Temporal Analysis of Time Bounded Digital Systems

Alan R. Martello and Steven P. Levitan

University of Pittsburgh, Department of Electrical Engineering, Pittsburgh, PA 15261

Abstract. To perform verification of digital systems with time bounded delays, it is essential to characterize the space of all possible system behaviors. In this paper, we describe our analysis technique which accepts a behavioral specification of the timing of a digital system and generates the set of all possible behaviors for the system. The ability to represent and reason about time ranges for events is a distinguishing characteristic of our technique and gives our analysis method both its power and complexity.

1 Introduction

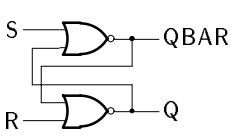
Our research focuses on the analysis of digital system descriptions which include time bounded delays. This analysis is part of the problem of verification: given behavioral specifications of a system, check that no possible behavior of the system would violate any requirement constraints imposed on the system. The requirements may come either from the designer or from the needs of the components (e.g. setup and hold constraints) which make up the system. In general, our model of digital systems with time bounded delays allows for the specification of digital systems, the generation of all possible system behaviors given the specification and an initial system state, and the analysis of the resulting system behaviors.

In this paper we focus on the key part of this analysis, the generation of all possible system behaviors from a system behavioral specification. We are primarily concerned with the temporal behavior of digital systems, not their function. We are only concerned with "data values" to the extent that one signal could enable or disable the occurrence of an event on another signal in the system. The ability to represent and reason about time ranges for events is a distinguishing characteristic of our technique and gives our analysis both its power and complexity.

Attempts to reason about digital systems have been ongoing [1–3]. Various methods for modeling the systems with time delay have been proposed and formal verification methods investigated[4–7]. Methods based on simulation have also been used to verify circuits[8,9]. Other recent work has taken a symbolic approach to attempt to reason about circuits[10,11] and our earlier work[12, 13] investigated a more simplistic verification technique upon which this work is based. Recent work in asynchronous verification [14] performs an analysis similar to our search process and also depends on the need to perform bifurcations (or splits) as system behaviors are generated.

2 Overview & Example

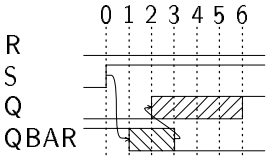
To provide an overview of the our analysis technique, we present an example of an RS latch constructed from two nor gates, each with gate delay in the range 1 to 3 (Fig. 1(a)), and with the system behavioral specification shown in Fig. 1(b). The RS latch starts with R, S, and Q low and with QBAR high. A rising transition occurs on S at $t = 0$. Figure 1(c) illustrates the timing waveform for this input transition of the latch.



(a)

R↑ **when** ¬QBAR → Q↓ (1,3)
R↓ **when** ¬QBAR → Q↑ (1,3)
QBAR↑ **when** ¬R → Q↓ (1,3)
QBAR↓ **when** ¬R → Q↑ (1,3)
Q↑ **when** R∨QBAR →
  Q↓ **when** R∨QBAR (1,1)
Q↓ **when** ¬R∧¬QBAR →
  Q↑ **when** ¬R∧¬QBAR (1,1)
S↑ **when** ¬Q → QBAR↓ (1,3)
S↓ **when** ¬Q → QBAR↑ (1,3)
Q↑ **when** ¬S → QBAR↓ (1,3)
Q↓ **when** ¬S → QBAR↑ (1,3)
QBAR↑ **when** S∨Q →
  QBAR↓ **when** S∨Q (1,1)
QBAR↓ **when** ¬S∧¬Q →
  QBAR↑ **when** ¬S∧¬Q (1,1)

(b)



(c)

Fig. 1. RS Latch (a), behavioral description (causalities) (b), and timing waveform (c)

If we were doing verification of the latch, we would be checking this waveform against a system requirement like "Q must rise within 5 of S rising." This requirement may be imposed by other modules in the system or by external constraints on the system. In order to perform verification, all system requirements are compared with all system behaviors generated, to verify proper system operation. The most difficult part of this verification process is the generation of all possible system behaviors.

In order to generate all possible system behaviors for digital systems with time bounded delays, we must enumerate (or search) the space of all possible interactions among signals which often have mutual interdependencies on their transitions. The search space is bounded by identifying system states which have previously occurred, effectively handling cycles and reconvergent behaviors. Conversely, the size of the search space grows based on the interaction among signals and the time ranges associated with their interdependent transitions.

As an example of such a search space explosion, we revisit the RS latch previously shown. The gates which comprise the latch in this example have gate delays

of 1 to 3 and both inputs fall from the "illegal input state" at $t = 1$. The four behaviors shown in Fig. 2 are representative of the 26 behaviors resulting from our analysis of 855 system states (taking 16.90 cpu seconds on a Sparcstation 2). The many behaviors explored are caused by the splitting or bifurcation of nodes in the search space of possible behaviors. From one starting state and initial transitions of R and S the search space grows, caused by alternative interactions based on system state and dissection of the time ranges.

