

Efficient Timing Analysis Algorithms for Timed State Space Exploration*

Wendy Belluomini
Computer Science Department
University of Utah
Salt Lake City, UT 84112

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

Abstract

This paper presents new timing analysis algorithms for efficient state space exploration during timed circuit synthesis. Timed circuits are a class of asynchronous circuits that incorporate explicit timing information in the specification which is used throughout the synthesis procedure to optimize the design. Much of the computational complexity in the synthesis of timed circuits currently is in finding the reachable timed state space. We introduce new algorithms which utilize geometric regions to represent the timed state space and partial orders to minimize the number of regions necessary. These algorithms operate on specifications sufficiently general to describe practical circuits.

1. Introduction

There has been a renewed interest in asynchronous circuits in recent years due to their advantages over synchronous circuits in performance and power consumption and as a way to eliminate problems related to clock skew [3, 5, 9, 16, 17]. However many of these advantages are often reduced or eliminated completely when the additional overhead in both speed and area that is required to build correct asynchronous circuits is included. This overhead mostly derives from the necessity to design a circuit that works correctly while making few or no assumptions about the timing of the signal transitions involved. This often leads to additional time and space being spent in the circuit to deal with contingencies that never happen. *Timed circuits*, which are designed using timing information, can often perform much better than asynchronous circuits designed neglecting explicit timing information [11, 1].

Timing information can either be used after the initial circuit has been designed to optimize out unnecessary circuitry [1] or in the process of the design to avoid generating it in the first place [11]. This paper concentrates on the second

case, although the techniques presented here may be applicable to the first case as well. In order to generate a circuit for a given specification, the design tool must find the reachable state space. This is generally done without including timing information, since adding timing information is thought to complicate an already exponential problem. However, in practice, timing information can often designate large portions of the state space as unreachable and therefore reduce the time it takes to generate the reachable state space and synthesize the circuit. As long as the timing analysis part of the exploration does not add too much overhead to the design process, it is worthwhile to do timing analysis at the same time as state space exploration.

Most of the timing analysis algorithms currently in use are not well suited for general timed state space exploration. They either place too many limitations on the types of specifications they can analyze, or they require that the complete specification be analyzed for each pair of events. Since timed state space exploration may involve unrolling a cyclic specification many times, keeping around the timing information for all events can cause an unacceptable overhead. In [14], algorithms are presented to do timed state space exploration while storing only local timing information. However, the algorithms only work on a restricted class of specifications where the firing time of each event can only be controlled by a single predecessor event.

This paper presents four new algorithms for timed state space exploration. The first extends the basic geometric timing analysis algorithm presented in [14] to allow the firing time of each event to be constrained by multiple predecessor events. The second shows how partial order information can be applied to efficiently determine a conservative approximation of the timed state space for the same extended class of specifications. The third algorithm is a partial order timing algorithm which finds the exact timed state space. This algorithm, however, can increase the computational complexity significantly, since it requires more than local timing information. Therefore, the fourth algorithm adds new data structures that allow the algorithm to find the exact timed state space with only local timing information. Each of these

* This research is supported by a grant from Intel Corporation, NSF CAREER award MIP-9625014, and an NSF Traineeship award.

algorithms maintains the advantages of the approach taken in [14], while eliminating the restriction. This allows efficient timed state space exploration of a much more general class of specifications.

2. Timed state space exploration

The objective of timed state space exploration is to take a specification of a circuit that has timing information and produce its reachable state space. This section presents the specification structure that we use and the generic algorithm that can be used to explore the state space of these structures.

2.1. Timed ER structures

Many methods have been proposed to specify timed systems. Two well known methods include *time Petri nets* where timing constraints are assigned to the transitions and *timed Petri nets* where timing is assigned to the places. All of the algorithms presented in this paper work on *timed event-rule (ER) structures* [11], which can represent a set of specifications equivalent to those represented by both types of Petri nets. Timed ER structures can also represent specifications such as the one in Figure 1(b) that cannot be easily translated into either Petri net semantics without somewhat complex graph transformations. We have also shown that timed ER structures can be automatically generated from some higher level languages such as CSP[11] or VHDL [21]. Since timed ER structures separate causality from conflict, they are both easier to generate from high level descriptions, and easier to analyze.

A timed ER structure S can be represented with the tuple $\langle A, E, R, \# \rangle$ where:

1. A is the set of atomic actions;
2. $E \subseteq A \times (N = \{0, 1, 2, \dots\})$ is the set of events;
3. $R \subseteq E \times E \times N \times (N \cup \{\infty\})$ is the set of rules;
4. $\# \subseteq E \times E$ is the conflict relation.

The set A contains the atomic actions possible in the system. The occurrence of an action is an *event* and is denoted (a, i) where a is the action and i is an *occurrence index* for the action. The rule set R represents a causal dependence between events. Each rule, of the form $\langle e, f, l, u \rangle$ is composed of an *enabling event* e , an *enabled event* f , and a *bounded timing constraint* $\langle l, u \rangle$. A rule is *enabled* if its enabling event has occurred. The timing constraint places a lower and upper bound on the timing of a rule. A rule is *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. Ignoring conflict, 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 differences in time between the enabled event and some enabling events exceed the upper bound of their timing constraints, but not for all enabling events.

The conflict relation is added to model *disjunctive* behavior and choice. When two events e and e' are in conflict, (denoted $e \# e'$), this specifies that either e can occur or e' can occur, but not both. Taking the conflict relation into account, if two rules have the same enabled event and conflicting enabling events, then only one of the two mutually exclusive enabling events needs to occur to cause the enabled event. This models a form of disjunctive causality. *Inherently disjunctive* behavior, or true OR causality, cannot currently be modeled, but we are working on techniques to address this. Choice is modeled when two rules have the same enabling event and conflicting enabled events. In this case, only one of the enabled events can occur.

If a specification is cyclic, then the timed ER structure representing it is infinite. However, due to its repetitive nature, this infinite behavior can be described with a finite model by adding an additional set of rules and conflicts which recursively define the infinite structure [11].

An example of a timed ER structure is shown in Figure 1(a). The vertices are events and the arcs are rules. Each rule is labeled with the delay range associated with it. Tokens can be used to indicate that a rule's enabling event has fired. An arc with a tick mark represents an *initial rule*. Ignoring the initial rule, this structure specifies that after A occurs, either B or C can occur, but not both since they conflict. If B occurs, D follows 1 to 5 time units later. If C occurs, E follows 2 to 6 time units later. F happens after either D or E . Note that since D and E conflict that only one of these events needs to occur to cause F . One other interesting note is that C happens at most 5 time units after A even though it is specified to have a maximum of 6. At time 5, a choice between B and C must be made, since if time advanced the rule between A and B would expire. This semantics is the same as the one used in time Petri nets. Timed Petri nets cannot model this behavior because a single timing constraint must be given for the choice place leading to B and C . If the timed Petri net semantics is desired, then we can simply set the timing constraint between A and B and between A and C to be equal.

