau} f$ (or $e \overset{\tau}{\bullet\!\!\rightarrow} f$ when the rule is initially marked, i.e., $\varepsilon = 1$). A cyclic constraint graph is similar to a STG in which timing constraints have been added to the arcs. The ER schema is defined more formally as follows:

Definition 2.1: (Event-Rule Schema): An event-rule schema is a pair $\langle E', R' \rangle$ where E' is a finite set of events, and R' is a finite set of rules. Each rule is denoted as $\langle e, f, \varepsilon, \tau \rangle$, where e and f are two events, ε is defined to be 1 if the rule has an initial marking and 0 otherwise, and $\tau = [l, u]$ where l is the lower bound and u is the upper bound of the timing constraint on the rule.

As an example, a SCSI protocol controller, originally specified with a STG [20], is specified by its cyclic constraint graph as shown in Fig. 1. An example of a rule in this constraint graph is between the two events $q \downarrow$ and $rdy \downarrow$, which is of the form $\langle q \downarrow, rdy \downarrow, 0, [0, 5] \rangle$.

Our synthesis procedure requires that each event in an ER schema is uniquely identified. This led to the restriction in [18] of only *one rising and one falling transition* of each signal per cycle in the specification. To remove this restriction in this paper, each occurrence of the rising and falling transition in a cycle is given a unique name. For example, a signal s specified to rise and fall twice in a cycle, is renamed to s_1 for the first rising and falling transitions and s_2 for the second. These events are treated separately during the timing analysis; however, they are recombined during synthesis as illustrated in an example later.

Another requirement is that the cyclic constraint graph is *well-formed*. A cyclic constraint graph is well formed if it is strongly connected, every cycle has the sum of the ε values along the cycle greater than or equal to 1, and for every event there exists a cycle including the event in which the sum of the ε values is equal to 1 [17]. Many specifications are not well-formed, but such specifications can often be synthesized by transforming them into ones which are well-formed as shown later in an example.

4) Event-Rule System: To construct the ER system, the cyclic constraint graph is transformed into an *infinite acyclic constraint graph*. Each event in the ER schema is mapped onto an infinite number of event occurrences, each corresponding to a different occurrence of that event. The rules are similarly mapped. Thus in the infinite acyclic constraint graph, each rule occurrence $\langle e, f, i, \varepsilon, \tau \rangle$ corresponds to a graph segment, $\langle e, i-\varepsilon \rangle \xrightarrow{\tau} \langle f, i \rangle$. The occurrence-index i is used to denote each separate occurrence of an event or rule in the ER schema. The first occurrence has $i = 0$, and i increments with each following occurrence. The occurrence-index offset ε is the difference in the occurrence-index of the enabled event f and the enabling event e. For each rule occurrence, the value of the occurrence-index offset ε is the same as the value of the initial marking ε for the corresponding rule in the ER schema.

A special *reset* event is added to the set of events in order to model the reset of the circuit. For each initially marked rule (i.e., $\varepsilon = 1$) with enabled event f, a *reset rule* is added between the *reset* event and the event f. This rule models special timing constraints on the initial occurrence of the event f. Effectively, the acyclic constraint graph is constructed by cutting the cyclic constraint graph at the initial marking and unfolding the graph an infinite number of cycles. The result is an ER system as defined below:

Definition 2.2: (Event-Rule System): Given the event-rule schema $\langle E', R' \rangle$, define an event-rule system to be a pair $\langle E, R \rangle$ where each event occurrence $\langle e, i \rangle$ in E where $i \geq 0$ represents an occurrence of an event e in E', and each rule occurrence $\langle e, f, i, \varepsilon, \tau \rangle$ in R where $i \geq \varepsilon$ is an occurrence of a rule $\langle e, f, \varepsilon, \tau \rangle$ in R'. The event $\langle \text{reset}, 0 \rangle$ is added to E. For each rule in R' in which $\varepsilon = 1$, a rule of the form $\langle \text{reset}, f, 0, 0, \tau_0 \rangle$ is added to R.

The specified circuit behavior is defined by simulating the acyclic constraint graph using the *timed firing rule* given below:

Definition 2.3: (Timed Firing Rule): Given that $t(\langle f, i \rangle)$ is the exact time of the event occurrence $\langle f, i \rangle$, it can take on any value within the bound defined in terms of the times of the event occurrences that enable it. The bound can be given as follows:

$$\max_{\langle e, f, i, \varepsilon, \tau \rangle \in R} \{t(\langle e, i-\varepsilon \rangle) + l\} \leq t(\langle f, i \rangle) \leq \max_{\langle e, f, i, \varepsilon, \tau \rangle \in R} \{t(\langle e, i-\varepsilon \rangle) + u\}.$$

A subgraph of the unfolded infinite acyclic constraint graph for the SCSI protocol controller is shown in Fig. 2. An example of a rule occurrence in this ER system is between the two event occurrences $\langle q \downarrow, 0 \rangle$ and $\langle rdy \downarrow, 0 \rangle$, which is of the form $\langle q \downarrow, rdy \downarrow, 0, 0, [0, 5] \rangle$. According to the timed firing rule, the event occurrence $\langle rdy \downarrow, 0 \rangle$ cannot occur until both the event occurrences $\langle q \downarrow, 0 \rangle$ and $\langle go \uparrow, 0 \rangle$ have occurred, and it must occur before 5 time units have elapsed since both the event occurrences occurred.

2.2. Timing Analysis

In order to transform an ER system specification into a timed circuit, our synthesis procedure requires a timing analysis algorithm to determine the minimum and maximum time difference between any two events. We have developed an efficient polynomial-time timing analysis algorithm to determine a sufficient estimate of these time differences based on only two finite subgraphs of the infinite acyclic constraint graph.

1) Worst-Case Time Difference: A *time difference* is a bound in the amount of time between two event occurrences (see Definition 2.4). The *worst-case time difference* is a bound on the minimum and maximum difference in time between two events for any occurrence (see Definition 2.5).

Definition 2.4: (Time Difference): Given two event occurrences, $\langle u, i-j\rangle$ and $\langle v, i\rangle$, and the occurrence-index offset between them j where $j \geq 0$, the time difference between these two event occurrences is the strongest bound $[L_i, U_i]$ such that:

$$L_i \leq t(\langle v, i\rangle) - t(\langle u, i-j\rangle) \leq U_i$$

Definition 2.5: (Worst-Case Time Difference): Given two events, u and v, and the occurrence-index offset between them j where $j \geq 0$, the worst-case time difference between these two events, $[L, U]$, is:

$$L = \min_{i \geq j}\{L_i\} \text{ and } U = \max_{i \geq j}\{U_i\},$$

where $[L_i, U_i]$ is the time difference for each occurrence of u and v with offset j (as defined in Definition 2.4).

2) Algorithm to Estimate Worst-Case Time Difference In our ER systems, a pair of events has an infinite number of occurrences; however, it is possible to analyze a finite number of occurrences to find a sufficient *estimate of the worst-case time difference* as defined in Definition 2.6.

Definition 2.6: (Estimate of the Worst-Case Time Difference): Given the worst-case time difference $[L, U]$ between two events, an estimate of the worst-case time difference is any $[L', U']$ such that $L' \leq L$ and $U' \geq U$.

Given two events u and v and an occurrence-index offset between them j, Algorithm 2.3 determines an estimate of the worst-case time difference between them by constructing two finite acyclic subgraphs to be analyzed by Algorithm 2.2. The first subgraph includes only events and rules with indexes $i-1$ and i for some arbitrary value of $i > 0$. A *source* event is added to this subgraph, and each rule with $\varepsilon = 1$ and with index $i-1$ is replaced with a rule from the *source* event to the enabled event with a timing constraint of $[0, \infty]$. This construction guarantees that no timing assumptions are made about previous cycles which are not modeled in our finite graph. For the special case when $i = 0$, another subgraph is constructed which includes only events and rules with $i = 0$. We prove later that the analysis of these two subgraphs yields an estimate of the worst-case time difference.

These two subgraphs are acyclic and finite so the algorithms described in [15] and [16] can be used to find the time difference between any two event occurrences $\langle u, i-j\rangle$ and $\langle v, i\rangle$ in these graphs. The function *MaxDiff* (defined recursively in Algorithm 2.1 [16]) is used to find the upper-bound of the time difference U_i. *MaxDiff* is also used to find the minimum time difference L_i since $MinDiff(\langle u, i-j\rangle, \langle v, i\rangle) = (-1) * MaxDiff(\langle v, i\rangle, \langle u, i-j\rangle)$ [15], [16]. The estimate of the worst-case time difference returned by Algorithm 2.3 is the minimum of the lower-bounds and the maximum of the upper bounds of the time differences for the ith and 0th occurrence. In our synthesis procedure, only time differences with values of $j = 0$ or $j = 1$ are of interest, so this algorithm does not produce a tight bound for $j > 1$. Also, since the worst-case time difference is only defined over values of i where $i \geq j$, the 0th occurrence only needs to be considered if $j = 0$. Finally, since this algorithm is called repeatedly in the synthesis procedure, the graphs are created only once for a given circuit, and once a time difference is calculated for a particular pair of event occurrences, it is stored in a table such that it need not be recalculated.

