Figure 9: Constraint graph specification for the *refresh* cycle of DRAM controller: (a) initial specification and (b) strongly connected specification.

Figure 10: Timed circuit implementation of the DRAM controller.

Figure 7: Cyclic constraint graph specification for the traffic light controller.

Figure 8: Timed circuit implementation of the traffic light controller.

Figure 5: SIS implementation of a two-bit synchronous counter: (a) smallest implementation and (b) fastest implementation.

Figure 6: Deterministic version of Mead and Conway's traffic light controller: (a) state diagram and (b) block diagram.

Figure 3: Choice of gate structure for signals which occur multiple times in a cycle: (a) merge structure and (b) complex gate structure.

Figure 4: Complex-gate Implementation of a two-bit synchronous counter.

Figure 2: The cyclic constraint graph specification for a two-bit synchronous counter: (a) initial specification and (b) final specification.

Figure 1: Summary of relationships between system timing models.

Covers for Various Gate Structures							
State		Merged		Gate-level		Complex Gate	
$s_{k,1}, s_{k,2}$	s_k	$set_{k,1}, reset_{k,1}$	$set_{k,2}, reset_{k,2}$	$set_{k,1}, reset_{k,1}$	$set_{k,2}, reset_{k,2}$	$set_{k,1}, reset_{k,1}$	$set_{k,2}, reset_{k,2}$
00	0	0-	0-	00	0-	0-	0-
R0	R	10	0-	10	00	10	-0
10	1	-0	0-	-0	00	-0	-0
F0	F	01	0-	01	00	01	0-
00	0	0-	0-	0-	00	0-	0-
R0	R	0-	10	00	10	-0	10
10	1	0-	-0	00	-0	-0	-0
F0	F	0-	01	00	01	0-	01

Table 1: Desired logic over a set of states for a disjunctive merge.

	SIS		ATACS	
Examples	Area (transistors)	Delay (inverters)	Area (transistors)	Delay (inverters)
Two-bit counter	32	6	22	2.5
Traffic light controller	162	9.5	86	4
DRAM controller	178	9.5	70	3

Table 2: Comparision of timed circuit implementations with SIS.

References

[1] E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, University of California, Berkeley, May 1992.

[2] A. J. Martin. Programming in VLSI: from communicating processes to delay-insensitive VLSI circuits. In C.A.R. Hoare, editor, *UT Year of Programming Institute on Concurrent Programming*. Addison-Wesley, 1990.

[3] T.-A. Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications*. PhD thesis, Massachusetts Institute of Technology, 1987.

[4] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messershmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.

[5] P. A. Beerel and T. H.-Y. Meng. Automatic gate-level synthesis of speed-independent circuits. In *Proceedings IEEE 1992 ICCAD* Digest of Technical Papers, pages 581–586, 1992.

[6] S. M. Nowick and D. L. Dill. Synthesis of asynchronous state machines using a local clock. In *International Conference on Computer Design, ICCD-1991*. IEEE Computer Society Press, 1991.

[7] S. M. Nowick, K. Y. Yun, and D. L. Dill. Practical asynchronous controller design. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.

[8] K. Y. Yun, D. L. Dill, and S. M. Nowick. Synthesis of 3D asynchronous state machines. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.

[9] K. Y. Yun and D. L. Dill. Unifying synchronous/asynchronous state machine synthesis. In *International Workshop on Logic Synthesis*, 1993.

[10] P. Vanbekbergen, C. Ykman-Convreur, B. Lin, and H. de Man. Generalizing signal transition graphs for modeling asynchronous/synchronous and arbitration behavior. In *International Workshop on Logic Synthesis*, 1993.

[11] G. Borriello. *A New Specification Methodology and its Applications to Transducer Synthesis*. PhD thesis, University of California, Berkeley, 1988.

[12] P.A. Subrahmanyam. Automated synthesis of systems with interacting asynchronous (self-timed) and synchronous components. In *International Conference on Computer Design, ICCD-1989*. IEEE Computer Society Press, 1989.

[13] L. Lavagno, K. Keutzer, and A. Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*, 1991.

[14] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.

[15] A. Shen, A. Ghosh, S. Devadas, and K. Keutzer. "On Average Power Dissipation and Random Pattern Testability of CMOS Combinational Logic Networks". In *Proceedings IEEE 1992 ICCAD* Digest of Papers, 1992.

[16] S. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[17] T. Amon, H. Hulgaard, G. Borriello, and S. Burns. "Timing Analysis of Concurrent Systems". Technical Report UW-CS-TR-92-11-01, University of Washington, 1992.