The infinite structure can be derived from cyclic representation of the timed ER structure by cutting the graph at the tick marks and giving each event an occurrence index of 0. Each initial rule in the cyclic graph is then appended to the cut graph with an enabled event which has an index greater than any other previous event with the same action. Finally,

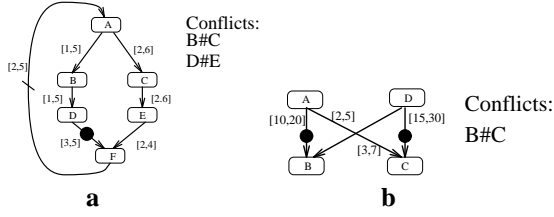


Figure 1. Example of a timed ER structure.

the rest of the rules are added with increased indices. This process can be repeated to obtain a structure of arbitrary size. The formal details are given in [11].

2.2. Timed Configurations

We define the behaviors specified by a timed ER structure using *timed configurations* [11]. Winskel defined the allowed behaviors of event structures as subsets of events, or *configurations* [20]. In order to add timing, we introduce timed configurations in which each event is paired with the time of its occurrence.

The first requirement for a subset of events to be a configuration is that it must be *conflict-free*. In other words, if two events are in conflict, they cannot both occur in a configuration. Winskel defined Con to be the set of finite conflict-free subsets of E , i.e. $Con \subseteq 2^E$, defined as follows:

$$Con = \{X \mid (X \subseteq E) \wedge (\forall e, e' \in X : \neg(e \# e'))\}.$$

In order to add timing, we define $TCon$ to be the set of conflict-free subsets of events in which each event is paired with the real-valued time that the event occurred (i.e., $TCon \subseteq 2^{E \times \mathbb{R}}$). To obtain Con from $TCon$, we define the function $untime : TCon \rightarrow Con$ in the obvious way.

The second requirement is that all events in the subset must be *time-secured*. Informally, this means that for each event in the set, all the events needed to enable the event are also in the set. To define this formally, we must first define when an event is enabled. The *untimed enabling relation* ($\vdash \subseteq Con \times E$) is defined as follows:

$$X \vdash f \Leftrightarrow [(\langle e, f, l, u \rangle \in R) \Rightarrow ((e \in X) \vee (\exists e' \in X : (e \# e') \wedge \langle e', f, l', u' \rangle \in R))]$$

Intuitively, this says given that the events in the set X have occurred that the event f is *untimed enabled*. This is true when a set of non-conflicting enabling events in rules in which f is the enabled event are in the set X . To incorporate timing, we now define the *timed enabling relation* ($\vdash_t \subseteq TCon \times \mathbb{R} \times E$) as follows:

$$Z \vdash_t f \Leftrightarrow [(untime(Z) \vdash f) \wedge (\forall (e, t') \in Z : \langle e, f, l, u \rangle \in R \Rightarrow t \geq t' + l)]$$

Intuitively, this says that given that the set of event-time pairs in Z have occurred and time has advanced to time t , the event f is *timed enabled*. This is true when f is untimed-enabled, and at time t the lower bounds of all timing constraints have been satisfied. With this relation, we can now define *time-secured* $\subseteq TCon \times E$ as follows:

$$time-secured(Z, e) \Leftrightarrow [\exists (e_0, t_0), \dots, (e_n, t_n) \in Z : e_n = e \wedge \forall i \leq n : \{(e_0, t_0), \dots, (e_{i-1}, t_{i-1})\} \vdash_{t_i} e_i]$$

Intuitively, this says that given that the set of event-time pairs in Z have occurred, the event e is time secured if and only if a sequence of events can be found in Z which lead to e being timed enabled.

The third requirement for a subset of events to be a configuration is that it is *non-expired*. Informally, this means that an enabled event must occur before it *expires*. An event is expired when for each of the rules enabling it, the time since the enabling event has exceeded the upper bound of the timing constraint. We define a relation *expired* $\subseteq TCon \times E \times \mathbb{R}$ as follows:

$$expired(Z, f, t) \Leftrightarrow [(Z \vdash_t f) \wedge (\forall (e', t') \in Z : \langle e', f, l, u \rangle \in R \Rightarrow t > t' + u)]$$

Using this relation, we say a timed configuration Z is non-expired if for each of the events either the event has occurred and was not expired when it occurred, a conflicting event occurred and was not expired when that event occurred, or it has not occurred and is not expired at the latest time of any event occurrence in the configuration. We define the relation *non-expired* $\subseteq TCon \times E$ as follows:

$$non-expired(Z, f) \Leftrightarrow [(\exists t : (f, t) \in Z \wedge \neg expired(Z, f, t)) \vee \exists (f', t) \in Z : (f \# f') \wedge \neg expired(Z, f, t) \vee \neg expired(Z, f, \max_{(e,t) \in Z} \{t\})]$$

Now, we can define all the timed configurations specified by a timed ER structure. For a timed ER structure $S = \langle A, E, R, \# \rangle$, a timed configuration of S is a subset of event-time pairs $Z \subseteq E \times \mathbb{R}$ which is:

1. conflict-free: $Z \in TCon$;
2. time-secured: $\forall e \in untime(Z) : time-secured(Z, e)$;
3. non-expired: $\forall f \in E : non-expired(Z, f)$.

2.3. Generic algorithm

In order to explore the state space, the transition relation between states must be defined. In the previous sections, an event e was defined to be untimed enabled if all

the rules that enable it are enabled (with exceptions for conflicts). An event e is defined to be timed enabled if all the rules that enable it are satisfied (with exceptions for conflicts). We now define a set R_{en} which contains all enabled rules, and a subset of R_{en} , R_s , which contains all satisfied rules. The set R_{en} defines an *untimed state* since it indicates which rules are enabled, but says nothing about timing. In order to determine which rules in R_{en} should also be in R_s , timing information is needed. How this timing information is represented depends on the specific timing analysis algorithm being used. We refer to an arbitrary set of timing information as TI . At a minimum, this information must contain how long each rule has been enabled. A *timed state* is defined to be R_{en} combined with the set of timing information ($TS = R_{en} \times TI$). A timed state contains all the information necessary to compute R_s .

All of the timed state space exploration algorithms presented here have the basic form shown in Figure 2. The algorithm simply does a depth-first search of the timed state space defined by the specification, and guided by the timing information, it finds all the timed states that are reachable. The *find_enabled_events* function uses timing information to determine what events should be included in the *event_list*, EL . Events are only added to EL if they can happen in the current timed state. An event is chosen from EL , and the current timed state and the rest of EL is pushed onto the stack. This event could be an action or it could simply be the advancement of time. If the event is an action, the *fire* function changes the set of enabled events (R_{en}) and the timing information. If the event is time advancing, only the timing information changes and R_{en} remains the same. The R_s set, however, may change due to the change in the timing information. When a new timed state is found, it is added to the state space, and a new list of enabled events is found. If a timed state is reached that has been reached before, the algorithm pops a timed state and the list of events that have not yet been explored for that state off the stack. When a state that has been seen before is reached and there are no unexplored events on the stack, the entire timed state space has been found.