Algorithm 2.1 (Find Max Time Difference in an Acyclic Graph):

int MaxDiff(acyclic graph G; event occurrences $\langle u, i-j\rangle$, $\langle v, i\rangle$) {
  If there is no path from $\langle v, i\rangle$ to $\langle u, i-j\rangle$ then
    maxdiff $= \max_{\langle e,v,i,\varepsilon,\tau_{ev}\rangle \in R} \{MaxDiff\,(G, \langle u, i-j\rangle, \langle e, i-\varepsilon\rangle) + u_{ev}\}$;
  Else
    maxdiff $= \min_{\langle e,u,i-j,\varepsilon,\tau_{eu}\rangle \in R} \{MaxDiff\,(G, \langle e, i-j-\varepsilon\rangle, \langle v, i\rangle) + l_{eu}\}$;
  Return(maxdiff);
}

Algorithm 2.2 (Find Time Difference in an Acyclic Graph)

bound TimeDiff(acyclic graph G; event occurrences $\langle u, i-j\rangle$, $\langle v, i\rangle$) {
  $L_i = (-1) * MaxDiff(G, \langle v, i\rangle, \langle u, i-j\rangle)$;
  $U_i = MaxDiff(G, \langle u, i-j\rangle, \langle v, i\rangle)$;
  Return($[L_i, U_i]$);
}

Algorithm 2.3:
(Find Estimate of the Worst-Case Time Difference in a Cyclic Graph):

bound WCTimeDiff(ER system $\langle E, R\rangle$; events u, v; occurrence-index offset j) {
  If ($j > 1$) then Return($[-\infty, \infty]$);
  Else {
    Construct subgraph G from $\langle E, R\rangle$ using only events and rules with indices $i-1$ and i for an arbitrary $i > 0$ and exclude rules with enabling event $\langle$reset, 0$\rangle$;
    Add event $\langle source, i-1\rangle$ to graph G;
    For each rule of the form $\langle e, f, i-1, 1, \tau\rangle$ in graph G, replace it with $\langle source, f, i-1, 0, [0, \infty]\rangle$;
    $[L_i, U_i] = TimeDiff(G, \langle u, i-j\rangle, \langle v, i\rangle)$;
    If ($j == 1$) then Return($[L_i, U_i]$);
    Else{
      Construct subgraph G′ from $\langle E, R\rangle$ using only events and rules with index $i = 0$;

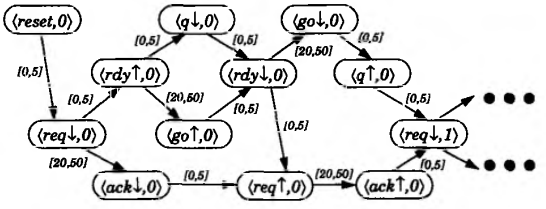


Fig. 2. A subgraph of the infinite acyclic constraint graph for the SCSI protocol controller.

$[L_0, U_0] = \mathit{TimeDiff}(G', \langle u, 0\rangle, \langle v, 0\rangle);$
$L' = \mathit{min}(L_i, L_0);$
$U' = \mathit{max}(U_i, U_0);$
$\mathit{Return}([L', U']);$
}}}

For the example shown in Fig. 2, the estimate of the worst-case time difference found by Algorithm 2.3 between the two events $rdy \downarrow$ and $q \downarrow$ with occurrence-index offset $j = 0$ is the bound $[15, 55]$. This means that $rdy \downarrow$ always occurs at least 15 units of time after $q \downarrow$, but no more than 55 units of time after $q \downarrow$.

Proof of Correctness: Theorem 2.1 shows that the bound for the ith occurrence, $[L_i, U_i]$, found in Algorithm 2.3 is an estimate for all $i > 0$. Therefore, combining this with the actual time difference for $i = 0$ results in an estimate of the worst-case time difference.

Theorem 2.1: Algorithm 2.3 determines an estimate of the worst-case time difference between two events.

Proof: In order to show that Algorithm 2.3 returns an estimate of the worst-case time difference, we must show that the following inequalities hold: $L' \leq L$ and $U' \geq U$ (from Definition 2.6). If $j > 1$ then Algorithm 2.3 returns $[L', U'] = [-\infty, \infty]$ which trivially satisfies Definition 2.6. If $j = 1$ then it returns $[L', U'] = [L_i, U_i]$. If $j = 0$ then Algorithm 2.3 returns $L' = \min(L_0, L_i)$ and $U' = \max(U_0, U_i)$. Since $[L_0, U_0]$ is an actual time difference for the 0th occurrence, we only need to show that $[L_i, U_i]$ always yields an estimate for $i > 0$. A maximum time difference is calculated recursively in terms of other maximum time differences (see Algorithm 2.1). Therefore, when calculating U_i using subgraph G, one of two cases may occur. Its value may be independent of *maxdiff* values for events not in graph G (i.e., events with indices less than $i - 1$). If this is the case, then $U_i = \min_{i \geq 1}\{U_i\}$. On the other hand, if it depends on time differences of earlier events not in graph G, then just before *MaxDiff* falls off the end of the graph, it will call either $\mathit{MaxDiff}(G, \langle source, i-1\rangle, \langle f, i-1\rangle)(1)$ or $\mathit{MaxDiff}(G, \langle f, i-1\rangle, \langle source, i-1\rangle)$ (2). Since the rule between $\langle f, i-1\rangle$ and $\langle source, i-1\rangle$ has timing constraint $[0, \infty]$, (1) will return ∞, and (2) will return 0. If graph G were extended to include another cycle, the rule between $\langle source, i-1\rangle$ and $\langle f, i-1,\rangle$ would be replaced with a rule of the form $\langle e, f, i-1, 1, \tau\rangle$. Now, $\mathit{MaxDiff}(G, \langle e, i-2\rangle, \langle f, i-1\rangle)$ would be called which would return a value less than or equal to ∞, or $\mathit{MaxDiff}(G, \langle f, i-1\rangle, \langle e, i-2\rangle)$ would be called which would return a value less than or equal to 0 (note this second case is never positive because from the ordering defined by the rule, we know that e always occurs before f). This relationship continues to hold if the graph is extended an infinite number of cycles. Since the value found for case (1) and for case (2) is greater than that found if graph G is extended back further, and since the maximum time difference is calculated by adding these values to values found on the rest of the graph, we know that the value calculated for U_i using graph G will be less than or equal to the actual value of U_i for $i > 1$. Therefore, $U' \geq U$, and we can similarly show that $L' \leq L$. Thus, Algorithm 2.3 gives an estimate of the worst-case time difference. □

4) Complexity of the Algorithm: Calculating the time difference of each pair of events using the *MaxDiff* algorithm has complexity $O(v \cdot e)$ where v is the number of vertices and e is the number of arcs in the graph [15]. Let $|E'|$ and $|R'|$ be the number of events and rules, respectively, in the cyclic constraint graph representation. The largest graph which Algorithm 2.3 analyzes has $2|E'|$ vertices and $2|R'|$ arcs. Therefore, using Algorithm 2.3 to calculate estimates for all time differences has complexity $O(|E'| \cdot |R'|)$.

5) Extensions to Find a Better Estimate: If either the bound is not tight enough or there is interest in finding worst-case time differences of events across more than one cycle (i.e., $j > 1$), the algorithm can be extended by increasing the size of the subgraphs which Algorithm 2.3 analyzes. Assuming subgraph G is enlarged to contain c cycles ($c = 2$ in Algorithm 2.3), the algorithm is modified in the following ways:

1. Construct subgraph G using only events and rules with indexes $i - (c-1), \ldots, i$ where $i > (c-2)$.
2. Construct subgraph G' using only events and rules with indices $i \leq (c-2)$.
3. If $j \leq (c-2)$ then using graph G', find $[L_j, U_j], \ldots, [L_{(c-2)}, U_{(c-2)}]$.
4. $L' = \min(L_i, L_j, \ldots, L_{(c-2)})$ and $U' = \max(U_i, U_j, \ldots, U_{(c-2)})$.