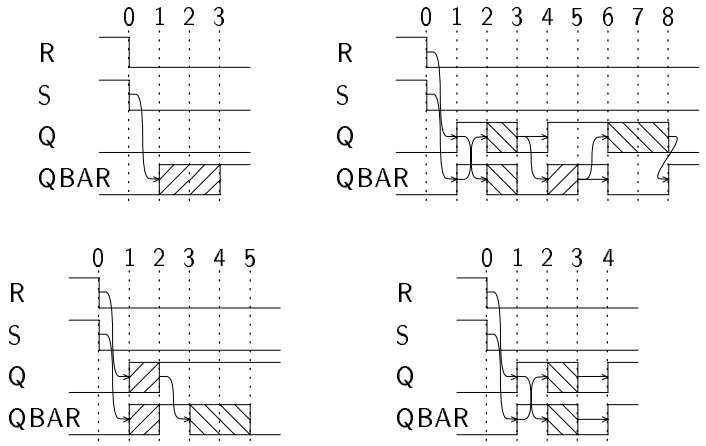


Fig. 2. Sampling of system behaviors if both S and R fall at $t = 0$

Our research shows that a useful model for the analysis and verification of digital systems must include both time ranges and signal interaction based on system state. The interaction of these two elements implies bifurcation and growth of the space of system behaviors which must be explored in this analysis. The next section provides a formal description of our specification and analysis technique followed by an overview of the algorithm developed to perform this analysis.[1]

3 Background & Definitions

For our analysis we define a system as a collection of components and the signals which interconnect them. To characterize the behavior of the system we define the system specification in terms of the events which occur on the signals of the system. In this work, the analysis process is dependent on an underlying assumption that the system is completely specified (i.e., all information regarding the system is known). This assumption is necessary and sufficient for the generation of all possible system behaviors. It is necessary since permitting the possibility

[1] The complete algorithm is fully specified in [15].

of unknown information in the system effecting its operation possibly results in the generation of system behaviors which cannot occur or the absence of system behaviors which can occur.

A partially specified system is one where not all information is known about how all signals in the system interact. Taken to an extreme, this implies that signals may change values in an undetermined manner, at any time, precluding the possibility of generating all possible system behaviors. The assumption that the system is fully specified is sufficient for the generation of all possible system behaviors. If all information is known about the system, then by definition no unanticipated action can occur in the system which affects the generation of possible system behaviors, within the scope of the model used.

We capture the specification as a set of causalities each, describing under what circumstances an event on one signal "causes" an event on another signal in the system. Informally causalities are of the form: "clock rising when D is high causes Q to go high after 10-20ns when reset is low." More formally we describe the temporal specification of the system using the following definitions.

Definition 1. The *specification of the system* is the set of all causal relationships among the signals of the system: $\mathcal{C} \equiv \{c \mid c = \langle \varepsilon_1, \varepsilon_2, \delta \rangle\}$; i.e. the set of causalities are the rules by which an event on one signal causes an event on another signal.

Definition 2. A *causality*, $c \equiv \langle \varepsilon_1, \varepsilon_2, \delta_c \rangle$ where ε_1 and ε_2 are two guarded events and $\delta_c \equiv (\alpha_c, \beta_c)$ is a time range.

For a given causality, the second event, ε_2, is specified to occur within δ_c relative to the first event, ε_1. The time range δ_c must contain two positive values.

A causality reflects a characteristic of the system being verified; it specifies that if ε_1 is enabled, ε_2 will occur provided it is also enabled within the time range δ_c after ε_1.

Definition 3. A *guarded event*, $\varepsilon \equiv \langle s, e, w \rangle$, is an event, e, on a signal, s, and an enabling expression, w, which identifies when the event is enabled. A guarded event is enabled at some time t if the enabling expression evaluates to *true* at time $t - 1$.

Definition 4. A *time range*, δ, specifies a time window for events where $\delta \equiv (min, max)$ and $min \leq max$.

Time is treated as a discrete quantity measured in an integral number of arbitrary time units. Time ranges are specified relative to some time, t. The fact that time is not continuous is not a strong constraint since the unit by which time is interpreted can be made arbitrarily small, approaching a closer approximation of continuous time, if needed.

Given two time ranges (α_1, β_1) and (α_2, β_2), if $(\beta_1 + 1) = \alpha_2$ then the union of the two time ranges is defined as $(\alpha_1, \beta_1) \cup (\alpha_2, \beta_2) = (\alpha_2, \beta_2) \cup (\alpha_1, \beta_1) = (\alpha_1, \beta_2)$.

Definition 5. A *signal*, s, is an information channel which is used to interconnect modules. Signals may connect multiple modules or a module to itself. The set of all values any signal may have is $\{high, low\} \equiv \{true, false\} \equiv \{1, 0\}$.

A signal is a physical entity (e.g., a wire) or a virtual identifier (e.g., a variable) which can have two values, high and low.