With this algorithm, untimed states are only explored if they can be reached given the timing information in the specification. This can eliminate large portions of the untimed state space for most designs since many states reachable without timing information are not reachable given the timing constraints in the specification. However, the algorithm must explore the entire timed state space. The size of the timed state space depends on the representation chosen for the timing information. For example, if a continuous clock is associated with each rule in R_{en} , the timed state space would be infinite. If the timing information is represented more concisely, however, the timed state space that the algorithm explores may actually be smaller than the untimed

Algorithm 2.1 (Find timed states)

```

set_of_states find_timed_states(timed ER structure TERS){
    timed_state TS=initial_state(TERS);
    set_of_states S = {TS};
    event_list EL = find_enabled_events(TS, TERS);
    bool done=false;
    while (!done){
        event e=head(EL);
        push(TS, tail(EL));
        TS=fire(e, TS, TERS);
        if (TS ∉ S)then
            S = S ∪ {TS};
            EL=find_enabled_events(TS, TERS);
        else if (TS ∈ S) then
            if (stack is not empty) then (TS, EL)=pop();
            else done = true;
    }
    return S;
}

```

Figure 2. Timed state space exploration.

state space. Since less states may be explored, timed state space exploration is often faster than exploring the the untimed state space, as long as maintaining the timing information does not have an overwhelming cost. The algorithms presented in this paper address how to construct and maintain the timing information TI so that the set R_s can be constructed from every timed state. What this data is, and how it is used to calculate R_s depends on the specific timing analysis algorithm. Ideally, the algorithm only needs to retain a limited amount of timing information, so that the operations involved in maintaining the information are fast.

3. Overview of existing algorithms

There are a number of timing analysis algorithms that have been developed. They all work for the specific class of problems for which they were designed, but none of them is efficient for use in general timed state space exploration.

In [10], an algorithm is presented for finding the minimum and maximum time separations between events in acyclic graphs. It is $O(n^3)$ in the number of events in the graph. This algorithm can be used for timed state space exploration if the specification graph is finite and acyclic. However, most circuit specifications are cyclic.

In [12], a polynomial time algorithm is presented to compute an estimate of the minimum and maximum time separations between all events in a cyclic, conflict free graph. The algorithm works by unfolding the cyclic graph into an infinite acyclic graph and examining two finite acyclic subgraphs of the infinite graph to determine bounds on time separations between events. The estimate is usually sufficient for timed

state space analysis and can be improved by analyzing larger subgraphs. The algorithm is $O(v \cdot e)$ where v is the number of vertices and e is the number of edges in the subgraph analyzed. The conflict free restriction is too limiting, however, since most circuits need a choice type semantics to represent non-deterministic behavior in the environment.

CTSE [7] provides a way to find a single exact time separation between two events in a cyclic graph including limited types of conflict. However, each time CTSE finds a separation it reanalyzes the complete graph, so using it to compute all of the possible separations in a graph is slow. For example, in [7], CTSE takes 2 seconds to find one time separation in a Petri net with 21 transitions. To explore the entire state space it would be necessary to compute at least 21 separations and would take over 40 seconds. In the same paper, *Orbits*, a tool explicitly designed to compute the entire state space, takes 5 seconds to explore the entire state space. While CTSE is more efficient if only one separation needs to be computed, it is not appropriate for timed state space exploration. Also, while CTSE does handle choice, it is limited to unique, free and extended free choice [7].

Orbits [14] is specifically designed to do timed state space exploration and has been applied to timed circuit synthesis [13]. It uses *geometric regions* to keep track of the timing information relevant to the current marking of the graph and uses these regions to determine which events are timed enabled. Only local information about timing relationships is necessary to find the next set of timed states. This allows timing analysis to always be done on only a small subset of the events in the graph and is thus efficient, even for large graphs. Unfortunately, *Orbits* places some pretty severe restrictions on the types of graphs it can analyze. Each event can have only one rule, called a *behavioral rule*, that actually controls its firing time. In some cases, this *single behavioral rule restriction* can be worked around through transformations on the initial graphs [11], however, the transformations cause a large increase in the complexity of the graphs which need to be analyzed. For example, if an event originally has n behavioral rules, the graph is transformed to model the $n!$ possible orderings of the n enabling events.

Finally, in [19], a partial enumeration algorithm is presented to analyze a net with multiple behavioral rules. It is an interesting approach and the authors claim it is useful for validation, but it does not find the entire state space, and so cannot be applied to synthesis.

4. Geometric algorithms

There are many different ways that the timing information needed in the generic algorithm can be represented. The most obvious way would be to simply attach a continuously advancing clock to each of the enabled rules. This would,

however, result in an infinite timed state space. A slightly better representation would be to attach a clock to each of the enabled rules that advances only in discrete time steps. It has been shown that for our class of specifications that this is equivalent to the continuous model [6, 14]. This does make the state space finite, but it still explodes, especially if the delay ranges are large [15].

All of the timing analysis algorithms presented here are based on geometric regions. Geometric regions are a good way to concisely represent timing information [4, 8, 2, 14]. Large numbers of discrete timed states can often be condensed into a single contiguous geometric region that contains all of them, producing a large reduction in the number of timed states generated [14]. While worst case behavior of geometric timing is actually worse than the discrete method it has been shown that it can work well in practice [15, 13].

When using geometric regions for timing analysis in the generic algorithm, we define *TI* to be a constraint matrix M that specifies the maximum differences in time between the firings of the enabling events of all the rules in R_{en} . The 0^{th} row and column of the matrix contain the separations between each rule in R_{en} and a dummy rule r_\emptyset . The firing time of r_\emptyset is defined to be uniquely 0. Each entry m_{ij} in the matrix M has the value $\max(t(\text{firing}(j)) - t(\text{firing}(i)))$, which is the maximum time difference between the firing time of event j and the firing time of event i . Since the firing time of r_\emptyset is always zero, the maximum time difference between event i and event r_\emptyset (m_{0i}) is just the maximum time since i fired, and the maximum time difference between event r_\emptyset and event i (m_{i0}) is the negation of the minimum time since i fired. Note that M only needs to contain information on the timing of the rules that are currently in R_{en} , not on the whole set of rules. This particular way of representing timed regions was first introduced in [4]. This constraint matrix represents a convex $|R_{en}|$ dimensional region. Each dimension corresponds to a rule and the firing times of the enabled events for the rules can be anywhere within the space.

4.1. Geometric timing in *Orbits*

This representation of timed regions is used by *Orbits* to keep track of which rules are in R_s for a given timed state. In *Orbits*, the only action possible is the firing of an event, so every action actually changes the untimed state. An event is timed enabled when its behavioral rule is in R_s . When the timing information is stored in the geometric format, finding the list of timed enabled events is straightforward. The maximum time since the firing of each event is kept in the first row of the matrix. To look for timed enabled events, the algorithm can determine R_s by scanning the first row of the matrix and then looking for events with their behavioral rule in R_s .

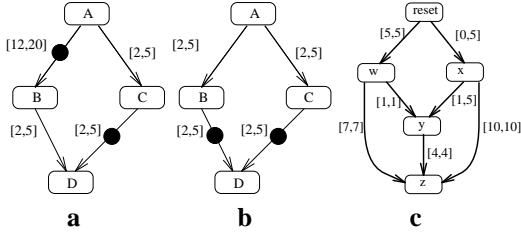


Figure 3. Some examples of ER structures.