In the modified algorithm, estimates of worst-case time differences with $j \leq (c-1)$ can now be found. Theorem 2.1 can easily be extended to show that the modified algorithm returns an estimate of the worst-case time difference. It is also easy to show that the complexity of the modified algorithm is $O(c|E'| \cdot c|R'|)$.

6) Termination of the Algorithm: In order to avoid unnecessary calculations, the algorithm can be modified to check if extending the size of the subgraphs analyzed (i.e., increasing c) is helpful. To do this, the algorithm is modified to return a best-case estimate, $[L_{best}, U_{best}]$, in addition to the worst-case estimate, $[L', U']$, where $L_{best} = \min(L_j, \ldots, L_{(c-2)})$ and $U_{best} = \max(U_j, \ldots, U_{(c-2)})$. Given the actual worst-case time difference is $[L, U]$, it is easily shown that these estimates satisfy the inequalities: $L' \leq L \leq L_{best}$ and $U_{best} \leq U \leq U'$. If tightening the bound to $[L_{best}, U_{best}]$ would not result in less circuitry than $[L', U']$, then it is not worth increasing c. In fact, if $L_{best} = L'$ and $U_{best} = U'$, then the actual worst-case time difference $[L, U]$ has been found. In general, increasing c does not guarantee that the exact bound $[L, U]$ can always be found, but in all the circuit examples that we synthesized, Algorithm 2.3 (i.e., $c = 2$) either found the exact bound or at least a sufficiently tight bound to detect all redundancies.

III. Synthesis Procedure

Given an ER system specification, we apply our timing analysis algorithm to derive an optimized timed circuit implementation. The synthesis procedure has three steps. The first step is to detect and remove *redundant rules* from the specification. The second step is to construct a *reduced state graph*. The third step is to derive a circuit implementation from the reduced state graph.

3.1. Removing Redundant Rules

The first step in the synthesis procedure is to detect and remove redundant rules in the timed specification. Since each internal rule results in a literal in the implementation in order to ensure the behavior specified by the rule, if it is determined that this behavior is guaranteed without the rule (i.e., the rule is redundant) then the literal can be removed from the implementation resulting in a smaller circuit.

1) Redundant Rules: A rule is redundant in the timed specification if its omission does not change the behavior specified by the timed firing rule. This is defined more formally as follows:

Definition 3.1 (Redundant Rule): A rule $\langle e, f, i, \varepsilon, \tau\rangle$ is redundant for all $i \geq \varepsilon$ if the bound on the time of the event occurrence $\langle f, i\rangle$ with the rule removed as defined below:

$$\max_{\langle e,f,i,\varepsilon,\tau\rangle \in R_{NR}} \{t(\langle e, i-\varepsilon\rangle) + l\} \leq t(\langle f, i\rangle) \leq \max_{\langle e,f,i,\varepsilon,\tau\rangle \in R_{NR}} \{t(\langle e, i-\varepsilon\rangle) + u\}$$

where $R_{NR} = R - \{\langle e, f, i, \varepsilon, \tau\rangle | i \geq \varepsilon\}$ is the same as the bound specified in the timed firing rule (see Definition 2.3).

2) Algorithm for Detecting Redundant Rules: If there are multiple rules enabling an event, then it is possible that some of them are redundant. Algorithm 3.1 checks each rule by using Algorithm 2.3 to find an estimate of the worst-case time difference between the enabled and enabling event. We prove later that if the lower-bound of this estimate is larger than the upper-bound of the timing constraint on the rule, then the rule is redundant.

Algorithm 3.1 (Find Redundant Rules)

```
set FindRed(ER system ⟨E, R⟩) {
  R_NR = R;
  For each rule of the form ⟨e,f,i, ε,τ⟩ where τ = [l, u] {
    [L', U']=WCTimeDiff(⟨E, R⟩, e, f, ε);
    If (L' > u) then R_NR = R_NR − {⟨e, f, i, ε, τ⟩|i ≥ ε};
  }
  Return(R_NR);
}
```

The SCSI protocol controller example depicted in Fig. 2 has four events that are enabled by multiple rules: $req\downarrow$, $rdy\downarrow$, $req\uparrow$, and $q\uparrow$. For the rule, $\langle q\downarrow, rdy\downarrow, i, 0, [0,5]\rangle$, Algorithm 2.3 estimates the worst-case time difference between the two events $rdy\downarrow$ and $q\downarrow$ to be the bound $[15, 55]$. Since the lower-bound of this time difference, 15, is greater than the upper-bound of the timing constraint on the rule, 5, the rule is found to be redundant. In other words, the rule between the events $q\downarrow$ and $rdy\downarrow$ can be removed without changing the specified behavior. Further analysis finds this to be the only redundant rule.

3) Proof of Correctness: Definition 3.1 defined a redundant rule as a rule which could be removed from the ER system without changing the behavior specified by the timed firing rule. By applying transformations to the timed firing rule, Theorem 3.1 proves that Algorithm 3.1 finds redundant rules.

Theorem 3.1 Algorithm 3.1 finds redundant rules.

Proof: (by contradiction) Given a rule $\langle e, f, i, \varepsilon, \tau\rangle$ satisfies the condition set forth in Algorithm 3.1 to be redundant (i.e., $L' > u$), assume that it is not redundant. In that case, there exists a value of i such that one of the following is true:

$$t(\langle e, i-\varepsilon\rangle) + l \leq t(\langle f, i\rangle) \text{ or } t(\langle f, i\rangle) \leq t(\langle e, i-\varepsilon\rangle) + u.$$

(from Definitions 2.3 and 3.1). Now, subtract $t(\langle e, i-\varepsilon\rangle)$ from each element:

$$l \leq t(\langle f, i\rangle) - t(\langle e, i-\varepsilon\rangle) \text{ or } t(\langle f, i\rangle) - t(\langle e, i-\varepsilon\rangle) \leq u.$$

These are instances of a worst-case time difference, so they are bounded by $[L, U]$,

$$L \leq l \leq t(\langle f, i\rangle) - t(\langle e, i-\varepsilon\rangle) \leq U \quad \text{or } L \leq t(\langle f, i\rangle) - t(\langle e, i-\varepsilon\rangle) \leq u \leq U.$$

(from Definition 2.5). Since L' returned by Algorithm 2.3 is an estimate of the worst-case time difference (from Theorem 2.1), $L' \leq L$ (from Definition 2.6). Also, $L' > u$ (from Algorithm 3.1) and $u \geq l$ (from Definition 2.2), so the following inequalities hold:

$$l \leq u < L' \leq L \leq l \text{ or } u < L' \leq L \leq u.$$

Thus, we have a contradiction in each case. □

3.2. Finding the Reduced State Graph

In order to generate a circuit implementation, many methodologies transform a higher level specification into a *state graph* so that Boolean minimization techniques can be applied [2], [3]. Essentially, a state graph is a graph in which the vertices are bitvectors and the arcs are signal transitions. Each bitvector specifies the binary value of every signal in the system when the system is in that state. In our method, timing analysis is utilized to generate a *reduced state graph* which often has significantly fewer states than a state graph generated without considering timing constraints. Since the size of the state graph and the complexity of the circuitry are strongly correlated, our method often results in simpler circuitry compared with other methods that do not fully utilize timing constraints.

1) Reduced State Graph Typically, a state graph is specified as a set of states and a set of transitions between states [2], [3]. Algorithm 2.3 can be utilized to detect states that can never be reached, resulting in a reduced state graph. These unreachable states are removed from the set of states, and the transitions leading to them are removed from the set of

transitions. It is not always possible to infer from the reduced state graph all enabled transitions, since a transition can be enabled in a particular state without an arc leading from it to a state where that transition has occurred. Although the transition cannot occur in the next state, the fact that it has been enabled is needed during synthesis. To solve this problem, a reduced state graph is fully characterized by a set of states that contain information on enabled transitions as described in Definition 3.2. Each such state is a vertex in the reduced state graph, and these vertices are connected by arcs as described in Definition 3.3.

Definition 3.2 (State): Each state S is of the form $S = s_1, \cdots, s_k, \cdots, s_n$, where n is the number of signals in the specification. Each state bit s_k has the value: 0 if the signal s_k is low, R if the signal s_k is low but enabled to rise, 1 if the signal s_k is high, and F if the signal s_k is high but enabled to fall. The function $\mathrm{VAL}[s_k] = 0$ if $s_k = 0$ or R and $\mathrm{VAL}[s_k] = 1$ if $s_k = 1$ or F.