[18] P. Vanbekbergen, G. Goossens, and H. de Man. "Specification and Analysis of Timing Constraints in Signal Transition Graphs". Technical report, Report ESPRIT 2260 Project, June 1991.

[19] E. J. McCluskey. *Logic Design Principles*. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[20] C. A. Mead and L. A. Conway. *Introduction to VLSI Systems*. Addison-Wesley, 1980.

[21] Ken Yun. *Private Communications*, 1992. Ken Yun is a graduate student at Stanford University.

[22] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.

4.2 Traffic-light Controller

Our second example is a deterministic version of the traffic light controller from [20]. In our version, the controller is used at the intersection of two busy streets. In other words, we assume that there are always cars on both streets, so the green light is alternated after a given delay between the two streets. The modified state diagram is shown in Figure 6a, and a block diagram for the circuit under design is shown in Figure 6b. The signal coding is the same as that given in [20].

A constraint graph specification for the traffic light controller is shown in Figure 7. Applying the timed circuit synthesis procedure, we derive a complex-gate implementation using only 86 transistors as shown in Figure 8 (4 are used to invert TL and TS and 56 are needed for the latches). All outputs are implemented in a single gate, so there is only a single gate and latch on the critical path with the largest gate and latch taking about 4 inverter delays. SIS produces a circuit which requires 162 transistors and a critical path of 9.5 inverter delays. Therefore, our implementation requires about half the area with less than have the delay.

4.3 DRAM Controller

Our last example is a DRAM controller which is an interface between a microprocessor and a DRAM array. Our specification is derived from a burst-mode specification described in [7]. The specification of the refresh cycle is shown in Figure 9(a). Notice that this constraint graph is not strongly connected. Applying Algorithm 2.1, the dashed arcs in Figure 9(b) are added to the constraint graph. The timing constraints used for the refresh cycle are also depicted in Figure 9(b) assuming the interface is with a a 68020/30 running at 16 to 20 MHz [21].

The specification 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 described in [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 such as the signal ras. Each occurrence is renamed, and the synthesis procedure described earlier is applied. This procedure leads to the implementation of the DRAM controller shown in Figure 10. Recall that although some of the gates are shown with multiple levels, they are all actually implemented as single complex gates. Our final implementation uses 36 transistors for the logic, 14 transistors to invert inputs, and we also add 20 transistors for staticizer circuitry. We estimate that the largest gate takes about 3 inverter delays. The best previously reported asynchronous version of this design used 110 transistors with a critical path of 4 inverter delays [8]. A synchronous implementation derived using SIS requires 178 transistors with a critical path of 9.5 inverter delays. Therefore, our design is more than 2 times smaller and more than 3 times faster.

5 Results and Conclusions

We have described a uniform approach to the design of both synchronous circuits and asynchronous circuits using a timing analysis procedure. We have applied our technique to several examples with the very surprising result that in addition to providing a uniform framework for synthesis of both types of circuits, significant reductions in area and delay can be achieved. Also, our implementations potentially use less power since they are glitch-free. This synthesis procedure has been fully automated in a CAD tool which has been used to compile all results reported in this paper as tabulated in Table 2.

Acknowledgments

We would like to thank Peter Beerel of Stanford University for many interesting discussions and comments on this manuscript. The authors would also like to thank Professor David Dill and Dr. Jerry Burch of Stanford University for their valuable guidance. Thanks also go to Professors 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.

the procedure generates a cover as described in Table 1. A dual procedure is used for conjunctive merges. The algorithm uses a function called *Problem* which 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 3.3 *(Find Conflicts)*
array FindConf(ER system $\langle E, R_{NR} \rangle$*; set set_of_states; array Q)* {
  For each state S *and each signal occurrence* $s_{k,l}$ *in* S
    If (s_k *is merged disjunctively) then*
      If (for every j *in which* $1 \leq j \leq mk$, $(s_{k,j} == F$ *or* $s_{k,j} == 0)$*) then*
        If (Problem$(S, s_{k,l}\uparrow, \{$*rules in* R'_{NR} *of the form* $\langle e, s_{k,l}\uparrow, \varepsilon, \tau \rangle\}))$ *then*
          For each signal s_i*, if (Solution*$(S, q_{(k,l)\uparrow,i}))$ *then* $C_{(k,l)\uparrow}[s_i, S] =$ *TRUE;*
      Else If (Problem$(S, s_{k,l}\downarrow, \{$*rules in* R'_{NR} *of the form* $\langle e, s_{k,l}\downarrow, \varepsilon, \tau \rangle\}))$ *then*
        For each signal s_i*, if (Solution*$(S, q_{(k,l)\downarrow,i}))$ *then* $C_{(k,l)\downarrow}[s_i, S] =$ *TRUE;*
    Else
      If (for every j *in which* $1 \leq j \leq mk$, $(s_{k,j} == R$ *or* $s_{k,j} == 1)$*) then*
        If (Problem$(S, s_{k,l}\downarrow, \{$*rules in* R'_{NR} *of the form* $\langle e, s_{k,l}\downarrow, \varepsilon, \tau \rangle\}))$ *then*
          For each signal s_i*, if (Solution*$(S, q_{(k,l)\downarrow,i}))$ *then* $C_{(k,l)\downarrow}[s_i, S] =$ *TRUE;*
      Else If (Problem$(S, s_{k,l}\uparrow, \{$*rules in* R'_{NR} *of the form* $\langle e, s_{k,l}\uparrow, \varepsilon, \tau \rangle\}))$ *then*
        For each signal s_i*, if (Solution*$(S, q_{(k,l)\uparrow,i}))$ *then* $C_{(k,l)\uparrow}[s_i, S] =$ *TRUE;*
  Return(C);
}