When events fire or new rules are added to R_{en} , the matrix needs to be updated to reflect the new timing information. Information about the newly enabled rules must be added to the constraint matrix, and information about rules that are no longer in R_{en} must be removed. The main operation `Orbits` uses to do this is *recanonicalization*. Recanonicalization takes a matrix M where some of the m_{ij} 's are greater than $\max(t(\text{firing}(j)) - t(\text{firing}(i)))$ and produces a matrix where all the m_{ij} 's have their maximum allowed value. The assignment of the m_{ij} 's so that they all have their maximum value is always unique, so `Orbits` can determine when a given region is equivalent to or contained in a region that has been seen before. Recanonicalization is essentially the all pairs shortest path problem and can be done in $O(n^3)$ time with Floyd's algorithm. In the case of `Orbits`, it can in fact be done incrementally in $O(n^2)$ time, since most of the entries in the matrix already have their canonical value[14]. The procedure that `Orbits` uses for updating the matrix when an event is fired is dependent on the single behavioral rule restriction. We present new procedures for adding timing information to the matrix that allow the same recanonicalization method to be used without the single behavioral rule restriction in the next section. These algorithms retain the efficiency of the `Orbits` approach while eliminating the restriction.

4.2. Geometric timing in ATACS

The `Orbits` algorithm described above only works with the single behavioral rule restriction in which each event can only be constrained by a single rule. This assumption is too restrictive for most circuit specifications. For example, a simple **and** gate could not be represented with this restriction since either input can constrain the time when the output changes. The restriction can be worked around with graph transformations as mentioned earlier, but this can cause a substantial increase in graph size. In this section, we present an extension to the `Orbits` algorithm that provides for multiple behavioral rules. If multiple behavioral rules are allowed, it is possible for a rule to become expired. The timed ER structure in Figure 3(a) shows an example of this. The rule $\langle C, D, 2, 5 \rangle$ has to remain enabled at least 9 time units before $\langle B, D, 2, 5 \rangle$ is satisfied and allows event D to fire. If

Algorithm 4.1 (Fire a rule)

void fire_rule(rule $\langle e, f, l, u \rangle$, constraint matrix M ,

rule set R_f, R_{en}, R_s) {

$M[\text{index}(\langle e, f, l, u \rangle)[0] = -l$;

recanonicalize(M);

project($M, \text{index}(\langle e, f, l, u \rangle)$);

$R_f = R_f \cup \langle e, f, l, u \rangle$;

$R_{en} = R_{en} - \langle e, f, l, u \rangle$;

$R_s = R_s - \langle e, f, l, u \rangle$;

if ($\forall \langle e_i, f, l_i, u_i \rangle \in R : ((r_i \in R_f) \vee$

$(\exists r_j = \langle e_j, f, l_j, u_j \rangle \in R_f \text{ s.t. } e_i \# e_j)))$

$R_{en} = R_{en} - \{ \langle e_i, f, l_i, u_i \rangle \in R_{en} : (f_i \# f) \}$;

$R_f = R_f - \{ \langle e_i, f, l_i, u_i \rangle \in R_f : f_i = f \vee f_i \# f \}$;

$R_{en} = R_{en} \cup \{ \langle e_i, f, l_i, u_i \rangle \in R : e_i = f \}$;

Add new rows and columns to matrix M for new members of R_{en} ;

Advance time;

recanonicalize(M);

}

Figure 4. Procedure for firing a rule.

rules are allowed to expire, non-convex regions can be produced. Since Floyd's algorithm only works on convex regions, this must be avoided. One way to eliminate the single behavioral rule restriction and avoid generating non-convex regions is to change the timing semantics of the specification so that rules are never allowed to exceed their maximum bounds (i.e., type 1 semantics [18]). In type 1 semantics, the specification is invalid if a rule cannot fire within its timing range. In real circuits, however, rules typically are allowed to exceed their maximum bounds (type 2 semantics[18]).

Our geometric timing algorithm eliminates the single behavioral rule restriction and allows timing analysis of specifications in which rules can expire. In the `Orbits` algorithm, the timing information is only updated when an event fires. The single behavioral rule restriction can be eliminated if timing information is updated whenever a rule fires and rules are allowed to fire independently of events. A rule can always fire when it is satisfied. The firing of a rule, however, does not always correspond to the firing of an actual event. An event only fires when all of the rules enabling it have fired. As rules fire, they are projected out of the constraint matrix, and are removed from both R_s and R_{en} . They are added to a new set of "fired" rules, R_f , which is part of the timing information. Since they have fired, timing information about them is no longer needed, but the fact that they have fired must be recorded. When a set of rules sufficient to enable an event e are in R_f , e can fire.

This new method for updating the timing information requires a change in the `Orbits` algorithm. When using the `Orbits` algorithm, the *find_enabled_events* function in Al-

gorithm 2.1 computes a list of events to fire. In the new algorithm, instead of determining a list of timed enabled events to fire, the function computes a list of satisfied rules to fire. The list of rules that are satisfied is simply R_s which can be calculated easily by scanning the first row of the constraint matrix which contains the maximum time since the enabling of all the rules in R_{en} .

The firing of actual events is handled within the *fire_rule* routine specified in Figure 4. This function takes as input the rule chosen to fire, the constraint matrix, and the corresponding rule sets. The *index* function used in the algorithm takes a rule, and returns its index in the constraint matrix. The first step of the function sets the minimum time since the enabling of the firing rule to be its lower bound since in order to fire, it must have been enabled as long as its lower bound. The matrix is then recanonicalized to produce a new region that is constrained by this firing time. The timing information for this rule is then removed from the matrix by the *project* operation. Projection simply removes the row and column corresponding to this rule from the matrix. This step is what allows the size of the constraint matrix to remain $|R_{en} + 1|$ instead of growing with the size of the specification. The rule is also added to R_f and removed from R_{en} and R_s . Next, the algorithm checks if firing this rule has caused any events to be fired. An event is fired if all of the rules that enable it are either in R_f or conflict with another rule that is in R_f . If it has not, the algorithm is done. If it has, the algorithm removes from R_{en} and R_f any rules with enabled events which conflict with the event that fired. The algorithm also removes from R_f any rules that enabled the firing event, and adds to R_{en} any rules enabled by the firing of the new event. Timing information on the newly enabled rules is then added to the matrix. When a rule is initially enabled, no time has passed since its enabling, so the entries in the matrix for the minimum and maximum times since its enabling are set to zero. The maximum difference between the enabling time of a newly added rule and any old rule is just the maximum time since the enabling of the old rule. Therefore, the new row of the matrix is set to equal the 0th row. The minimum difference between the enabling times of a new rule and an old rule is the minimum time since the enabling of the old rule, so the new column is set to the 0th column. Finally, in the *advance time* step, the maximums in the 0th row are set to their maximum specified value (the upper bounds on the rules), and the matrix is recanonicalized. We now have a constraint matrix representing the region of possible firing times for the rules in the updated R_{en} set.