Definition 3.3: (Reduced State Graph): A reduced state graph is a graph in which its vertices are states and the arcs are allowed transitions between states. There exists an arc from state S to state S' if there exists a signal s_k, such that for all $l \neq k$, $\mathrm{VAL}[s'_l] = \mathrm{VAL}[s_l]$, and either $s_k = R$ and $\mathrm{VAL}[s'_k] = 1$, or $s_k = F$ and $\mathrm{VAL}[s'_k] = 0$.

2) Constrained Token Flow: The reduced state graph is derived using *constrained token flow* described in Algorithm 3.3. This is similar to *token flow* which is used for finding state graphs as described in [3] [2]. The algorithm begins with the initial marking of the constraint graph which is defined as the set of rules enabled by *reset*. The function *FindState* is then used to find the state as defined in Definition 3.2 for the marking. Given a marking, an event is enabled if all rules which enable that event are in the marking. If in a marking more than one event is enabled, all possible event sequences need to be generated. With timing constraints, it may be possible that one of the enabled events is always preceded by another, in which case the function *Slow*, implemented in Algorithm 3.2, is used to check if an enabled event is slower than some other enabled event. If so, the occurrence of the slower event is postponed. The result is that some states are no longer reachable, yielding a reduced state graph. Note that if the function *Slow* is changed to always return *FALSE* then the resulting state graph is the same as generated using regular token flow.

Algorithm 3.2: (Check If Event Is Slow)

```
boolean Slow(ER system ⟨E, R⟩; event occurrence ⟨u, k⟩;
    marking M) {
  For each event ⟨v, l⟩ that is enabled in M where u ≠ v,
    If (l ≥ k) then {
      [L', U']=WCTimeDiff(⟨E, R⟩, u, v, l − k);
      If (U' < 0) then Return(TRUE)
        else Return(FALSE);
    } Else {
      [L', U']=WCTimeDiff(⟨E, R⟩, v, u, k − l);
      If (L' > 0) then Return(TRUE)
        else Return(FALSE);
}}
```

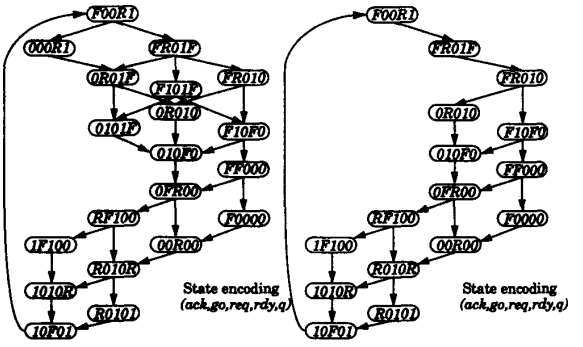


Fig. 3. (a) State graph for the SCSI protocol controller. (b) Reduced state graph for the SCSI protocol controller.

Algorithm 3.3: (Find Reduced State Graph)

```
set FindRSG(ER system ⟨E, R⟩) {
  initial_marking = {rules in R of the form
    ⟨reset, f, 0, 0, τ0⟩};
  set_of_markings = {initial_marking};
  present_state = FindState(⟨E, R⟩, initial_marking);
  set_of_states = {present_state};
  While (set_of_markings ≠ ∅) {
    Take marking from set_of_markings (i.e.,
      set_of_markings = set_of_markings − {marking} );
    For each enabled event ⟨f, i⟩ in marking {
      If not (Slow(⟨E, R⟩, ⟨f, i⟩, marking)) then {
        new_marking = marking − { rules in marking
          of the form ⟨e, f, i, ε, τ⟩}
          + {rules in R of the form ⟨f, g, i + ε, ε', τ'⟩};
        present_state = FindState(⟨E, R⟩, new_marking);
        If (present_state ∉ set_of_states) then {
          set_of_states = set_of_states
            +{ present_state };
          set_of_markings = set_of_markings
            +{ new_marking };
  }}}}
  Return(set_of_states);
}
```

Using this algorithm on the SCSI protocol controller with the function *Slow* replaced with *FALSE* (i.e., ignoring the timing constraints), the state graph obtained contains 20 states as shown in Fig. 3(a). If the timing constraints are considered, a reduced state graph is derived which contains 16 states as shown in Fig. 3(b).

3.3. Derivation of a Circuit Implementation

Several methods exist which transform a state graph into a circuit implementation such as those described in [2]–[4]. We present a method similar to *guard strengthening* described in [21] but derive the circuit implementation from a state graph. A *guard* is a conjunction of signals and their negations. When this conjunction evaluates to true, the transition it is guarding can occur. The reason this method is called *guard*

strengthening is that it starts with *weak guards* (i.e., the guard may evaluate to true in states in which the transition it is guarding should not occur) to which signals are added to strengthen them.

1) Finding the Enabled State: The first step is to determine the *enabled state* for the transitions on each signal. The enabled state for a transition is the value of each signal in all states in which that transition is enabled to occur. This provides information on which signals are stable during a particular transition, and thus, can be used to strengthen the guard for that transition. This is defined more formally in Definition 3.4. Algorithm 3.4 shows how the enabled state for each transition can be found from the reduced state graph.

Definition 3.4: (Enabled State): For each transition $s_k \uparrow$, the enabled state is of the form $Q_{k\uparrow} = q_{k\uparrow,1}, \ldots, q_{k\uparrow,l}, \ldots, q_{k\uparrow,n}$, where n is the number of signals in the specification. Each $q_{k\uparrow,l}$ is determined as follows: if in all states where $s_k = R$, VAL$[s_l] = 0$ then $q_{k\uparrow,l} = 0$; if in all states where $s_k = R$, VAL$[s_l] = 1$ then $q_{k\uparrow,l} = 1$; otherwise, $q_{k\uparrow,l} = X$. The enabled state for the transition $s_k \downarrow$ is similarly defined.

Algorithm 3.4: (Find Enabled State)

array FindES(ER system $\langle E, R\rangle$; set set_of_states) {
  For each signal s_k, $Q_{k\uparrow} = Q_{k\downarrow}$ = undefined, ..., undefined;
  For each state and each signal s_k in each state
    If ($s_k == R$) then
      For each signal s_l
        If ($q_{k\uparrow,l}$ == undefined) then $q_{k\uparrow,l}$ = VAL$[s_l]$;
        Else if ($q_{k\uparrow,l} \neq$ VAL$[s_l]$) then $q_{k\uparrow,l} = X$;
    Else if ($s_k == F$) then
      For each signal s_l
        If ($q_{k\downarrow,l}$ == undefined) then $q_{k\downarrow,l}$ = VAL$[s_l]$;
        Else if ($q_{k\downarrow,l} \neq$ VAL$[s_l]$) then $q_{k\downarrow,l} = X$;
    Return(Q);
}

In the SCSI protocol controller, the enabled state for the transition $req \uparrow$ is $0X000$, since there are two states $0FR00$ and $00R00$ where the transition $req \uparrow$ is enabled to occur. In this case, both the state graph and the reduced state graph give the same enabled state. However, for the transition $rdy \uparrow$, if the state graph is used, the enabled state is $X0001$, but if the reduced state graph is used, the enabled state is 10001. Therefore, using timing constraints, the enabled state can contain less uncertainty.

2) Detecting and Resolving Conflicts: The next step is to check for *conflicts* in each state. A conflict occurs when the weak guard evaluates to true in a particular state, but in that state the signal is enabled to change or has changed to the opposite value. This either results in *interference*, where a signal is being both pulled high and low at the same time, or it can result in a *misfiring*, where a transition occurs in a state in which the signal should remain stable. Both cases represent circuit hazards and must be prevented.

The non-redundant rules are used to construct the weak guards for each transition. To prevent a conflict, *context signals* are added to a weak guard to guarantee that the transition being guarded cannot occur in the particular problem state. A signal can be used as a context signal if it is stable in the enabled state for the transition, and its value in the enabled state is different from that in the problem state. For each transition, a table is constructed where the columns are the conflict problem states, and the rows are the signals which can be chosen to solve each problem. An outline of the basic procedure is described in Algorithm 3.5. The function *Problem* determines if a set of rules is sufficient to prevent a given transition from occurring in a particular state. The function *Solution* checks if a signal or its negation can be used to prevent a transition from occurring in a given state.