Definition 6. An *event*, e, is a rising or falling transition of a signal which occurs instantaneously.

There is no rise time or fall time associated with events, however, events themselves can occur at any point within a time range. This time range may have a zero width in which case the event occurs at a specific time. The transition of s from high to low is called the event s falling ($s\downarrow$) and the transition of signal s from low to high is the event s rising ($s\uparrow$).

Definition 7. An *enabling expression* (also referred to as a *when clause*), w, is a (possibly null) set of conjunctive (i.e. logical "and") expressions of signal names.

Each element in the set is disjunctively combined (logical "or-ed" together). Each conjunctive expression is a set of signal names or negated signal names that specifies a logical expression. The enabling expression is used to allow or disallow the occurrence of an event as described in "guarded event" (Definition 3). An example conjunctive expression is $(A \wedge \neg B \wedge C) \vee (D \wedge E)$, and is specified in set notation as $\{\{A, \neg B, C\}\{D, E\}\}$.

Definition 8. A *causal graph* is an augmented signal transition graph (STG) where the nodes are events and the directed arcs are the causalities (causal relationships between events). The graph is augmented such that the arcs contain the two enabling expressions, w_1 and w_2, and the time range, δ specified by the causality. Multiple arcs may exist between any two nodes in the graph.

With the causal graph and a set of initial conditions, it is possible to construct a system history graph which describes all possible behaviors of the system. The algorithm for the construction of the history is given in the next section. Once a history graph is generated, it is possible to traverse the graph to generate a visual representation (timing diagram) of each system history.

Definition 9. A *system history graph*, $\mathcal{H}$, is a directed graph which describes all possible behaviors from an initial state. Each node in the history graph is a dynamic state, $\mathcal{D}$, and the arcs specify a temporal ordering of the states.

Definition 10. A *dynamic state* of the system is $\mathcal{D} \equiv \langle \mathcal{S}, \mathcal{P}, t, \mathcal{S}_0, \mathcal{P}_0, \langle ds\text{-}flags \rangle \rangle$ where $\mathcal{S}$ is the static state of the system, $\mathcal{P}$ is the set of pending transitions, $\mathcal{S}_0$ is the initial static state when $\mathcal{D}$ is created, $\mathcal{P}_0$ is the initial dynamic state when $\mathcal{D}$ is created, t is the time of the earliest occurring transition in $\mathcal{P}_0$, and $\langle ds\text{-}flags \rangle \equiv \langle ProcessedFlag \rangle$ is used in the construction of $\mathcal{H}$. We say that t is the time at which $\mathcal{D}$ occurs.

The dynamic state holds all the information about the state of the system. Therefore, given any dynamic state (and the causal graph), all possible behaviors evolving from that state can be determined.

For any two dynamic states, $\mathcal{D}_i$ and $\mathcal{D}_j$, we define $\mathcal{D}_i = \mathcal{D}_j$ if $\mathcal{S}_{0:i} = \mathcal{S}_{0:j}$ and $\mathcal{P}_{0:i} = \mathcal{P}_{0:j}$. That is, these two states are equivalent such that the entire set of derivable system behaviors which can evolve from each of them is identical. Note that for all transitions in a dynamic state, all times are relative to t for that state.

The history graph is complete when $\forall i$, all successors (children) to $\mathcal{D}_i$ already appear in the graph or there are no successors to $\mathcal{D}_i$. We call such a node a terminal node. The predecessor of $\mathcal{D}_i$, called $\mathcal{D}_i$'s parent, is identified as $\mathcal{D}_i^P$.

The history graph has the following properties.

- There is a single starting dynamic state, $\mathcal{D}_0$ such that $t_0 = 0$; this is the only node with no parent.
- $\forall i \neq j, \mathcal{D}_i \neq \mathcal{D}_j$.
- $\forall i$, if $\mathcal{D}_i^P$ exists, then $t_i^P < t_i$.

Definition 11. The *static state* of the system, $\mathcal{S}$, is the last transition for all signals in the system relative to t, the time of the dynamic state: $\mathcal{S} \equiv \{l(s_i) \mid \forall signals, s_i\}$. The static system state is a snapshot of the most recent transition for every signal in the system relative to t.

Definition 12. A *signal transition*, τ, is a tuple $\langle \varepsilon, \delta, \delta_0, \eta, \eta_c, \langle st\text{-}flags \rangle \rangle$ where ε is a guarded event, $\delta \equiv (\alpha, \beta)$ is the current time range for the transition, $\delta_0 \equiv (\alpha_0, \beta_0)$ is the initial time range for the transition, η is a unique serial number for the transition, η_c is the serial number for the transition which caused this transition, and $\langle st\text{-}flags \rangle \equiv \langle TryFalseFlag, DidOccurFlag \rangle$ are used in processing the transition.

When a transition is first created, $\delta_0 = \delta$ and δ_0 is never changed after creation. During the processing of the transition, the relationships $\alpha \geq \alpha_0, beta \leq \beta_0$, and $\alpha \leq \beta$ are always maintained.