Figure 5 shows an example of how Algorithm 4.1 would be applied to a simple ER structure. The ER structure, with its enabled rules marked with tokens, is shown in the first column, the constraint matrix is shown in the second column, and the contents of the R_{en} , R_s , and R_f sets are shown in the third column. Initially, rules $\langle A, C \rangle$ and

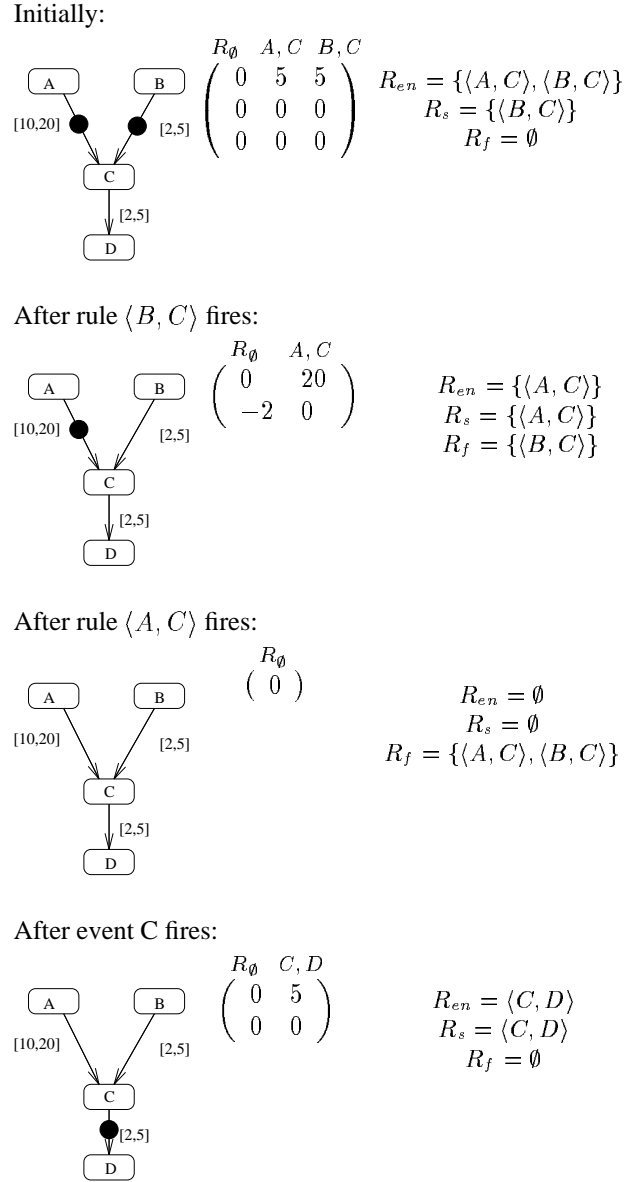


Figure 5. Firing rules.

$\langle B, C \rangle$ are in R_{en} . The constraint matrix indicates that the maximum time since both these rules were enabled is 5. Since the lower timing bound on $\langle A, C \rangle$ is 10, it is not timed enabled and therefore not in R_s . The lower bound on $\langle B, C \rangle$ is 2, which is less than 5, so it is timed enabled and is in R_s . The rule $\langle B, C \rangle$ then fires. It is added to the R_f set and its timing information is projected out of the constraint matrix. No events are enabled to fire, so no new rules are enabled. After the firing of $\langle B, C \rangle$, the constraint matrix indicates that the maximum time since the firing of $\langle A, C \rangle$ is 20. This is larger than the lower timing bound on $\langle A, C \rangle$, so it can be added to R_s . The rule $\langle A, C \rangle$ then fires. Its timing information is projected out of the constraint matrix, and

the rule is added to R_f . The R_f set is then sufficient to enable an event, C , which causes the rule it enables, $\langle C, D \rangle$, to be added to the R_{en} set. The recanonicalization procedure that produces the values in the constraint matrix is not shown here, but is described in detail in [14].

This algorithm allows us to analyze any timed ER structure including those with multiple behavioral rules and arbitrary conflicts. It can, however, generate a larger number of regions to be explored than `Orbits` since there are more rules than events in a given ER structure. The next section introduces methods to reduce the number of regions.

5. Partial order algorithms

The number of explored regions can be reduced significantly by using partial order techniques [15]. Partial order techniques take advantage of the inherent concurrency in the ER structure and prevent additional regions from being added for different sequences of event firings that lead to the same untimed state. For example, in Figure 3(b) it does not matter whether the firing sequence is $[A, B, C]$ or $[A, C, B]$, the untimed state of the system is the same after both firing sequences. The geometric techniques presented in the previous section produce a different region, and therefore a different timed state, for each firing sequence. With partial order techniques, these regions can be combined so that one region is generated that encompasses all the timed states possible after all firing sequences.

5.1. Partial orders in `Orbits`

`Orbits` [14] provides an algorithm to eliminate extraneous regions by keeping a *process matrix* in addition to the constraint matrix. A *process* is an acyclic, choice-free, graph created from an ER structure and a firing sequence. It is constructed from the ER structure as follows: the process initially contains all of the events whose enabling rules are initially in R_{en} . Events are added in the same order as they occur in the firing sequence. For an event in the firing sequence, a correspondingly labeled event is added to the process. Rules are added to connect the newly enabled event to the events in the process that enabled it.

The process represents the concurrency in a particular firing sequence. One process can correspond to a number of different firing sequences that only differ in the order of firings of concurrent events. All the firing sequences in one process lead to the same untimed state. The process matrix stores the minimum and maximum possible separations of all the events in the process defined by the current firing sequence. At each iteration, the separations in the process matrix are copied into the entries of the constraint matrix that restrict the differences in the enabling times of the rules. With

the restriction `Orbits` places on the semantics, the calculations to maintain the process matrix can be done with a simple $O(n^2)$ application of Floyd's algorithm. Events can also be projected out of the process matrix just like they are projected out of the constraint matrix, so the algorithm only needs to retain and operate on local timing information. This is a big advantage when dealing with large specifications. However, this technique as presented in `Orbits` does not work in the general case since rules may stay enabled for a time greater than their maximum bounds.

5.2. Approximate Partial Orders

We have modified the partial order algorithm presented in [14] to produce an algorithm that finds an approximation of the timed state space for type 2 specifications. Floyd's algorithm is still used to recanonicalize the matrix, but the way that the timing information about new events is added to the matrix is changed. It is shown in a later section that this algorithm always finds a timed state space that contains the exact timed state space. Extra timed states may be included, causing the circuit to be non-optimal, but a circuit synthesized from the approximate state space is always a correct implementation of the specification.

When specifications have the single behavioral rule restriction, it is simple to determine the minimum and maximum separations between the firing time of an event e and the event that enabled it, e_i . Since no other rule can constrain the firing of e , the minimum separation is the lower bound on the rule that relates e and e_i and the maximum separation is the upper bound. When a new event fires and is added to the process matrix, the minimum and maximum time separations between its firing time and the firing times of all other events in the matrix must be determined. With the single behavioral rule restriction, the minimum and maximum separations between the new event, and all events that enable it are immediately known from the lower and upper bounds on the rules. These separations can be entered directly in the matrix. The separations between the new event and events that do not enable it are unknown, and their value is set to infinity. The recanonicalization operation constrains those unknown separations to their correct value [14].

If the specification can have multiple behavioral rules, the minimum and maximum separations between the firing time of an event e and the event that enabled it e_i are not immediately known. They are not simply the time bounds on the rule that relates the events, since another rule may delay the firing of e . For example, in Figure 5, the maximum separation between the firing times of event B and event C is 20 even though the maximum time bound on the rule that relates them is 5. This is the case because the firing of C can be delayed until time 20 by the rule $\langle A, C \rangle$.