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 [19].

4 Examples

This section describes three examples: a two-bit counter, a traffic-light controller, and a DRAM controller. The two-bit counter and the traffic-light controller are synchronous designs. The DRAM controller is an asynchronous circuit which interfaces with a synchronous environment. For each example, a timed circuit implementation is synthesized, and the result is compared with a synchronous implementation designed using SIS. In each example, the timed circuit is substantially faster and smaller.

4.1 Two-bit Synchronous Counter

Our first example is the two-bit synchronous counter which we gave a specification for earlier in Figure 2. The complex gate structure synthesized for the counter is shown in Figure 4a. The gate for C_0 can be obviously optimized since the gates for the separate regions are the same and can be shared as shown in Figure 4b. Upon closer inspection of the transistor-level diagram for this gate shown in Figure 4c, we observe that the gates are actually typical synchronous latches, and the circuit can be redrawn as shown in Figure 4d. This final implementation takes 6 transistors for the logic and 16 for clock latches. The critical path through the logic is an inverter, a pass gate, and a latch (approximately 2.5 inverter delays). Using SIS and a standard synchronous gate library, the implementation for the counter shown in Figure 5 is derived. The implementation uses 32 transistors and has a critical path through an inverter and 2 NAND gates and a latch (approximately 6 inverter delays).

Our implementation is more than 30 percent smaller and more than twice as fast as that produced using the synchronous synthesis tool. Comparing the implementations, we find that both implement C'_0 using a single inverter. The difference is in the implementation of C'_1. Our timed circuit implementation makes use of the information that C'_1 only changes in states where C_0 is high. Thus, it is implemented using an inverter and a pass gate which is gated on C_0. SIS' implementation, on the other hand, does not take into account the sequencing of the states. For example, if a sequence of states in which the counter is counting 00-11-01-10 were possible, this circuit would generate the correct next state given the current state. This extra logic, however, is unnecessary since this counter always goes through the states in the same order: 00-01-10-11-00,etc.

care). The second approach is a simplified version of what is used for a gate-level implementation of the structure shown in Figure 3b (the cover actually needs to be connected [5]). The third approach is for the complex gate structure in which additional *don't cares* are allowed over the gate-level implementation. Consider, for example, the *don't care* added for $reset_{k,1}$ in the first state. This is not allowed in a gate-level implementation because it is not possible to acknowledge the transition on $reset_{k,1}$, since $reset_{k,2}$ has already caused s_k to transition low. However, since we implement this as a single complex gate, this transition is on a signal which is internal to the gate so it does not need to be acknowledged. Therefore, using this *don't care* information can result in smaller covers needed for the complex gate approach.

While, in general, these complex gates can become large, the largest gate in our examples uses 12 transistors with at most 4 transistors in a stack. In fact, as illustrated later, these single-level complex gate implementations can be significantly smaller than multi-level implementations derived using synchronous design tools. This is primarily due to the sequential state information which we use in our synthesis procedure. Synchronous procedures typically use a finite-state machine approach in which the resulting circuit must be prepared to enter any state at any time. Therefore, they tend to use combinational logic rather than the state-holding logic used in our procedure, resulting a large penalty in area.

3.2.3 Finding the Enabled State

In order to transform our reduced state graph into a complex gate implementation, we derive an *enabled state* for each transition. The enabled state for a transition is the merged value of each signal in all states in which that transition is enabled to occur. From this, we can determine which signals are stable during a particular transition, and thus, can be used in the implementation for that transition. Using the complex gate structure that we have chosen, only the merged value for a signal can be used, so the values of the signal occurrences are merged in the enabled state. The enabled state is defined more formally in Definition 3.5. Algorithm 3.2 shows how the enabled state for each transition can be found from the reduced state graph.