Algorithm 3.5: (Find Conflicts):

array FindConf(ER system $\langle E, R_{NR}\rangle$; set set_of_states; array Q) {
  For each state S and each signal s_k in S
    If (($s_k == F$ or $s_k == 0$) and (Problem($S, s_k \uparrow$, {rules in R_{NR} of the form $\langle e, s_k \uparrow, i, \varepsilon, \tau\rangle$}))) then
      For each signal s_l,
        If (Solution($S, q_{k\uparrow,l}$)) then $C_{k\uparrow}[s_l, S]$ = TRUE;
    Else If (Problem($S, s_k \downarrow$, {rules in R_{NR} of the form $\langle e, s_k \downarrow, i, \varepsilon, \tau\rangle$})) then
      For each signal s_l,
        If (Solution($S, q_{k\downarrow,l}$)) then $C_{k\downarrow}[s_l, S]$ = TRUE;
    Return(C);
}

In the SCSI protocol controller, the transition $req \uparrow$ has the weak guard $\neg ack \wedge \neg rdy$. Using this guard and the state graph shown in Fig. 3(a), there is a conflict with the state $000R1$. This problem state is compared with the enabled state for $req \uparrow$, $0X000$, as determined earlier. The only signal which can be chosen to solve this problem is $\neg q$. Using the reduced state graph in Fig. 3(b), the state $000R1$ is not reachable, so there is no conflict. Thus, the guard is not strengthened with $\neg q$, and the timing constraints have again helped reduce the complexity of the implementation.

3) Finding an Optimal Cover: Determining which context signals to use to optimally solve all conflicts constitutes a covering problem, which is solved by treating the table of conflict problems and possible solutions as a prime implicant table [22]. Thus for each transition, a prime implicant table is solved using the procedure outlined in Algorithm 3.6. The function *Choose_essential_rows* determines if a problem has only one possible solution. If so, the signal associated with that solution is added to the guard and all problems solved by this signal are removed from the table. The function *Rm_dominating_columns* detects if solving a problem implies another will be solved, and if so, removes the second problem. The function *Rm_dominated_rows* checks if one solution solves all the same problems that another solution does and more, and if so, removes the second solution. If there is only one remaining problem to solve, the function *Solve_essential_columns* solves it, and, if possible, does so by selecting a signal which provides symmetry between guards for the rising and falling transition.

This procedure is repeated until all problems are solved, or the number of problems solved is no longer decreasing. At the end of the procedure, all problems may not be resolved if the table is *cyclic*, in which case the remaining problems can be solved by inspection or a *branching method* [22] implemented in the function *Solve_remaining_problems*.

TABLE I
CONTEXT SIGNAL TABLE FOR THE TRANSITION $rdy \uparrow$ FROM THE SCSI PROTOCOL CONTROLLER

Context Signal Table for $rdy \uparrow$						
Problems / Solutions	P10P0	010P0	PP000	0PR00	P0000	00R00
ack		√		√		√
$\neg go$	√	√	√	√		
$\neg rdy$	√	√				
q	√	√	√	√	√	√

Algorithm 3.6: (Find Optimal Cover):
set FindCover(ER system $\langle E, R_{NR}\rangle$*; array C)* {
$R_C = \emptyset$;
For each transition t {
While $((NumProb(C_t) > 0)$ *and* $(NumProb(C_t)$ *is decreasing)) do* {
$R_C = R_C + Choose_essential_rows(C_t)$;
$Rm_dominating_columns(C_t)$;
$Rm_dominated_rows(C_t)$;
$R_C = R_C +$
$Solve_essential_columns(\langle E, R_{NR} + R_C\rangle, C_t)$;
}
If $(NumProb(C_t) > 0)$ *then* $R_C = R_C$
$+ Solve_remaining_problems(\langle E, R_{NR} + R_C\rangle, C_t)$;
}
$Return(R_C)$;
}

Returning to our example, in the reduced state graph, there are still conflicts associated with the transition $rdy \uparrow$. A table of problems and possible solutions is shown in Table I. In this table, there is an essential row since the fifth column can only be solved by choosing q. Strengthening the guard with this signal solves all the problems.

4) A Complex Gate Implementation: For each output signal s the *trigger signals* (i.e., those given in the rules) and the context signals (i.e., those added to solve conflicts) for $s \uparrow$ are implemented in series in a pullup network, and similarly, the signals needed for $s \downarrow$ are implemented in series in a pulldown network. The resulting circuit is a state-holding element called a *generalized C-element* [1]. The complex gate implementations for both the speed-independent and the timed versions of the signal req from the SCSI protocol controller are shown in Fig. 4, with the guards that are being implemented. If a signal appears only in the pullup, but not in the pulldown, then it is annotated with a "+". If a signal appears only in the pulldown then it is annotated with a "−". Otherwise, the signal has no annotation. A static CMOS implementation for each element is also shown in Fig. 4. Since signal $\neg q$ was not needed in the guard for $req \uparrow$ for the timed implementation, the resulting circuitry needs two less transistors. Similarly, since the rule $\langle q \downarrow, rdy \downarrow, i, 0, [0,5]\rangle$ is found to be redundant, the signal $\neg q$ is not used in the guard for the transition $rdy \downarrow$, and two transistors are saved there as well.

3.4. Exceptions

Throughout the synthesis procedure, there are various exception conditions which can occur if the procedure finds that it has a specification for which it cannot derive an implementation. Each is briefly described here with suggestions on how

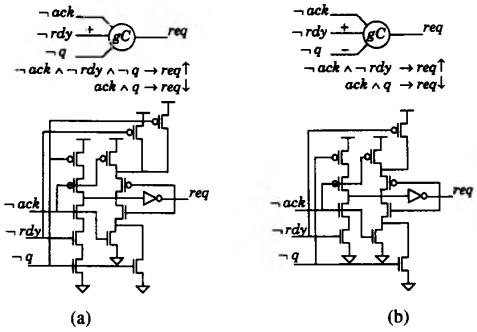


Fig. 4. (a)Speed independent implementation of req. (b) Timed implementation of req.

to modify the specification to solve the problem, but a general solution for timed circuits is still an open area of research.

1) Complete State Coding Violation: A timed specification violates the *complete state coding property* if in the reduced state graph, two states have the same binary value, but different transitions on non-input signals are enabled in each state (see Definition 3.5). To solve this problem, *state variables* are usually added to the specification.

Definition 3.5: (Complete State Coding Property): A reduced state graph has the complete state coding property if for any two states S and S' either there exists a signal s_k such that $\mathrm{VAL}[s_k] \neq \mathrm{VAL}[s'_k]$, or for all non-input signals s_k, $s_k = s'_k$.

2) Persistency Violation: After the enabled state is found, the synthesis procedure verifies that the timed specification is *persistent* [2], [3] as defined below. While in general the persistence property is not a necessary requirement for synthesis [4], it is required to use the enabled state approach. Persistence problems can be solved by either adding state variables or *persistence rules* [2], [3].

Definition 3.6:(Persistence) For each rule of the form $\langle e, f, i, \varepsilon, \tau\rangle$ in the set of non-redundant rules R_{NR}, if event e is a rising transition on the signal s_k and the enabled state Q_f of event f has $q_{f,k} = 1$, then event e is persistent. If event e is a falling transition on the signal s_k and the enabled state has $q_{f,k} = 0$, then event e is persistent.

3) Unresolvable Conflicts: Finally, it is possible that there may be no available context signal to resolve a conflict. This problem may be caused by a potential context signal which is non-persistent [4]. To solve this problem, state variables are again added.

3.5. Putting It All Together

The entire synthesis procedure neglecting exceptions can be given as follows:

Algorithm 3.7 (Automated Timed Asynchronous Circuit Synthesis)

circuit ATACS(ER system $\langle E, R\rangle$*)* {
$R_{NR} = FindRed(\langle E, R\rangle)$;
$RSG = FindRSG(\langle E, R\rangle)$;
$Q = FindES(\langle E, R\rangle, RSG)$;

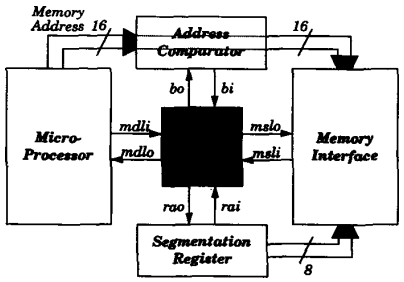


Fig. 5. Block diagram for part of the MMU controller.

$C = FindConf(\langle E, R_{NR}\rangle, RSG, Q);$
$R_C = FindCover(\langle E, R_{NR}\rangle, C);$
$Circuit = FindCircuit(\langle E, R_{NR} + R_C\rangle);$
$Return(Circuit);$
}

IV. Examples

This section describes two practical examples: a memory management unit (MMU) and a DRAM controller. The MMU is derived from a CSP specification, and it is used to illustrate the complexity reduction of timed circuits compared to speed-independent circuits. The DRAM controller is derived from a BSM specification, and it is used to demonstrate how timed circuits can be used in a synchronous environment.