In this algorithm, when calculating the separations

Algorithm 5.1 (Add an event to the process matrix)

```

void add_event(process P, new event e){
  foreach  $e_i \in P$ 
  if( $\exists \langle e_j, f_j, l_j, u_j \rangle \in P$  s.t.  $((e_j = e_i) \wedge (f_j = e))$ ) then
     $max\_sep(e_i, e) = \max(u_j, \max_{\langle e_k, f_k, l_k, u_k \rangle \in P \text{ s.t. } (f_k = e)} (max\_sep(e_j, e_k) + u_k))$ ;
     $min\_sep(e_i, e) = \max(l_j, \max_{\langle e_k, f_k, l_k, u_k \rangle \in P \text{ s.t. } (f_k = e)} (min\_sep(e_j, e_k) + l_k))$ ;
  else
     $max\_sep(e_i, e) = \infty$ ;
     $min\_sep(e_i, e) = -\infty$ ;
  recanonicalize(process matrix for P);
}

```

Figure 6. Finding separations to add an event.

between the firing time of an event e and the event that enabled it, e_j , the fact that the firing of e may be delayed by the other rules that enable it is accounted for. The maximum separation is originally set to the value of the upper bound on the rule $\langle e_j, e, l_j, u_j \rangle$, since the separation must have at least that value. Then the algorithm checks if there are any other rules that could delay the firing of e beyond that upper bound. This is done by examining the maximum separations between all other events that enable e and the event e_j . If there is an event e_k that enables e with the rule $\langle e_k, e, l_k, u_k \rangle$, and the firing time of e_k is at most t time units after e_i then e_k delays the firing of e if $t + u_k \geq u_j$. If e_k delays the firing of e then the value $t + u_k$ is used as the maximum separation between the firing times of e and e_j .

Figure 6 shows the algorithm for adding a new event to the process matrix. Placement of the time separation values in the process matrix is not shown in the algorithm. References to the min_sep or max_sep of two events are really references to process matrix entries and do not require any computation. The $min_sep(e_i, e_j)$ is simply $-\max_sep(e_j, e_i)$, and all of the separations in the actual matrix are interpreted as maximum separations between the column and row event. All of the new separations are computed from only local information about the timing of the events that enable them. If a new event e is enabled by an event e_j , then the maximum separation between them is at least the upper bound on the rule that relates them (u_j). However, due to the type 2 semantics it may be more than that. Another event that enables e may delay the firing of e beyond the maximum bound on the rule that relates e_j and e . This is the case if there is some other event e_k that also enables e with an upper bound u_k and $\max_sep(e_k, e_j) + u_k > u_j$. The minimum separation between e and e_j is at least the lower bound on the rule that relates them (l_j). But again, due to the type 2 semantics, it may be more than that. If another rule always delays the firing of e the minimum is greater than

l_j . If there is no rule relating e and e_j then maximum separation is set to ∞ and the minimum is set to $-\infty$. These separations are set to their maximum and minimum possible values given the constraints imposed by the events that do have causal relationships when the matrix is recanonicalized [14].

Since only local timing information is necessary to determine the timing of new events, when all the events enabled by an event e have fired and been added to the process matrix, e can be projected out. This keeps the process matrix small and the computations fast. However, the resulting timed state space is not always exact. For example, when the approximate algorithm is run on the ER structure in Figure 3(c) the minimum separation between event y and event z is found to be 4 while the actual minimum is 5. This causes a region to be generated that is larger than necessary, but the circuit synthesized using this region is still correct. The reason for this imprecise result is explained later.

5.3. MaxDiff

One way to guarantee finding the exact timed state space is to use a more general timing analysis algorithm than Floyd's to analyze the acyclic process graphs in order to determine the separations for the process matrix. Since the process is acyclic, an algorithm for acyclic graphs such as the ones in [10, 11] can be used to maintain the process matrix. The separations between all the events in the current unwound portion of the specification can be computed, and then those that are needed can be copied into the constraint matrix. Note that the process matrix contains separations between events, and the constraint matrix contains separations between rules. However, the time separation between two rules is simply the time separation between the two events that enabled them.

We use the MaxDiff algorithm, presented in [11], to maintain the process matrix. This algorithm is similar to the acyclic algorithm presented in [10] discussed earlier. When using the MaxDiff algorithm to maintain the process matrix, every time a new event is added to the process, the algorithm is used to compute the maximum separation between this new event and all the other events in the process. Information about the separations of previous events is stored in the process matrix, and as soon a known value is found, the algorithm can end the recursion and begin returning the answer. However, calling the MaxDiff requires recursing back to the initial event in the process. This means that the process matrix for the entire firing sequence must be kept around to use this algorithm. As the process gets bigger, the time it takes to compute each separation between a new event and the older events grows, as does the size of the matrix. At the same time, more separations must be computed, since the number of old events grows as well. If the number of necessary unfoldings of the specification is small, this al-

gorithm works well. But if many unfoldings are required, the overhead to maintain the process matrix becomes excessive, and another approach is needed.

5.4. Consistency

Algorithms such as MaxDiff that calculate separations between events with type 2 semantics need to analyze the entire process graph instead of just basing the timing of each new event on the timing of the events that enable it. This is necessary because the set of separations that has already been computed may not be *consistent*. A set of separations is consistent if there is a possible timing assignment that would cause all the separations to have their maximum value. With type 1 semantics (or the single behavioral rule restriction) the separations in the process matrix are always consistent. When type 2 semantics is allowed, this may not be the case as illustrated with the example shown in Figure 3(c). The maximum separation between w and y is 5 and the maximum separation between x and y is 6. However there is no one timing assignment that allows both of these maximums to occur. For the separation between w and y to be 5, x must fire at time 5, and for the separation between x and y to be 6, x must fire at time 0. Clearly, this cannot happen in one timing assignment. If these separations are used by themselves to determine the separations between following transitions, the separations calculated may be impossible to achieve and therefore greater than their real value. In this example the minimum separation between the firing of y and the firing of z would be found to be 4. The actual minimum is 5 since $sep(w, y)$ and $sep(x, y)$ cannot achieve their maximum separations in the same firing sequence. This distinction is subtle and led to at least one erroneous timing analysis algorithm to be developed [18].

However, most of the time, inconsistencies can be projected out of the process matrix before they can effect the timing of later events. Inconsistencies are only generated when an event is enabled by a group of preceding events. Usually, this event is then used in the rest of the ER structure to indicate that all of the preceding events have occurred, and the preceding events are no longer used. If this is the case, the preceding events are projected out of the process matrix as soon as the event they have enabled has fired. This means that the inconsistencies that are generated are projected out as soon as they are created. The approximate algorithm presented previously used this observation. If all inconsistent sets are projected out of the process matrix as soon as they are generated, then the approximate algorithm generates the exact timed state space of the specification. If there are inconsistencies that do get used in computation, more of the state space is explored since the regions generated are larger than necessary. In this case, the circuit generated from the approximate state space is still correct, al-

though possibly not optimal. If the circuit is highly dependent on timing, no implementation may now be possible.