Definition 3.5 *(Enabled State) For each transition $s_{k,l}\uparrow$, the enabled state is of the form $Q_{(k,l)\uparrow} = q_{(k,l)\uparrow,1}, \ldots, q_{(k,l)\uparrow,i}, \ldots, q_{(k,l)\uparrow,n}$, where n is the number of signals in the specification. Each $q_{(k,l)\uparrow,i}$ is determined as follows: if in all states where $s_{k,l} = R$, MVAL$[s_i] = 0$ then $q_{(k,l)\uparrow,i} = 0$; if in all states where $s_{k,l} = R$, MVAL$[s_i] = 1$ then $q_{(k,l)\uparrow,i} = 1$; otherwise, $q_{(k,l)\uparrow,i} = X$. The enabled state for the transition $s_{k,l}\downarrow$ is similarly defined.*

Algorithm 3.2 *(Find Enabled State)*

```
array FindES(ER system ⟨E, R⟩; set set_of_states) {
    For each signal occurrence s_{k,l}, Q_{(k,l)↑} = Q_{(k,l)↓} = undefined, ..., undefined;
    For each state and each signal occurrence s_{k,l} in each state
        If (s_{k,l} == R) then
            For each signal s_i
                If (q_{(k,l)↑,i} == undefined) then q_{(k,l)↑,i} = MVAL[s_i];
                Else if (q_{(k,l)↑,i} ≠ MVAL[s_i]) then q_{(k,l)↑,i} = X;
        Else if (s_{k,l} == F) then
            For each signal s_i
                If (q_{(k,l)↓,i} == undefined) then q_{(k,l)↓,i} = MVAL[s_i];
                Else if (q_{(k,l)↓,i} ≠ MVAL[s_i]) then q_{(k,l)↓,i} = X;
    Return(Q);
}
```

3.2.4 Detecting and Resolving Conflicts

The next step is to check for *conflicts* in each state. A conflict occurs when, using only the non-redundant rules, a transition can occur in a particular state, but in that state the merged value of the signal is enabled to change or has changed to the opposite value. The result is a hazard which must be prevented. To prevent a conflict, *context signals* are added to guarantee that the transition 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 its merged value in the problem state. Algorithm 3.3 is used to detect when conflicts are possible and to generate a table of possible context signals which can be added to solve each conflict. The procedure taken for each signal depends on whether the signal is merged disjunctively or conjunctively. If it is a disjunctive merge,

$R_O = R_O - \{\langle e, f, i, \varepsilon, \tau\rangle \mid i \geq \varepsilon\}$;
} *Else EXCEPTION("Input Ordering Rule Not Satisfied");*
}
Return(R_{NR});
}

3.2 Synthesis with Multiple Occurrences of an Event in a Cycle

As mentioned earlier, events that occur multiple times in a cycle, such as transitions on a clock signal, are given a unique name for each occurrence. This is necessary for the timing analysis procedure used to find redundant rules (see Algorithm 3.1) and to derive a *reduced state graph* [14]. These events, however, must be recombined to construct a circuit implementation. The procedure for doing this is described this section.

3.2.1 Expanded States

Our synthesis procedure derives its implementation from a reduced state graph. Essentially, a reduced 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. It is reduced because our timing analysis has been employed to detect and remove states which the system will never enter given the timing constraints.

In this paper, we use an *expanded state* defined in Definition 3.2 which splits each signal into a set of values for each of the pair of signal transitions on that signal. The value of a signal occurrence in a state is defined in Definition 3.3 while the value of a signal in a state is evaluated over all the signal occurrences as defined in Definition 3.4.

Definition 3.2 *(Expanded State) Each expanded state S is of the form $S = (s_{1,1}, \ldots, s_{1,m1})$, $\ldots$, $(s_{k,1}, \ldots, s_{k,mk})$, $\ldots$, $(s_{n,1}, \ldots, s_{n,mn})$, where n is the number of signals in the specification and mk is the number of occurrences of a pair of transitions on the signal s_k. Each signal occurrence $s_{k,l}$ has the value: 0 if the signal $s_{k,l}$ is low, R if the signal $s_{k,l}$ is low but enabled to rise, 1 if the signal $s_{k,l}$ is high, and F if the signal $s_{k,l}$ is high but enabled to fall.*

Definition 3.3 *(Value) The function VAL$[s_{k,l}] = 0$ if $s_{k,l} = 0$ or R and VAL$[s_{k,l}] = 1$ if $s_{k,l} = 1$ or F.*