4.1. Memory Management Unit

The first example is a MMU designed for use with a 16-bit asynchronous microprocessor [11]. The original implementation was derived using Martin's synthesis method [12]. The basic operation of the MMU is to convert a 16-bit *memory address* to a 24-bit *real address*. There are six possible cycles that the MMU controller can enter, depending on data from the environment. For simplicity, the design of only one cycle is discussed: *memory data load*. A simplified block diagram is shown in Fig. 5 in which only signals involved in this cycle are depicted.

1) From CSP to a Timed Specification: The high-level CSP specification for the memory data load cycle is: $*[\overline{MDl} \rightarrow (RA \;\|\; B); MSl; MDl]$ (see [12]). This specification is initially transformed into the following *handshaking expansion*:

$$*[[mdli \wedge \neg rai]; rao\uparrow; [\neg bi]; bo\uparrow; [rai \wedge bi \wedge \neg msli]; mslo\uparrow; [msli]; mdlo\uparrow; rao\downarrow; bo\downarrow; [\neg mdli]; mslo\downarrow; mdlo\downarrow],$$

which is then converted to the constraint graph shown in Fig. 6.

The transformation from CSP to a handshaking expansion is not unique. A more concurrent constraint graph shown in Fig. 7 also satisfies the high-level CSP specification. This specification is simply a *reshuffling* [1] of the earlier one. This reshuffling is not considered in [12] because it results in a complete state coding violation [2]. This means that the more concurrent specification cannot be implemented without adding state variables. Adding state variables not only changes the specification, but can also add extra circuitry and/or delay

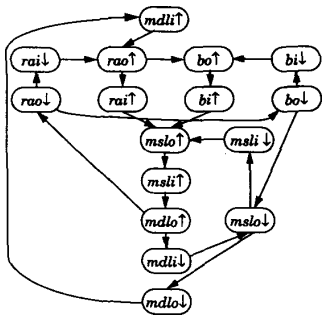


Fig. 6. The cyclic constraint graph specificationn for the unoptimized MMU.

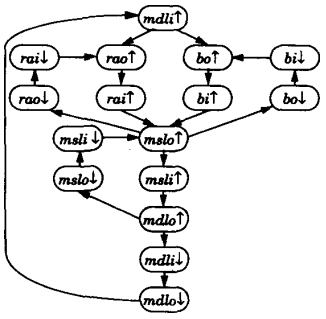


Fig. 7. The cyclic constraint graph specification for the optimized MMU.

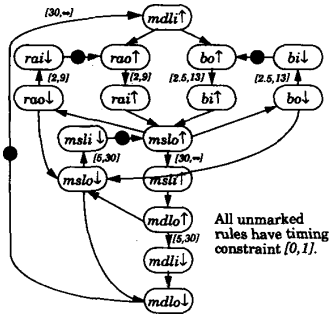


Fig. 8. The cyclic constraint graph specification for the persistent MMU.

to the implementation. This cost often outweights the benefit of the higher degree of concurrency. This particular problem can also be solved by adding persistence rules, but this can reduce the concurrency in the specification. If conservative timing constraints are also added, the reduced state graph of the more concurrent specification shown in Fig. 7 does not have a complete state coding violation, and thus, it can be implemented without adding state variables or persistence rules. To make the specification in Fig. 7 persistent, three arcs are added to the constraint graph as shown in Fig. 8; the specification can now be implemented speed-independently. As shown later, the speed-independent implementation is still more complex than the original implementation derived from the specification in Fig. 6.

2) Speed-Independent versus Timed Implementation: A speed-independent and a timed implementation of the specifi-

TABLE II
COMPARISON OF THE GUARDS FOR THE SPEED-INEDEPENDENT AND TIMED IMPLEMENTATIONS OF THE MMU

Speed-Independent Guards			Simplified Timed Guards		
$mslo \wedge msli$	$\mapsto$	$mdlo \uparrow$	$mdli \wedge msli$	$\mapsto$	$mdlo \uparrow$
$\neg mslo \wedge \neg mdli$	$\mapsto$	$mdlo \downarrow$	$\neg mdli$	$\mapsto$	$mdlo \downarrow$
$\neg mdlo \wedge \neg mslo \wedge \neg rai \wedge mdli$	$\mapsto$	$rao \uparrow$	$\neg mdlo \wedge \neg mslo \wedge mdli$	$\mapsto$	$rao \uparrow, bo \uparrow$
$mslo$	$\mapsto$	$rao \downarrow$	$mslo$	$\mapsto$	$rao \downarrow, bo \downarrow$
$\neg mdlo \wedge \neg mslo \wedge \neg bi \wedge mdli$	$\mapsto$	$bo \uparrow$			
$mslo$	$\mapsto$	$bo \downarrow$			
$rao \wedge bo \wedge \neg msli \wedge rai \wedge bi$	$\mapsto$	$mslo \uparrow$	$rai \wedge bi$	$\mapsto$	$mslo \uparrow$
$\neg rao \wedge \neg bo \wedge mdlo$	$\mapsto$	$mslo \downarrow$	$mdlo$	$\mapsto$	$mslo \downarrow$

cation shown in Fig. 8 are compared. For the timed implementation, the timing constraints used are depicted in Fig. 8. The lower-bound of the timing constraint on $mdli \uparrow$ states that the processor does not issue memory requests faster than every 30 ns. The lower bound of the timing constraint on $msli \uparrow$ states that the DRAM access time takes at least 30 ns. Both of their upper-bounds are infinite since the processor could choose never to do a load, or the interface could choose never to process the request. The reseting of the acknowledgement (i.e., $mdli \downarrow$ and $msli \downarrow$) is assumed to be somewhat faster, and must occur within 5 to 30 ns of the reset of the request. The other numbers were obtained from SPICE simulations of the datapath circuitry for a 0.8μm CMOS process. The comparator, denoted bi, has a delay of between 2.5 to 13 ns, and the registers, denoted rai, have a delay of between 2 to 9 ns depending on temperature, voltage, and processing variations. All output signals have a delay of 0 to 1 ns where 1 ns was found to be the maximum delay of the gates in the library used.

In the MMU specification, there are five events with multiple rules enabling them: $rao \uparrow$, $bo \uparrow$, $mslo \uparrow$, $mslo \downarrow$, and $mdlo \downarrow$. Timing analysis determines that at least one rule associated with each event is redundant. In all, 6 of the 15 rules on output signals in the original specification are redundant. This includes the 3 persistence rules. To determine which context signals must be added, the first step is to determine the reduced state graph and the enabled state for each signal using the timing constraints. A state graph generated without any timing constraints results in 92 states while the reduced state graph only has 22 states. Using the reduced state graph, the timed implementation needs 5 context signals as opposed to 7 needed for the speed-independent implementation.

After adding context signals to our original specification, 22 literals (note that we define a literal to be a signal in a guard) are required for a speed-independent implementation as shown in Table II. The timing constraints reduce the circuit to only 10 literals. Thus, our circuit complexity is reduced by over 50 percent using conservative timing constraints. A complex gate implementation for both is shown in Fig. 9. Note that this reduction is possible not only because of removing redundant literals, but also because the gate needed for implementing rao and bo can be shared after the optimizations.

4.2. DRAM Controller

Our next example is a DRAM controller which is an interface between a microprocessor and a DRAM array. This example is interesting for two reasons. It is an asynchronous design in a synchronous environment, and it is an example which includes non-deterministic behavior (i.e., input choice)

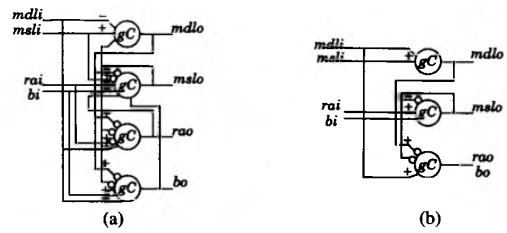


Fig. 9. (a) Speed-independent implementation of the MMU controller. (b) Timed implementation of the MMU controller.

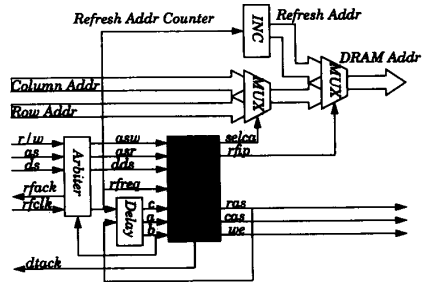


Fig. 10. Block diagram of the DRAM controller (courtesy of [13]).