If an inconsistent set of separations is used to compute the minimum and maximum separations for a new event, imprecise results occur. However the regions generated are always larger than the actual set of regions. Suppose that the set of separations between event e and all the other events in the process e_i is not consistent. Each separation between two events in the process matrix is the maximum value that that separation can achieve over all possible timing assignments. In an actual timing assignment, the maximum value for some separations may be lower, but never higher. Since $min_sep(e_i, e_j)$ is $-\max_sep(e_j, e_i)$ this means that actual minimum separations may be higher, but never lower. Therefore, when the algorithm computes new separations using an inconsistent set, it may be using maximums that are too large, and minimums that are too small, but never the opposite. Using maximum separations that are too large and minimums that are too small in the computation when the algorithm calculates the separations for a new event can only produce maximums that are larger and minimums that are smaller than their actual value. Therefore, the regions produced always contain the actual regions. Also, the approximate algorithm can record the number of inconsistencies it has encountered, so it is known when the algorithm completes whether the solution is exact.

5.5. Consistent Sets

The algorithm presented in this section is an adaptation of the approximate algorithm that looks for inconsistencies and keeps track of a number of consistent sets of separations when they occur. The maximum over all the sets is actually copied into the constraint matrix. This algorithm preserves the ability to project timing information out of the process matrix while still producing the exact timed state space.

The first step is figuring out when an inconsistent set has been created and is actually going to be used in later computation. The *add_event* algorithm presented in Figure 6 is used to generate an initial set of separations. As the algorithm is running it also keeps track of which separation in the matrix each new separation *depends* on. A separation $\max_sep(e_j, e)$ depends on $\max_sep(e_k, e_j)$ if $\max_sep(e_j, e)$ as computed in *add_event* is equal to $\max_sep(e_k, e_j) + u_j$. Dependence for minimums is defined symmetrically. This means that each new separation *depends* on one previous separation that actually caused it to reach its maximum value. When the initial computation is done, the list of dependencies is checked for *inconsistencies*. An inconsistency occurs when there are two maximums or minimums in the computed set that depend on the same separation having different values. Since the matrix only stores maximums and minimums, inconsistencies can

only occur when one new maximum depends on a previous separation having its maximum value and another new maximum depends on the same separation having its minimum value. The situation for new minimums is symmetric. Note that a new minimum and a new maximum can depend on a separation having different values since the set of new maximums and the set of new minimums do not need to be generated by the same timing assignment. If there are no inconsistencies, then the set of new maximums and minimums is consistent and the algorithm is done. If there are inconsistencies, then they are checked to see if they only involve events that are going to be immediately projected out of the process matrix due to the firing of the new event. If they are, then they do not cause inaccuracies, and no further computation is needed. If the inconsistencies are not immediately projected out, additional sets of maximums and minimums must be generated to maintain consistency.

When a new event is added to the process and it generates inconsistencies, multiple sets of minimum and maximum separations between it and all the previous events still in the process matrix need to be generated. Each individual set of separations can be generated by a possible timing assignment. New sets are created by *resolving* inconsistencies. An inconsistency is resolved by computing two new sets of minimums and maximums. One set of separations contains the values possible if the inconsistent separation always had its maximum value and the other set of separations contains the values possible if the inconsistent separation always had its minimum value. Both new sets are added to the process matrix, and the original, inconsistent set is removed.

A new data structure, called the *resolution_array*, keeps track of which way inconsistencies are resolved in each computed set. When a new event is added, multiple sets of minimums and maximums may need to be calculated. A new set needs to be calculated for every possible, *valid* combination of the sets generated for every event in the matrix. A combination of sets is invalid if the resolution array specifies the same constraint was resolved in different ways for a pair of sets in the combination. Therefore, the number of sets generated when a new event is added is dependent on the total number of conflicts that have been resolved. This dependence is actually exponential in the number of resolved inconsistencies, but since events and hence inconsistencies are projected out of the matrix as they are no longer needed, for typical specifications this is not a problem. Clearly it is possible to create examples that would cause the number of sets to explode, but in that case, the MaxDiff or ordinary geometric algorithm should be used instead. The number of inconsistencies in a specification can be computed without running the entire consistent sets algorithm by simply keeping track of them as the approximate algorithm is running.

Figure 7 gives an overview of the consistent sets al-

gorithm. The details about how all the sets are kept track of in the process matrix are not shown. Calls to *add_event* refer to the *add_event* function defined in Figure 6, with the addition that it maintains a list of inconsistencies as it calculates a set of separations. When a call to *add_event* is made, it is assumed that the resulting separations are placed in the final process matrix in the correct row and column. In the consistent sets algorithm, when a new event is to be added to the process, the function *add_new_event* is called. It causes a new set of separations to be calculated for each valid combination of sets already in the process matrix. A new set inherits all the resolution decisions from the sets in the combination used to compute it. For each new set of separations, *add_new_event* calls *make_consistent* to resolve any inconsistencies that may have been generated. The recursive calls to *make_consistent* are necessary if multiple conflicts need to be resolved in one set.

Algorithm 5.2 (Add event with consistent sets)

```

process_matrix add_new_event(process_matrix P, new event e,
                             resolution array R){
    foreach(valid combination of sets in P){
        I=add_event(P, e);
        Add entry to R for this set that contains the union
            of all the entries for the sets in this combination;
        Pnew=make_consistent(e,P,I,R);
        return Pnew;
    }
}

```

Algorithm 5.3 (Resolve all the inconsistencies in a set)

```

process_matrix make_consistent(event e, process_matrix P,
                              inconsistency List I, resolution array R){
    if (no inconsistencies, or all are projected immediately)
        return;
    Select an inconsistency i from I;
    if (i resolved by entry in R for this set){
        Remove inconsistent set from P;
        Pnew = P resolved as specified by R;
        I=add_event(Pnew,e);
        Pnew=make_consistent(e, Pnew, I, R);
        return Pnew;
    } else{
        Remove inconsistent set from P;
        Pnew = P with i resolved with min;
        I=add_event(Pnew,e);
        Add resolution to R;
        Pnew=make_consistent(e, Pnew, I, R);
        Pnew = P with i resolved with max;
        I=add_event(Pnew,e);
        Add resolution to R;
        Pnew=make_consistent(e, Pnew, I, R);
        return Pnew;
    }
}

```

Figure 7. Consistent sets algorithms.

Table 1. Experimental results. Time values are given in seconds.