Definition 3.4 *(Merged Value) The function MVAL$[s_k] = \bigvee_{l=1}^{mk}$ VAL$[s_{k,l}]$ for a disjunctive merge, and MVAL$[s_k] = \bigwedge_{l=1}^{mk}$ VAL$[s_{k,l}]$ for a conjunctive merge.*

3.2.2 Choices of Gate Structure

There are several different implementations that may be derived for signals which occur multiple times in a cycle depending on the design decisions made. The simplest approach is to synthesize each signal occurrence separately and to merge the result. This *merge structure* is shown in Figure 3a in which each signal occurrence is synthesized separately in a single complex gate (a *generalized C-element* [2]), and the result is merged with an *OR*-gate for a disjunctive merge (an *AND* gate is used for a conjunctive merge). The major advantage of this method is that the logic can often be the simplest due to having a lot of *don't cares* and having the signal occurrence available for use in the cover. The disadvantages are that there are multiple storage elements which can incur a significant area penalty, and the logic is multi-level which results in additional delay and a need for analysis of hazards on the internal signals.

Another approach shown in Figure 3b is a *complex gate structure* which implements the entire signal in a single complex gate (note that while the gate is shown as multi-level, it is easily implemented as a single compact complex gate). The disadvantage of this structure is that it may need more complicated logic to implement a particular signal occurrence than the merge structure. We believe, however, that the advantages to be gained from needing only a single storage element and the sharing of logic by different signal occurrences outweight this disadvantage in most cases. Therefore, this structure is the one which we use throughout the rest of the paper.

Although a gate-level synthesis method that maps to this structure could be used [5], simpler logic can be derived based on additional *don't cares* available when using complex gates. In Table 1, the values desired for the logic generating the appropriate set or reset for each signal occurrence, for several possible implementations, over a series of states are given. The first column represents the logic for the merge structure (note that a "-" represents a *don't*

These events are treated separately during the timing analysis; they must, however, be recombined during synthesis as described in a later section.

There are two different orderings in which the events can occur: ($s_{k,1}\uparrow$, $s_{k,1}\downarrow$, $s_{k,2}\uparrow$, $s_{k,2}\downarrow$, ..., $s_{k,m}\uparrow$, $s_{k,m}\downarrow$) or ($s_{k,1}\downarrow$, $s_{k,1}\uparrow$, $s_{k,2}\downarrow$, $s_{k,2}\uparrow$, ..., $s_{k,m}\uparrow$, $s_{k,m}\downarrow$) where m is the number of multiple occurrences of a transition pair on a signal s_k in a cycle. The first ordering requires a *disjunctive merge* because $s_k = \bigvee_{l=1}^{m} s_{k,l}$, and it occurs when the signal s_k is initially low. The second requires a *conjunctive merge* because $s_k = \bigwedge_{l=1}^{m} s_{k,l}$, and it occurs when the signal s_k is initially high. In order to infer the type of merge to use for a particular signal, we require that the initial value of each signal is specified. The counter specification with unique event names is shown in Figure 2(b), and for the initial state information, we assume that all signals are initially low.

3 The Synthesis Procedure

From a timed circuit specification, we systematically derive a complex gate implementation. This section describes our synthesis procedure for timed circuits as it is applied to the design of synchronous circuits and asynchronous circuits that interface to a synchronous environment. This procedure incorporates sequential state information and uses complex gates, resulting in significant area and delay improvements over combinational synchronous synthesis methods.

3.1 Verifying Ordering Rules

The first step in the synthesis procedure is to determine which rules are *redundant*. A rule is redundant if its omission does not change the behavior specified. If a causal rule is found to be redundant, it can result in a simpler circuit implementation. When designing synchronous circuits or asynchronous circuits interfacing with a synchronous environment, we typically need to add ordering rules. If an ordering rules that enables input signal is found not to be redundant then our procedure cannot derive an implementation that will satisfy the given timing constraints because we do not allow our procedure to modify the specification of the environment. If, on the other hand, an ordering rule which enables an output signal is determined not to be redundant, then the rule is added as a causal rule.

In order to determine which rules are redundant, a function called *WCTimeDiff* is called to determine an estimate of the worst-case time difference between two events as defined in Definition 3.1. A heuristic polynomial-time algorithm to determine this value is given in [14]. An exponential-time algorithm to find the exact worst-case time difference is the subject of [17].