Definition 13. A *pending signal transition*, p, is a signal transition with $\alpha > 0$ which may occur in the future.

Pending transitions are created when an enabled transition occurs which has an event that matches the event specified in the left hand side of a causality. The set of all pending transitions in the system for a particular dynamic state, $\mathcal{D}$, is $\mathcal{P}$.

Definition 14. The *last transition* for a signal, $l(s)$, relative to t is a signal transition τ such that $\beta \leq 0$ and β is a maximum relative to t.

Definition 15. The *signal value*, $v(\tau)$ is defined as the resulting state a signal takes on after transition τ: $v(\tau) = 1$ if $e = \uparrow$ and $v(\tau) = 0$ if $e = \downarrow$.

Definition 16. A *system history* is defined as any path through the completed system history graph from the starting node, $\mathcal{D}_0$ to a terminal node.

4 Algorithm

The behavior of the system is completely specified by the history graph, $\mathcal{H}$. The construction of the history graph from the causal graph begins with the specification of an initial state, $\mathcal{D}_0$. Given this initial state, the main processing loop of the algorithm simply chooses any node in the graph which has not yet been processed and processes that node as shown in Fig. 3.

```
1  while node D_i in history graph not processed {
2      ProcessNode(D_i)
3  }
```

Fig. 3. Main system history graph processing loop

Pending transitions in the current dynamic state are processed by first testing if the transition is enabled, moving enabled transitions to the static state, and possibly creating a a bifurcation in the history graph. A bifurcation is a splitting of a dynamic state, $\mathcal{D}$, in the system history graph into two separate dynamic states. The composition (or union) of the two dynamic states is equivalent to the original dynamic state. The split is necessary to disambiguate possible system history alternatives. The procedure specified by *ProcessNode* shown in Fig. 4 handles the processing of a given history node, $\mathcal{D}$, which has not yet been processed.

```
1  ProcessNode(D) {
2      foreach p ∈ P s.t. α = 0 {
3          if (TransitionOccurred?(p, D)) { ProcessCausalities(p, D) }
4          α_p ← 1
5          RemoveTransitionIfApplicable(p, D)   }
6      #   check if done processing, i.e. D has no pending transitions
7      if (P = ∅) { return }
8      #   create the new node by copying the parent
9      D_new ← ⟨S, P, t, S, P, ⟨false⟩⟩
10     #   find the time offset between the new node and its parent
11     tDelta ← min(α ∈ P)
12     #   set the new time for the new node
13     t_new ← tDelta + t
14     #   adjust all times in D_new to be relative to t_new rather than t
15     foreach α, β ∈ S_new, P_new, S_0:new, P_0:new, subtract tDelta
16     #   make sure a node the same as D_new doesn't already exist
17     if (∃D_i ∈ H s.t. D_i = D_new) {
18         point D to D_i, destroy D_new, return }
19     else { point D to D_new, return }   }
```

Fig. 4. Procedure to process a dynamic history node

First, all pending transitions which might occur at t (i.e. $\alpha = 0$) are processed (line 2). If the pending transition occurred, causalities which are enabled by the transition are processed, possibly creating additional pending transitions (line 3). If a particular transition occurs, the causality graph is checked for all new transitions which may be caused by this transition and the new transitions are added to $\mathcal{P}$. Finally, all pending transitions processed are removed from $\mathcal{P}$ if they can no longer occur (i.e. $\beta = 0$) (line 5).

Next, the node is checked for pending transitions. If no pending transitions are present, then no further processing need be performed on $\mathcal{D}$ (line 7). If $\mathcal{D}$ has pending events, then a successor (child node) to $\mathcal{D}$ is created by copying $\mathcal{S}$ into both $\mathcal{S}_{new}, \mathcal{S}_{0:new}$ and copying $\mathcal{P}$ into both $\mathcal{P}_{new}, \mathcal{P}_{0:new}$ (line 9). The initial copies are saved in $\mathcal{S}_{0:new}$ and $\mathcal{P}_{0:new}$ to permit backtracking or rollback of time to a previous state during history bifurcation. Next, the time of $\mathcal{D}_{new}$ is determined by taking the minimum starting time for all pending events in $\mathcal{D}_{new}$ and all times in $\mathcal{D}_{new}$ are adjusted to be relative to t_{new} (lines 11-15).

A check is then made to see if $\mathcal{D}_{new}$ is equivalent to a node which already exists in the graph, $\mathcal{D}_i$ (i.e. if $\mathcal{S}_0$ and $\mathcal{P}_0$ of the two history nodes match). This case may occur as shown in Fig. 5 if a system has a common sequence of states as in (a) which are eventually entered or if a cycle exists in the system history graph, as in (b).