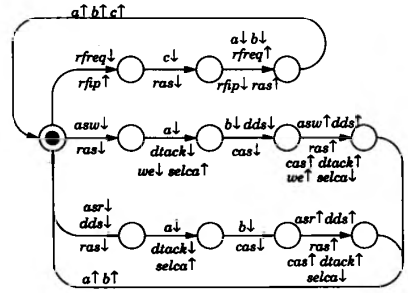


Fig. 11. Burst-mode specification for the DRAM controller (courtesy of [13].

which can be synthesized by transforming it into a deterministic specification. The DRAM controller has three possible modes of operation: *read, write*, and *refresh*. A block diagram for the entire DRAM controller is shown in Fig. 10. The design of the refresh cycle is discussed in detail in the next subsection to illustrate how synchronous inputs can be incorporated into an asynchronous design. The three cycles are combined to illustrate synthesis of a specification with non-determinism and multiple occurrences of events in a single cycle.

1) From Burst Mode to Timed Specification: Our specification is derived from a burst-mode specification shown in Fig. 11 [13]. The specification of the refresh cycle is converted to the constraint graph shown in Fig. 12. Notice that this constraint graph is not well-formed (i.e., it is not strongly connected), so our timing analysis procedure cannot be applied directly. To solve this problem, the dashed arcs in Fig. 13 are added to the constraint graph. For this example, these new *ordering rules* are chosen to make the specification satisfy the fundamental mode assumption (i.e., outputs must occur before inputs can change). For example, the transition $rfip \uparrow$

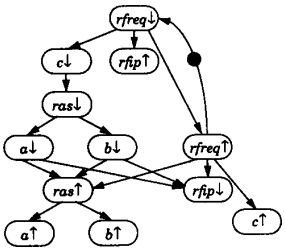


Fig. 12. Constraint graph specification for the *refresh* cycle of the DRAM controller.

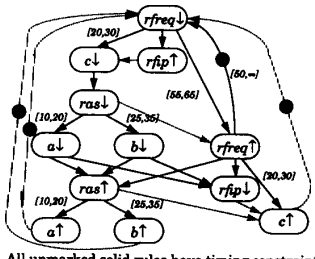


Fig. 13. Well-formed constraint graph specification for the *refresh* cycle of the DRAM controller. Unmarked solid rules have timing constraint $[0, 2]$. Dashed rules have timing constraint $[0, 0]$.

must occur before $c \downarrow$, so a rules is added between them. The timing constraints for these rules are $[0, 0]$ which means that only the ordering of the two events is important and not the time difference between them.

In general, an ordering rule can be added between two events if the enabling event is guaranteed by the timing constraints to *always* precede the enabled event by at least the amount of time given in the upper bound of the ordering rule. In other words, the rule must be declared redundant using the timing analysis, since it is not actually enforced with circuitry. If this is the case, the implementation synthesized is valid; otherwise different ordering rules need to be chosen. If no ordering rules can be found to make the graph well formed, then our procedure cannot derive an implementation which can satisfy the given timing constraints. Currently, these ordering rules must be added before the synthesis procedure can be applied, but future research will incorporate finding appropriate ordering rules into the procedure.

2) Burst Mode versus Timed Implementation: The implementation of a timed version of the DRAM controller is compared with implementations from two burst-mode design styles [13], [7]. For our timed implementation, the timing constraints used for the refresh cycle are depicted in Fig. 13 [23]. These timing constraints are derived assuming the environment is as depicted in Fig. 10, and the controller is being used with a 68020/30 running at 16–20 MHz.

The implementation of the refresh cycle is considered first. As before, the first step is to determine which rules in the specification are redundant. All 7 of the ordering rules added to make the graph well formed are found to be redundant. In addition, the rules from $a \downarrow$ to $ras \uparrow$ and $rfip \downarrow$ are also redundant. Next, the synthesis procedure derives a reduced state graph with 16 states. Using this reduced state graph

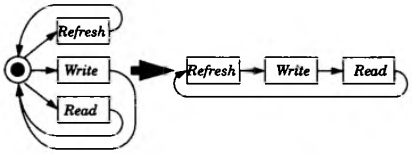


Fig. 14. Removal of data dependence from the DRAM controller specification.

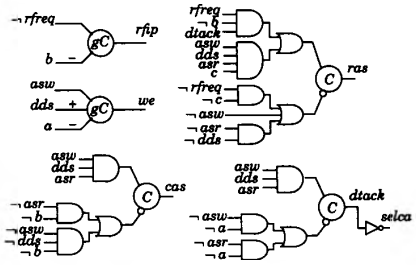


Fig. 15. Timed implementation of the DRAM controller.

and the non-redundant rules, one context signal is needed for the implementation. In all, 7 literals are needed for the implementation of the two output signals in the refresh cycle, $rfip$ and ras.

The implementation of the complete DRAM controller is non-deterministic; i.e., the environment can choose to do a refresh cycle, a write cycle, or a read cycle. Our timing analysis algorithm cannot analyze specifications with non-determinism directly. To solve this problem, the specification is converted to a long cycle going through a refresh, a write, and a read cycle sequentially as illustrated in Fig. 14. In this example, since each cycle always returns to the same state before the next cycle is chosen, all possible behaviors are modeled.

The resulting cyclic constraint graph has multiple occurrences of the same event in a cycle. For example, the transition $ras \uparrow$ now occurs three times in a single cycle. Each event which occurs multiple times is given a unique name for each occurrence, and these events are noted to be on the same signal. For example, the three occurrences of $ras \uparrow$ are replaced with $ras_1 \uparrow$, $ras_2 \uparrow$, and $ras_3 \uparrow$. These events will be treated separately during timing analysis, but together during synthesis.

The same procedure described earlier is used to find redundant rules and the reduced state graph. When determining the enabled state, the multiple occurrences of an event are considered together. For example, when determining the enabled state for the transition $we \downarrow$, there is a state where $dtack_1 = F$ and $dtack_2 = 1$, and another state where $dtack_1 = 0$ and $dtack_2 = 1$. Therefore, in the enabled state for $we \downarrow$, both $dtack_1$ and $dtack_2$ are set to X. To find conflicts, the individual occurrences of the same event are used, but to determine context signals, only the merged value is available. For example, ras can only be used as a context signal if ras_1, ras_2, and ras_3 all qualify as context signals.

This procedure leads to the implementation of the DRAM controller shown in Fig. 15. Note that although some of the gates are shown with multiple levels, they are all actually

TABLE III
COMPARISONS BETWEEN SPEED-INDEPENDENT AND TIMED IMPLEMENTATIONS OF SEVERAL EXAMPLES. "SV" UNDER CONTEXT RULES INDICATES THAT A STATE VARIABLE IS NEEDED FOR SYNTHESIS

Examples Optimizied Using Timing Constraints								
	Speed-Independent				Timed			
Example	Trigger Signals	States	Context Signals	Total Literals	Trigger Signals	States	Context Signals	Total Literals
SCSI Protocol Controller [20]	10	20	2	12	9	16	1	10
Pipeline Handshake [3], [18]	8	16	0	8	6	12	0	6
MMU unoptimized	13	56	3	16	9	33	3	10
MMU optimized	12	174	sv	n/a	7	22	3	10
MMU persistent	15	92	7	22	7	22	3	35
DRAM controller [13]	n/a	n/a	n/a	n/a	24	96	11	35
Microprocessor [11], [14] Fetch	8	32	sv	n/a	7	14	3	10
PCAdd	10	49	sv	n/a	7	18	3	10
Exec	42	354	sv	n/a	17	76	sv	n/a
ALU	26	123	sv	n/a	17	44	3	20

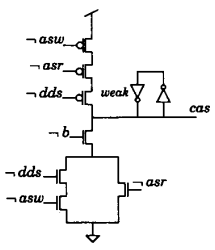


Fig. 16. Complex-gate implementation of the DRAM controller.

implemented as single complex gates. For example, a dynamic gate implementing *cas* is shown in Fig. 16. Our final implementation has 35 literals (41 literals if the gate for *dtack* and *selca* are not shared). A locally clocked implementation as reported in [13] used 62 literals and 1 state variable. A 3-D implementation as reported in [7] used 46 literals and 1 state variable. Our implementation did not need a state variable.

4.3. Other Results

The synthesis procedure described in this paper has been fully automated in a CAD tool which transforms a well-formed ER system specification into a complex gate implementation. All results reported in this paper were compiled using this program, and they appear tabulated in Table III.

Additional examples in this table are parts of an asynchronous microprocessor described in [11], and their specifications are taken from [14]. All of the microprocessor specifications need state variables for a speed-independent implementation; however, three of the four can be implemented without state variables if conservative timing constraints are added.