Definition 3.1 *(Estimate of the Worst-Case Time Difference) Given two events, u and v, and the occurrence-index offset between them j where $j \geq 0$, an estimate of the worst-case time difference between these two events is defined to be the bound $[L', U']$ where for all values of $i \geq j$:*

$$L' \leq t(\langle v, i\rangle) - t(\langle u, i-j\rangle) \leq U'$$

Algorithm 3.1 is used to find all redundant causal rules and verify that all ordering rules are satisfied. It can be shown that a causal rule is redundant if the lower-bound of the worst-case time difference L' between the enabled event and the enabling event is larger than the upper-bound of timing constraint for the rule u [14]. We can also use the same procedure to verify whether ordering rules are redundant. This is achieved, by utilizing the fact that if the following relation is true, then the ordering rule is redundant.

$$t(\langle v, i\rangle) - t(\langle u, i-j\rangle) \geq L' > u = l.$$

Algorithm 3.1 *(Find Redundant Rules)*
set FindRed(ER system $\langle E, R\rangle$,set of ordering rules R_O) {
  $R_{NR} = R \cup R_O$;
  For each rule in R of the form $\langle e,f,i,\varepsilon,\tau\rangle$ {
    $[L', U']$=*WCTimeDiff*$(\langle E, R \cup R_O\rangle, e, f, \varepsilon)$;
    If ($L' > u$) then $R_{NR} = R_{NR} - \{\langle e, f, i, \varepsilon, \tau\rangle \mid i \geq \varepsilon\}$;
    Else if ($\langle e,f,i,\varepsilon,\tau\rangle$ in R_O) then
      If (f is a transition on an output signal) then {
        $R = R + \{\langle e, f, i, \varepsilon, \tau\rangle \mid i \geq \varepsilon\}$;

path from u to v. For specifications of asynchronous circuits which interface with other asynchronous circuits, the constraint graph is usually strongly connected. This is not true, however, for asynchronous circuits that interface with synchronous circuits or for synchronous circuits, such as the counter presented earlier.

In order to satisfy the property of strong connection, *ordering rules* are added to the specification. All rules defined above are *causal rules* which are enforced with actual circuitry. Ordering rules, on the other hand, specify ordering relationships between events which must be *satisfied* by the circuitry, but not *enforced* by circuitry. In other words, as defined in Definition 2.3, an ordering rule is satisfied if the time difference between the enabled and enabling event is always greater than the given timing constraint. As opposed to the ordering constraints discussed in [18], our ordering rules only place a lower bound on this time difference. In order for the synthesis to be valid, all ordering rules must either be satisfied, or they must be made causal rules. An algorithm to verify that ordering rules are satisfied is given in a following section.

Definition 2.3 *(Ordering Rule) An ordering rule is a rule of the form* $\langle e, f, \varepsilon, \tau \rangle$ *where* $\tau = [l, u]$ *and* $l = u$. *Given that* $t(\langle f, i \rangle)$ *is the time of the* i^{th} *occurrence of the event* f, *the rule is satisfied if for all values of* $i > \varepsilon$, $t(\langle f, i \rangle) - t(\langle e, i - \varepsilon \rangle) > l$.

In order to select a minimal set of ordering rules to be added to make a constraint graph strongly connected, rules are added according to the *persistence* property. Persistence essentially guarantees that once a rule is enabled it cannot be disabled without the enabled event first occurring [3] [4]. It can be shown that if a cyclic constraint graph has the persistence property then the graph is strongly connected. Furthermore, rules added according to the persistence property place the weakest restrictions on concurrency implementable with our synthesis procedure.

Algorithm 2.1 adds ordering rules until the ER schema is strongly connected. If the graph is not initially strongly connected, it adds ordering rules to solve each persistence problem found in the original constraint graph using the procedure *Add_Persistence_Rules*. Algorithms for *Add_Persistence_Rules* can be adapted from those in [3] [4]. Essentially, for each rule $\langle e, f, \varepsilon, \tau \rangle$, it checks that the event $\neg e$ is reachable from the event f. If this is not the case, an ordering rule of the form $\langle f, \neg e, \varepsilon, [0, 0] \rangle$ is added. The timing constraint for this rule is $[0, 0]$ which means that only the ordering of the two events is important and not the time difference between them. The value of ε is determined from occurrence and initial state information. This procedure is repeated until the graph is strongly connected. Since persistence implies strong connectivity, we are guaranteed that this algorithm will terminate with a strongly connected constraint graph. Since the algorithm is terminated as soon as the graph is found to be strongly connected, the resulting graph may not satisfy the persistence property. If we apply Algorithm 2.1 to the counter specification presented earlier, we obtain the strongly connected graph shown in Figure 2(b).