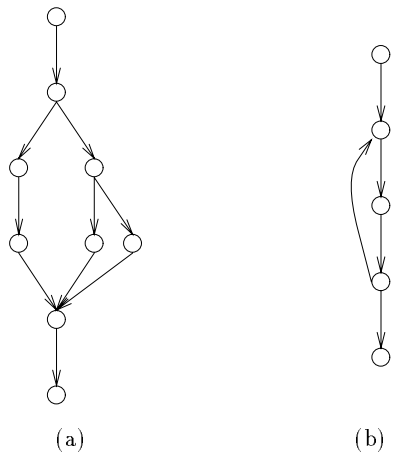


Fig. 5. History graph: reconvergent dynamic states (a) and cycling dynamic states (b)

If $\mathcal{D}_{new}$ is equivalent to $\mathcal{D}_i$, then no new histories would be generated by processing $\mathcal{D}_{new}$, therefore $\mathcal{D}_{new}$ is destroyed and $\mathcal{D}$ is pointed to $\mathcal{D}_i$ (line 18). If $\mathcal{D}_{new}$ does not match some other node, $\mathcal{D}_i$ in the graph, then processing continues, linking $\mathcal{D}_{new}$ into the history graph by pointing $\mathcal{D}$ to $\mathcal{D}_{new}$ (line 19). At this point, the processing of $\mathcal{D}$ is complete.

The procedure *TransitionOccurred?* seen in Fig. 6 and Fig. 8 handles the

```
    #  process a single transition pending now, p
1   TransitionOccurred?(p, 𝒟) {
2       #  check if all signals used in enabling expressions are defined
3       if ((SigList ←ReturnListOfUndefinedSignals(w_p, 𝒟))≠ null) {
4           SplitUndefinedEnablingSignals(p, SigList, 𝒟) }
5       #  only process p if it is enabled now (at t)
6       if (not EnabledNow?(w_p, η_p, 𝒟)) {
7           #  try the false case as long as p did not cause its own
8           #  enabling expression to become false
9           if (not TransitionCausedEnablingChange?(p, 𝒟)) {
10              ⟨TryFalseFlag⟩ ← ⟨true⟩ }
11          #  not enabled ⇒ no additional processing for p
12          return false  }
13      if (∃p_i ∈ 𝒫 s.t. α_{p_i} = 0 ∧ β_p ≠ β_{p_i} ∧ p_i is enabled ∧p ≠ p_i ∧ s_p = s_{p_i}) {
14          #  split since both p and p_i can occur
15          SplitOtherChangesPendingNow(p, 𝒟) }
16      #  check for another enabled transition, p_i, pending now
17      #  (α = 0) with direction opposite of p (↑ vs. ↓)
18      if (∃p_i ∈ 𝒫 s.t. α_{p_i} = 0 ∧ p_i is enabled ∧ s_p = s_{p_i} ∧ e_p ≠ e_{p_i}) {
19          #  both p and p_i can occur, try the false case later
20          ⟨TryFalseFlag⟩ = ⟨true⟩ }
```

Fig. 6. Procedure to process a single transition – part 1

processing of a given transition, $p \in \mathcal{P}$. First, a check is made to see if all signals used in w_p are defined. For each signal that is unknown, a bifurcation occurs for the signal to take on a high and low value (lines 3-4). Next, the enabling expression in p is checked to see if the event is enabled at t (line 6). If the event is not enabled, if may not be enabled because p itself may cause a change on a signal used in its own enabling expression. It is not possible for a signal to effect its own enabling expression since by definition, it must have already occurred to have any effect on any other signal in the system.

If p is not enabled, then a flag is set indicating that a bifurcation may be necessary when p is removed from $\mathcal{P}$ (i.e. when $\beta = 0$) and the processing of p is complete at t (line 12). If p can occur, then a test is made to see if another transition is pending in $\mathcal{P}$, on the same signal s, of the same event event type as e, which can occur now ($\alpha = 0$) and is enabled (line 13). If all these conditions are true, then a bifurcation in $\mathcal{H}$ occurs (line 15).

Figure 7 (a) illustrates the case where multiple transitions are pending on A at $t = t_0$. Here A′ is the transition associated with p and A″ and A‴ are other pending transitions on A.

The bifurcation results in this single dynamic state, $\mathcal{D}$, being replaced by two new dynamic states which also occur at t, shown in Fig. 7 (b). In the first newly created dynamic state, p (A′ in this example) is left to transition at t and all other enabled transitions on the same signal in $\mathcal{P}$ have their start time (α) incremented by one time unit to start their transitions at t_1. In the second newly created created dynamic state, the only transition modified is p which has its