The timed implementation for the MMU controller and the *refresh* cycle of the DRAM controller have been verified using Burch's timed circuit verifier [24], [25] to be hazard free under the given timing constraints. Here, hazard-freedom is defined to mean that no transition once enabled to occur can be disabled without it occurring.

V. CONCLUSIONS AND FUTURE RESEARCH

We have proposed a new methodology for the specification of timed asynchronous circuits, the event-rule system, and developed a timing analysis algorithm to deduce timing information sufficient for the synthesis of timed circuits. A synthesis procedure based on our timing analysis algorithm has been constructed to detect and remove redundancy in the specification and to produce a reduced state graph. From the reduced state graph, our procedure systematically derives a complex gate implementation. Our results indicate that by using conservative timing constraints, our synthesis procedure can significantly reduce a circuit's complexity. While reducing circuit area, we also increase circuit performance, not only because smaller circuits switch faster but also because we are able to synthesize more concurrent specifications than can often be considered practical using other design styles. Finally, we have applied our technique to the synthesis of asynchronous circuits in a mixed synchronous/asynchronous environment.

At present, our synthesis procedure requires a well-formed, deterministic ER system specification. While we have shown through an example how these restrictions can be relaxed, a systematic method has not yet been incorporated into our synthesis procedure. In the future, we plan to incorporate transformations to make a specification well-formed into the synthesis procedure. We also plan to generalize our timing analysis algorithm to handle non-deterministic behavior. The third direction for future work is develop a procedure for adding state variables to a timed specification to resolve exceptions: complete state coding violations, persistency violations, and unresolvable conflicts. This problem is not as straightforward as it sounds because adding state variables changes the specification, and thus may invalidate earlier timing analysis. Therefore, techniques used for adding state variables in other

methodologies may not be directly applicable. Also, while we are able to verify our designs to be hazard free, verifying that they satisfy a specification has not yet been completed and will be addressed in the future. Finally, we intend to apply our technique to larger examples, and implement the IC design of interesting timed circuits to better assess the area and performance gain.

ACKNOWLEDGMENT

The authors would especially like to thank Prof. David Dill of Stanford University who dedicated considerable amount of time in assisting us in formalizing our work. We would also like to thank Peter Beerel of Stanford University for his invaluable comments on numerous versions of this manuscript. Their thanks also go to Dr. Jerry Burch of Stanford University for verifying several of our designs. The authors would also like to thank Profs. Steve Burns and Gaetano Borriello of the University of Washington and their students for many valuable discussions on timing analysis and the synthesis of timed circuits. Finally, they would like to express our appreciation of the work done in Professor Alain Martin's group at the California Institute of Technology, for their insight in the design of asynchronous circuits—their assistance in deriving the specification and speed-independent implementation of the MMU example is also gratefully acknowledged.

REFERENCES

[1] Alain J. Martin, "Programming in VLSI: From communicating processes to delay-insensitive VLSI circuits," in *UT Year of Programming Institute on Concurrent Programming*, C.A.R. Hoare, Ed. Reading, MA: Addison-Wesley, 1990.

[2] Tam-Anh Chu, "Synthesis of self-timed VLSI circuits from graph-theoretic specifications," Ph.D. dissertation, Massachusetts Inst. of Technology, Cambridge, 1987.

[3] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messershmitt, "Automatic synthesis of asynchronous circuits from high-level specifications," *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 1185–1205, Nov. 1989.

[4] Peter A. Beerel and Teresa H. Y. Meng, "Automatic gate-level sythesis of speed-independent circuits," in *Proc. IEEE 1992 ICCAD Dig. Papers*, pp. 581–586, 1992.

[5] S. H. Unger, *Asynchronous Sequential Switching Circuits*. New York: Wiley-Interscience, 1969. (re-issued by Robert E. Krieger, Malabar, 1983).

[6] Steven M. Nowick and David L. Dill, "Synthesis of asynchronous state machines using a local clock," presented at the IEEE Int. Conf. on Computer Design, ICCD-1991, 1991.

[7] K. Y. Yun, D. L. Dill, and S. M. Nowick, "Synthesis of 3D asynchronous state machines," presented at the IEEE Int. Conf. on Computer Design, ICCD-1992, 1992.

[8] Al Davis, Bill Coates, and Ken Stevens, "The Post Office Experience: Designing a large asynchronous chip," in *Proc. Twenty-Sixth Ann. Hawaii Int. Conf. on System Sciences*, pages 409–418. IEEE Computer Science Press, 1993.

[9] Gaetano Borriello and Randy H. Katz, "Synthesis and Optimization of Interface Transducer Logic," in *Proc. IEEE 1987 ICCAD Dig. Papers*, pp. 274–277, 1987.

[10] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli, "Algorithms for synthesis of hazard-free asynchronous circuits," in *Proc. 28th ACM/IEEE Design Automation Conf.*, 1991.

[11] A. J. Martin, S. M. Burns, T. K. Lee, D. Borković, and P. J. Hazewindus, "The design of an asynchronous microprocessor," in *Decennial Caltech Conf. on VLSI*, pp. 226–234, 1989.

[12] Chris J. Myers and Alain J. Martin, "The design of an asynchronous memory management unit," Tech. Rep. CS-TR-92-25, California Inst. of Technology, 1992.

[13] S. M. Nowick, K. Y. Yun, and D. L. Dill. "Practical asynchronous controller design," presented at the IEEE Int. Conf. on Computer Design, ICCD-1992, 1992.

[14] Steve Burns, " Performance analysis and optimization of asynchronous circuits," Ph.D. dissertation, California Inst. of Technology, Pasadena, 1991.

[15] Kenneth McMillan and David L. Dill, "Algorithms for interface timing verification," presented at the IEEE Int. Conference on Computer Design, ICCD-1992, 1992.

[16] P. Vanbekbergen, G. Goossens, and H. De Man, "Specification and analysis of timing constraints in signal transition graphs," in *Proc. European Design Automation Conf.*, 1992.

[17] T. Amon, H. Hulgaard, G. Borriello, and S. Burns, "Timing analysis of concurrent systems," Tech. Rep. UW-CS-TR-92-11-01, Univ. of Washington, Seattle, 1992.

[18] Chris Myers and Teresa H.-Y. Meng. "Synthesis of timed asynchronous circuits," presented at the IEEE Int. Conf. on Computer Design, ICCD-1992, 1992.

[19] Glynn Winskel, "An introduction to event structures," in *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. Norway: Noordwijkerhout, 1988.

[20] Tam-Anh Chu, Private communication, July 1991.

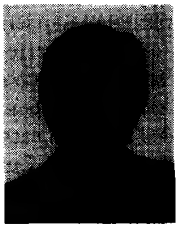
[21] Alain J. Martin. "Formal program transformations for vlsi circuit synthesis," in *UT Year of Programming Institute on Formal Developments of Programs and Proofs*, E. W. Dijkstra, Ed. Reading, MA: Addison-Wesley, 1989.

[22] Edward J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuits*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

[23] Ken Yun, Private communication, 1992.

[24] Jerry R. Burch. "Modeling timing assumptions with trace theory," in *1989 Int. Conf. on Computer Design: VLSI in Computers and Processors*, pp. 208–211, 1989.

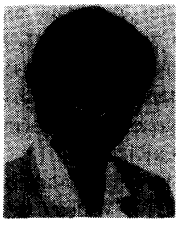
[25] Jerry R. Burch, " Trace algebra for automatic verification of real-time concurrent systems," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, 1992.



Chris J. Myers received the B.S. degree in electrical engineering and Chinese history in 1991 and the M.S.E.E. degree in 1993 from the California Institute of Technology, Pasadena. Currently he is working toward the Ph.D. degree at the same university.

His current research include asynchronous logic synthesis, algorithms for analysis of real-time concurrent systems, CAD for VLSI systems, computer architecture.

Mr. Myers has received the Caltech Carnation Merit Award in 1990, the Rodman W. Paul History Prize in 1991, and a National Science Foundation Fellowship in 1991.



Teresa H.-Y. Meng received the B.S. degree from National Taiwan University in 1983, and the M.S. and Ph.D. degrees from the University of California at Berkeley, in 1984 and 1988, respectively.

Since 1988 she has been with the Department of Electrical Engineering, Stanford University as as assistant professor. Her current research activities include wireless data communication, asynchronous logic synthesis, and low-power design of real-time DSP applications.

Dr. Meng has received the IEEE Signal Processing Society's Best paper Award in 1988, the 1989 NSF Presidential Young Investigator Award, the 1989 ONR Young Investigator Award, a 1989 IBM Faculty Development Award, and the 1988 Eli Jury Award at U.C. Berkeley for recognition of excellence in systems research.