Algorithm 2.1 *(Make Strongly Connected)*
set MakeSC(ER schema $\langle E', R' \rangle$*)* {
    $R'_O = \emptyset$;
    While not (strongly_connected($\langle E', R' \cup R'_O \rangle$*))*
        $R'_O = R'_O \cup$ *Add_Persistence_Rules(*$\langle E', R' \cup R'_O \rangle$*);*
    Return(R'_O*);*
}

We utilize Algorithm 2.1 as a preprocessing step on our specification. Therefore, we can weaken the well-formed requirement to be that the graph is initially *semiconnected*, (i.e., for every two events u and v there exists a path either from u to v or from v to u). We also require that all events appear as an enabled event in some rule.

Ordering rules may also be added to specify specific timing requirements. These ordering rules differ from those added by Algorithm 2.1 in that they may have a specific timing constraint (i.e., $[l, u] \neq [0, 0]$). For example, if we want to specify that the event $C'_0 \uparrow$ occurs at least 5 time units before the event $\varphi \downarrow$, an ordering rule of the form $\langle C_0 \uparrow, \varphi \downarrow, 0, [5, 5] \rangle$ can be added.

2.3 Multiple Occurrences of an Event in a Cycle

The timing analysis used by our synthesis procedure requires that each event in an ER schema is uniquely identified. This property does not hold for synchronous systems such as the counter, since the clock signal, as well as possibly other signals, transition multiple times in a specification. Therefore, to allow specifications with multiple occurrences of an event in a cycle, each occurrence of each event is given a unique name. For example, a signal s_k specified to rise and fall twice in a cycle, is renamed to $s_{k,1}$ for the first rising and falling transitions and $s_{k,2}$ for the second.

2 Timed Circuit Specification Language

The specification language for our timed circuits is the *event-rule (ER) system* as introduced in [16] and modified in [14]. The ER system can be presented in the form of a *cyclic constraint graph* as illustrated by an example of a two-bit synchronous counter shown in Figure 2(a). To specify synchronous designs, two properties of this specification call for special attention. First, the constraint graph is not strongly connected. This is typical of synchronous designs because all events are caused by transitions on the clock signal, and therefore, there are no arcs leading from transitions on other signals. Second, there are multiple occurrences of some events, such as the rising transition of the clock signal $\varphi\uparrow$. In this section, we describe our specification language and how to transform a specification with the above two properties to one that can be synthesized.

2.1 Event-Rule System

A brief review of the ER system is given here. For a more detailed description see [16] [14]. An ER system is composed of a set of atomic actions, *events*, and the causal dependencies between them, *rules*. 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. Each rule is composed of an *enabling event*, an *enabled event*, and a *bounded timing constraint*. 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 its timing constraint. 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 its timing constraint. An event cannot occur until *all* rules enabling it are satisfied. An event must always occur before *every* rule enabling it has expired. Thus, the causality requirement is *conjunctive*.

An ER system can be specified using an *ER schema* and initialization information. An ER schema defines the cyclic constraint graph which is a weighted marked graph in which the vertices are the events, the arcs are the rules, and the weights are the bounded timing constraints. 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 essentially 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.*

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. 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 $\langle reset, 0\rangle$ is added to E. For each rule in R' in which $\varepsilon = 1$, a rule of the form $\langle reset, f, 0, 0, \tau_0\rangle$ is added to R.*

2.2 Ordering Rules

A requirement of the cyclic constraint graph derived from the ER schema is that it 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]. A cyclic constraint graph is strongly connected if for every two events u and v there exists a

1 Introduction

As the sizes of systems grow, there has been an increasing interest in the design of systems containing a mixture of synchronous and asynchronous modules. In small systems, synchronous design is often preferred due to its simplicity. In large systems, however, communication distances can be very long and variable, and different modules can operate at very different rates. In these cases, the worst-case communication delay may be significantly different than the average-case, and thus, asynchronous communication may be necessary. These problems have even become apparent at the chip level where integration density has made distributing a global clock across a single chip a task which requires significant amounts of chip area, power, and design time.

There has been considerable successful research in the development of synthesis tools for synchronous circuits. SIS is one such tool which is used in our comparisons later [1]. Recently, there has also been several synthesis procedures developed for asynchronous circuits. Most synthesis procedures, however, have been limited to the design of either synchronous or asynchronous circuits. On the one hand, synchronous design tools make no effort to remove hazards or races which would cause erratic behavior in an asynchronous circuit. While, on the other hand, asynchronous design tools typically do not incorporate timing analysis and cannot easily handle time-dependent behavior. Many synthesis procedures for asynchronous circuits use the *speed-independent* model in which behavior is independent of individual gate delays [2, 3, 4, 5]. With this model, the circuit is only guaranteed to work if the environment is another asynchronous circuit.