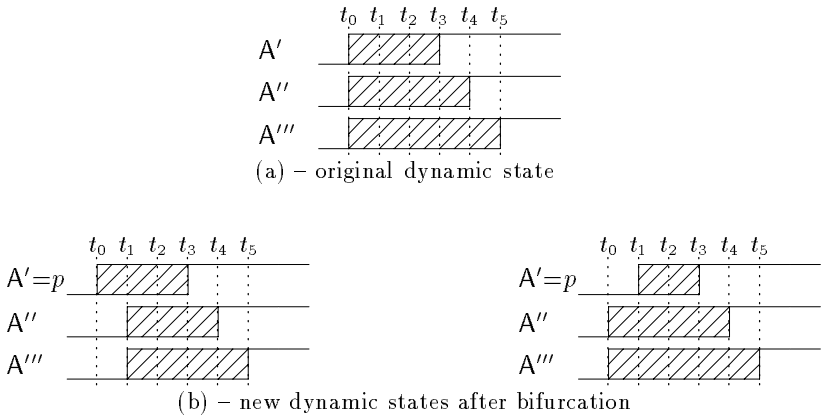


Fig. 7. Example of multiple pending transitions on signal A with $\alpha = 0$

start time incremented by one.

This bifurcation results in the consideration of both cases with respect to p: the case where A′ happens before A″ and A‴ and the case where A′ happens after some other transition. This is necessary since all three events, A′, A″, and A‴, are events on signal A, all events are of the same type (rising events in this example), but the enabling expressions of each event may be different, and in fact may be enabled in a mutually exclusive manner or may generate mutually exclusive results due to the value of signals used in the enabling clauses of each event, and in other transitions which are enabled. For this example, a second bifurcation will occur at t_0 since both transitions A″ and A‴ are pending now ($\alpha = 0$), but this will occur later in the processing of $\mathcal{D}$ when p equals either A″ or A‴.

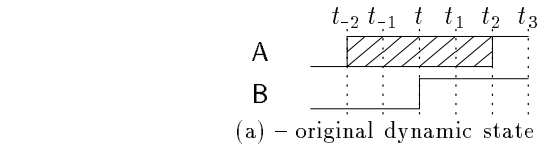
In *TransitionOccurred?*, the next step is to check if another transition pending now on p, is enabled, and is opposite to e (i.e. rising vs. falling) (line 18). If such a transition is pending, then a flag is set in p indicating that a bifurcation is necessary when p is removed from $\mathcal{P}$ (line 20). After this processing (after line 20), the dynamic state is updated to reflect the occurrence of p.

In the next part of *TransitionOccurred*? shown in Fig. 8, if the transition on s_p in the static state, τ_{last}, has the same serial number, η as p in the dynamic state, then the current transition is the same as the past transition (line 24). In this case, the ending time of the transition is simply updated in τ_{last} if s_p is not used in any enabling expressions in $\mathcal{P}_0$ (line 33). If s_p is used in an enabling expression in $\mathcal{P}_0$, then a bifurcation is necessary to guarantee that the value of s_p is stable at the time used, $t - 1$ (line 29). An example of such a bifurcation is shown in Fig. 9. Here $s_p \equiv$ A and A is used in the enabling expression of the pending transition "B↑ when A"; (a) is the original dynamic state and (b) shows the dynamic states after bifurcation.

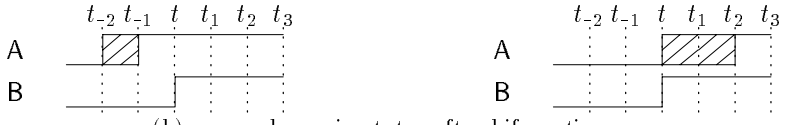
If p does not match the past transition in $\mathcal{S}$, there are two cases. The first case is that p was already processed and some other intervening transition occurred

```
21      #  find the previous transition in the static state S
22      τ_last ← l(s_p)
23      #  check if p is a continuation (elongates) τ_last
24      if (ε_p = ε_τlast  ∧  η_p = η_τlast  ∧  β_τlast = −1) {
25          #  the current transition is the same as the last transition,
26          #  a split is needed if s_p is used in any enabling expression
27          #  in a pending transition in P_0
28          if (SignalUsedInEnablingExpression?(p, D)) {
29              SplitSignalUsedInEnablingExpression(p, D)
30              return false }
31          else {
32              #  elongate τ_last by one time unit
33              β_τlast ← β_τlast + 1
34              return true  }  }
35      else {
36          #  p is not a simple elongation of τ_last
37          if (⟨DidOccurFlag⟩ = ⟨true⟩) {
38              #  try the false case if p and τ_last aren't directly related,
39              #  i.e. if τ_last didn't cause p  ∧  p didn't cause τ_last
40              if (not MutualCause?(τ_last, p, D)) {
41                  ⟨TryFalseFlag⟩ ← ⟨true⟩  }
42              #  part of p already occurred, ignore p for now
43              return false  }
44          else {
45              #  p did not previously occur, add the new transition
46              ⟨DidOccurFlag⟩ ← ⟨true⟩
47              p_copy ← copy of p
48              α_pcopy ← β_pcopy ← 0
49              τ ← l(s_p) and replace τ ∈ S with p_copy
50              return true  }  }  }
```

Fig. 8. Procedure to process a single transition – part 2



(a) – original dynamic state



(b) – new dynamic states after bifurcation

Fig. 9. Bifurcation resulting from the pending transition "B↑ when A"

on p (line 37). If this first case is true and the the past transition on s and the pending transition on s, p, do not have a direct causal relationship, then set the flag indicating a bifurcation is necessary when p is removed from $\mathcal{P}$ (line 41).