Examples	SI		Geometric only regions time		Partial order methods					
	Untimed states	Untimed states			MaxDiff regions time		Approx regions time		Con Sets regions time	
scsiSVT	20	15	56	0.31	33	0.21	33	0.20	33	0.17
spdor	185	18	40	0.18	40	0.22	40	0.19	40	0.17
spdand	88	10	134	0.48	64	0.31	fail		64	0.40
mmuoptSV	397	108	8083	397	806	41	806	30	806	28
slatch	54	30	151	0.97	65	0.77	65	0.67	65	0.52
JSPslatch	54	30	150	1.29	65	0.67	65	0.58	65	0.63
SELOpt	351	113	657	17	308	18	308	12	308	11
TSBM	5832	403	out of memory		1946	1096	1946	80	1946	77

Figure 8 shows the process matrix and resolution array generated for the ER structure in Figure 3(c). Each entry in the matrix is the maximum amount of time that can pass between the firing time of the column event and the firing time of the row event. For example, the maximum separation between the firing times of event x and event y_1 is 5. The subscript indicates that multiple sets were added for this event. The X symbol in the matrix indicates that separation would be derived from an invalid combination of sets, and therefore is not computed. The resolution array shows how inconsistencies were resolved to create each set. When separations from this process matrix are copied into the constraint matrix, the maximum value over all sets for each separation is used. For example, the value copied for the $sep(y, x)$ would be 6 and the value copied for $sep(y, z)$ would be -5.

$$\begin{array}{c}
 w \quad x \quad y_1 \quad y_2 \quad z_1 \quad z_2 \\
 \begin{pmatrix}
 w & 0 & 5 & -1 & -1 & -7 & -7 \\
 x & 0 & 0 & -1 & -6 & -5 & -12 \\
 y_1 & 5 & 5 & 0 & 0 & -5 & X \\
 y_2 & 1 & 6 & 0 & 0 & X & -6 \\
 z_1 & 10 & 10 & 9 & X & 0 & 0 \\
 z_2 & 7 & 12 & X & 9 & 0 & 0
 \end{pmatrix} \\
 \begin{array}{c}
 w \\
 x \\
 y_1 \\
 y_2 \\
 z_1 \\
 z_2
 \end{array}
 \begin{pmatrix}
 \emptyset \\
 \emptyset \\
 sep(x, w) = 0 \\
 sep(x, w) = -5 \\
 sep(x, w) = 0 \\
 sep(x, w) = -5
 \end{pmatrix}
 \end{array}$$

Figure 8. Proc. matrix and resolution array.

The consistent sets algorithm provides a way to compute the exact timed state space of a timed ER structure without maintaining a process matrix that grows with the size of the specification and the number of unfoldings needed. The size of the matrix grows only with the maximum number of events that can be active at any one time and the number of

inconsistencies. If the number of inconsistencies is low, this algorithm is very efficient. If it is high, one of the other algorithms can be used.

6. Results

We have implemented each of the timing analysis algorithms described in this paper within the timed circuit design tool ATACS, and we have applied them to several examples as shown in Table 1. The first two columns compare the number of untimed states found, first neglecting the timing information then using it. We see that the state space can be reduced by up to an order of magnitude using timing information. The rest of the table compares the number of regions found and runtimes for each of the algorithms presented. The runtimes are reported in seconds on a 75 MHz Sparc20 with 128 Mbytes of memory. The results show that partial order information can reduce the number of regions and runtimes by up to an order of magnitude. Indeed, in one example, TSBM, geometric timing without partial orders runs out of memory. Furthermore, the approximate and consistent set partial order algorithms can yield an order of magnitude improvement over the MaxDiff partial order algorithm. Finally, we observe in one example, **spdand**, that the approximate partial order algorithm fails because it generates regions which are too large and explore numerous extra states. This occurs because **spdand** has inconsistencies and is highly timing dependent for its correctness.

These results show that each of the algorithms presented here is appropriate under different circumstances. If the amount of concurrency in the specification is low, then the geometric timing algorithm can efficiently explore the state space with a minimum of overhead. If concurrency is high, but the specification produces only consistent sets of maximums or the resulting circuit does not need to be optimal, the approximate partial order algorithm can reduce the num-

ber of regions explored and not increase overhead significantly. In the course of running the approximate algorithm, the number of inconsistencies can be computed to give the user guidance as to whether the consistent sets or MaxDiff algorithm could be used to get a more optimal circuit. The consistent sets or MaxDiff algorithms can be used to compute the exact state space if an optimal circuit is needed for a specification that produces inconsistent sets of maximums. If the circuit produces a large number of inconsistencies, then the MaxDiff algorithm is most appropriate. If the specification only produces a small number of inconsistencies, then the consistent sets algorithm is more efficient than the MaxDiff algorithm. Together these algorithms allow designers to choose tradeoffs between circuit performance and synthesis time that meet their needs.

7. Conclusion

We have presented a group of timing analysis algorithms specifically designed for timed state space exploration. Doing timing analysis based on the firing of rules instead of events allows us to analyze timed ER structures with type 2 semantics and eliminate the restrictions placed on the semantics by previous algorithms. The partial order techniques presented here reduce the number of regions generated by the basic geometric algorithm. The approximate partial order algorithm and consistent sets algorithms allow partial order techniques to be applied while maintaining only local timing information. In the future, we plan to extend the algorithms to analyze specifications with OR causality and explore BDD techniques to create a more efficient representation of the timed state space. We also plan on applying the algorithms to optimizing synthesized logic.

8 Acknowledgments

We would like to thank Dr. Steve Burns of Intel Corporation, Dr. Tomas Rokicki of Hewlett Packard, and Robert Thacker of the University of Utah for their helpful comments and encouragement.

References

- [1] W. Belluomini. Transistor level optimizations of asynchronous circuits. Master's thesis, University of Washington, 1996.
- [2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time petri nets. *IEEE Transactions on Software Engineering*, 17(3), March 1991.
- [3] B. Coates, A. Davis, and K. Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341–366, Oct. 1993.
- [4] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the Workshop on Automatic Verification Methods for Finite-State Systems*, June 1989.
- [5] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In *VLSI '93*, 1993.
- [6] T. A. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP 92: Automata, Languages, and Programming*, pages 545–547. Springer-Verlag, 1992.
- [7] H. Hulgaard and S. Burns. Bounded delay timing analysis of a class of CSP programs with choice. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 2–11, November 1994.
- [8] H. R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical report, Harvard University, July 1989.
- [9] A. Marshall, B. Coates, and P. Siegel. Designing an asynchronous communications chip. *IEEE Design & Test of Computers*, 11(2):8–21, 1994.
- [10] K. McMillan and D. L. Dill. Algorithms for interface timing verification. In *International Conference on Computer Design, ICCD-1992*. IEEE Computer Society Press, 1992.
- [11] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.
- [12] C. J. Myers and T. H.-Y. Meng. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems*, 1(2):106–119, June 1993.
- [13] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis of gate-level timed circuits with choice. In *Proc. 16th Conf. on Advanced Research in VLSI*, pages 42–58. IEEE Computer Society Press, 1995.
- [14] T. G. Rokicki. *Representing and Modeling Circuits*. PhD thesis, Stanford University, 1993.
- [15] T. G. Rokicki and C. J. Myers. Automatic verification of timed circuits. In *International Conference on Computer-Aided Verification*, pages 468–480. Springer-Verlag, 1994.
- [16] J. A. Tierno, A. J. Martin, D. Borkovic, and T. K. Lee. A 100-MIPS GaAs asynchronous microprocessor. *IEEE Design & Test of Computers*, 11(2):43–49, 1994.
- [17] C. K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Saeijs. A fully-asynchronous low-power error corrector for the digital compact cassette player. In *IEEE International Solid-State Circuits Conference*, 1994.
- [18] P. Vanbekbergen, G. Goossens, and H. de Man. Specification and analysis of timing constraints in signal transition graphs. In *Proceedings of the European Design Automation Conference*, 1992.
- [19] E. Verlind, G. de Jong, and B. Lin. Efficient timing analysis of highly concurrent systems. In *International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, November 1995.
- [20] G. Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. Noordwijkerhout, Norway, June 1988.
- [21] H. Zheng and C. J. Myers. Specification and compilation of mixed-timed systems using vhdl. forthcoming paper.