In order to synthesize asynchronous circuits that interface with a synchronous environment, synthesis procedures have been developed which use a *fundamental-mode* model in which allowable concurrency is limited, and assumptions on delays are required [6, 7, 8]. Essentially, fundamental-mode requires that the environment wait long enough for the circuit to stabilize before inputs can be changed. Extensions have recently been introduced to allow this model to be used to specify and synthesize systems with synchronous behavior [9]. Similar extensions have been introduced for *signal transition graphs (STG)* used in speed-independent synthesis [10]. These approaches do not, however, use explicit timing analysis in their synthesis procedure.

There have been some earlier attempts at synthesis procedures which incorporate timing analysis [11, 12, 13]. These techniques, however, do not produce optimal results since timing analysis is only performed after synthesis to verify hazards do not exist, and delay is added or the specification is transformed to avoid these hazards. While techniques are described in [11, 12] for both synchronous and asynchronous design, modules must be determined to be either synchronous or asynchronous and different synthesis procedures are applied for each.

In [14], a synthesis procedure for *timed circuits* is described. In this paper, we propose that timed circuits are actually a superset of both synchronous and asynchronous circuits. Timed circuits can be specified using both causal relationships and timing constraints. To specify a speed-independent circuit, the timing constraints are not used. By adding timing constraints, asynchronous circuits with a synchronous environment can be specified. Finally, by treating the clock signal as just another input, synchronous circuits can be specified. The relationships between the various models are summarized in Figure 1. In the following sections, we describe how to apply our timed circuit synthesis procedure to the design of synchronous circuits and asynchronous circuits which interface with a synchronous environment.

In addition to providing a uniform synthesis approach for both synchronous and asynchronous circuits, our procedure results in significantly smaller and faster circuits when compared with circuits synthesized using Berkeley's SIS, an academic synchronous design tool. We demonstrate that for several examples, our timed circuits can be more than 2 times smaller and more than 3 times faster compared with those synthesized by SIS. These surprising results can be attributed to two features of our synthesis procedure. First, the logic is reduced by utilizing sequential state information to guide synthesis which combinational synchronous synthesis procedures are not capable of. Second, each signal is implemented using a single complex gate, reducing the number of gates on the critical path. Utilizing our state information, we make these complex gates state-holding, and thus, they can also be very compact. Finally, since our circuits are designed to be glitch-free and glitching can cause 20 to 70 percent of the power dissipated in CMOS circuits [15], they consume less power.

This paper contains five sections. Section 2 describes our timed circuit specification language. Section 3 discusses our synthesis procedure as applied to synchronous design. Section 4 presents several examples. Section 5 gives our results and conclusions.

Contents

1 Introduction **1**

2 Timed Circuit Specification Language **2**
2.1 Event-Rule System 2
2.2 Ordering Rules 2
2.3 Multiple Occurrences of an Event in a Cycle 3

3 The Synthesis Procedure **4**
3.1 Verifying Ordering Rules 4
3.2 Synthesis with Multiple Occurrences of an Event in a Cycle 5
3.2.1 Expanded States 5
3.2.2 Choices of Gate Structure 5
3.2.3 Finding the Enabled State 6
3.2.4 Detecting and Resolving Conflicts 6

4 Examples **7**
4.1 Two-bit Synchronous Counter 7
4.2 Traffic-light Controller 8
4.3 DRAM Controller 8

5 Results and Conclusions **8**

Copyright ©1994

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

A UNIFORM APPROACH TO THE SYNTHESIS OF SYNCHRONOUS AND ASYNCHRONOUS CIRCUITS

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

Technical Report: CSL-TR-94-650

December, 1994

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055
pubs@shasta.stanford.edu

Abstract

In this paper we illustrate the application of a synthesis procedure used for timed asynchronous circuits to the design of synchronous circuits. In addition to providing a uniform synthesis approach, our procedure results in circuits that are significantly smaller and faster than those designed using the synchronous design tool SIS.

Key Words and Phrases:

Computer-aided design, asynchronous circuit synthesis, synchronous circuit synthesis, timed circuits, event-rule systems, and generalized C-elements.

A UNIFORM APPROACH TO THE SYNTHESIS OF SYNCHRONOUS AND ASYNCHRONOUS CIRCUITS

Chris J. Myers
Teresa H.-Y. Meng

Technical Report: CSL-TR-94-650

December, 1994

This research was supported by an NSF fellowship, ONR Grant no. N00014-89-J-3036, and research grants from the Center for Integrated Systems, Stanford University and the Semiconductor Research Corporation, Contract no. 92-DJ-205.