An example of this first case is shown in Fig. 10 where signal A is low and two transitions are pending, A′ and A″. At time t_{-3} and t_{-2}, A′ was the only event which could occur. At time t_{-1}, A″ can occur (provided A′ has occurred as shown), and thus the resulting A waveform is shown. Once A″ has occurred, it is not possible for A′ to reoccur, even though part of A′ is pending in $\mathcal{P}$. Therefore, the resulting transition on A only shows one rising event although A′ has a pending event in $\mathcal{P}$ for $t \geq t_0$. Since the transition on A′ can only cause one event on A; it is not possible for a single transition, A′, to cause signal A to rise twice as shown in the illegal case.

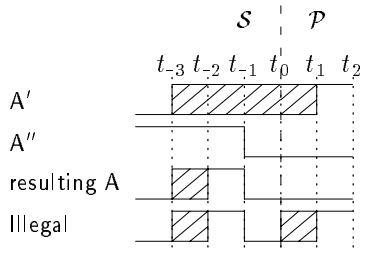


Fig. 10. Example of how a single transition on a signal cannot cause two unique events

The second case handled by *TransitionOccurred*? is if p did not previously occur, then the leading edge of the pending transition is peeled from p and the last transition in $\mathcal{S}$ is replaced by a new transition specified by the leading edge of p (lines 46-50).

The routine *ProcessCausalities* shown in Fig. 11 is executed only if p occurred at t. In this case, all causalities which have p as their left hand side are evaluated. For each such causality, the enabling expression is tested and each causality with a true enabling expression on its left hand side has the transition specified by its right hand side placed in a new pending transition, p_{new} (lines 7-10). A check if made to see if p_{new} is an extension of an existing pending transition in $\mathcal{P}$; if so, the time is updated in the existing transition (lines 13-16). Otherwise a check is made to verify that a transition identical to p_{new} does not already exist; if a transition identical to p_{new} does not exist, then p_{new} is added to $\mathcal{P}$ (line 21).

The procedure *RemoveTransition*, shown in Fig. 12, handles the removal of p from $\mathcal{P}$ if the transition can no longer occur ($\beta_p = 0$) (line 2). In this case if the flags are set indicating first, that the transition did occur and second, that the false case should be tried, then a bifurcation occurs in $\mathcal{H}$ to guarantee that the dynamic state in which p never occurs is explored (lines 3-5).

```
1  ProcessCausalities(p, 𝒟) {
2      foreach c ∈ 𝒞 {
3          #  check if all signals used in causality lhs are defined
4          if ((SigList ←ReturnListOfUndefinedSignals(w_{1,c}, 𝒟))≠ null) {
5              SplitUndefinedEnablingSignals(p, SigList, 𝒟) }
6          #  check if the lhs transition of the causality occurred
7          if (s_{1,c} = s_p ∧ e_{1,c} = e_p ∧ ε_1 is enabled) {
8              #  the causality can occur, create a new pending transition
9              #  specified by the rhs of the causality
10             p_new ← ⟨⟨s_{2,c}, e_{2,c}, w_{2,c}⟩, δ_c, δ_c, η_unique, η_p, ⟨false, false⟩⟩
11             #  check if the new transition caused is an extension
12             #  of an existing transition
13             if (∃p_i s.t. p_i ∈ 𝒫 ∧ ε_{p_i} = ε_{p_new} ∧ η_{c:p_i} = η_{c:p_new} ∧
14                     (α_{p_i} + 1) ≤ α_{p_new} ∧ β_{p_i} ≥ α_{p_new} ∧ β_{p_i} ≤ β_{p_new}) {
15                 #  update the ending time of the transition
16                 β_{0:p_i} ← β_{p_i} ← β_{p_new}  }
17             else {
18                 #  make sure p_new is not redundant in 𝒫
19                 if (not(∃p_i ∈ 𝒫 s.t. ε_{p_i} = ε_{p_new} ∧ η_{c:p_i} = η_{c:p_new})) {
20                     #  add new pending transition
21                     add p_new to 𝒫
22                     #  check if this pending transition already occurred
23                     #  and has been deleted from 𝒫
24                     if (∃τ ∈ 𝒮 s.t. ε_τ = ε_{p_new} ∧ η_{c:τ} = η_{c:p_new}) {
25                         ⟨DidOccurFlag⟩_{p_new} ← ⟨true⟩
26                         η_{p_new} ← η_τ  }  }  }  }  }  }
```

Fig. 11. Procedure to create new transitions from causalities

```
1  RemoveTransitionIfApplicable(p, 𝒟) {
2      if (β_p = 0) {
3          if (⟨TryFalseFlag, DidOccurFlag⟩ = ⟨true, true⟩) {
4              SplitFalseCase(p, 𝒟) }
5          remove p from 𝒫  }  }
```

Fig. 12. Removal of completed transitions

This bifurcation is performed by finding the dynamic state, $\mathcal{D}_i$ such that $t_i = \alpha_{0:p}$. This dynamic state is the state in which p could have first occurred. This dynamic state is copied (i.e. $\mathcal{D}_{copy} = \mathcal{D}_i$) and p removed from $\mathcal{P}_{copy}$ and $\mathcal{P}_{0:copy}$. Finally, the new dynamic state, $\mathcal{D}_{copy}$ is linked into the graph by pointing $\mathcal{D}_i^P$ to $\mathcal{D}_{copy}$.

5 Conclusion & Summary

The verification of digital circuits is a challenging problem. Our method of verifying correct operation is to generate all possible signal interactions and then to check that specified relationships hold between the signals (e.g. setup or hold

times). This approach has some characteristics of a "brute force" approach. One hopes that symbolic processing could be performed to reduce the amount of enumeration necessary and therefore reduce the complexity. However, to accurately model digital systems, both bounded time ranges and state (i.e. enabling expressions) are a necessity. Once these two elements are included in the model, it is difficult to significantly reduce the size of the search space due to complex signal interactions.

In this paper, we have presented a method for generating all possible system behaviors of a bounded delay digital circuit model given a specification and initial conditions. Although our technique explicitly enumerates the entire space of possible behaviors, we have shown that it is reasonable to analyze non-trivial systems in a short period of time. Further, describing systems with causalities is feasible for non-trivial systems. Causalities can often be derived directly from vendor supplied component specifications. Although a large number of dynamic states may be enumerated during the analysis, the automatic generation of timing diagrams from system histories makes it possible for a human to understand the behavior of complex systems.

We have emphasized the generation of system behaviors from the specification of digital circuits since this is key to any analysis, not only verification. Our future work consists of investigating ways of automatically generating appropriate initial states and transitions to enumerate all states which could result in requirement violations, thus providing a total verification system for time bounded digital circuits.

We gratefully acknowledge the support of this research by the National Science Foundation grant number MIP-9102721.

References

[1] Thomas M. McWilliams, "Verification of Timing Constraints on Large Digital Systems," *Proc. 17th Design Automation Conference* (June, 1980).

[2] Robert B. Hitchcock, Sr., "Timing Verification and the Timing Analysis Program," *Proc. 19th Design Automation Conference* (June, 1982).

[3] John J. Wallace, "On Automatic Verification of SLIDE Descriptions," Design Research Center, Carnegie-Mellon University, DRC-01-2-80, Pittsburgh, PA, Aug., 1979.

[4] David E. Wallace, "Abstract Timing Verification for Synchronous Digital Systems," Computer Science Division, Univ. of Calif. at Berkeley, UCB/CSD 88/425, Berkeley, CA, June 27, 1988.

[5] David L. Dill, "Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits," Dept. of Computer Science, Carnegie-Mellon University, CMU-CS-88-119, Pittsburgh, PA, Feb., 1988.

[6] M. Browne, E. Clarke, D. Dill & B. Mishra, "Automatic Verification of Sequential Circuits Using Temporal Logic," Dept. of Computer Science, Carnegie-Mellon University, CMU-CS-85-100, Pittsburgh, PA, Dec., 1984.

[7] Patrick C. McGeer & Robert K. Brayton, "Timing Analysis in Precharge / Unate Networks," *Proc. 27th Design Automation Conference* (June, 1990).

[8] Tod Amon & Gaetano Borriello, "On the Specification of Timing Behavior," *Proceedings of the 1990 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU '90)*, New York, NY, SIGDA (Aug., 1990).

[9] Dimitrie Doukas & Andrea S. LaPaugh, "CLOVER: A Timing Constraints Verification System," *Proceedings of the 1990 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU '90)*, New York, NY, SIGDA (Aug., 1990).

[10] F. Mavaddat & T. Gahlinger, "On Deducing Tight Bounds from Partial Timing Specifications," *Proceedings of the 1990 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU '90)*, New York, NY, SIGDA (Aug., 1990).

[11] Michael C. McFarland, "CPA: Giving an Account of Timed System Behavior," *Proceedings of the 1990 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU '90)*, New York, NY, SIGDA (Aug., 1990).

[12] Alan R. Martello & Steven P. Levitan, "Causal Timing Verification," *Proceedings of the 1990 ACM International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU '90)*, New York, NY, SIGDA (Aug., 1990).

[13] Alan R. Martello, Steven P. Levitan & Donald M. Chiarulli, "Timing Verification Using HDTV," *Proc. 27th Design Automation Conference* (June, 1990).

[14] Srinivas Devadas, Kurt Keutzer, Sharad Malik & Albert Wang, "Verification of Asynchronous Interface Circuits with Bounded Wire Delays," *Proc. 1992 Inter. Conf. for Computer-Aided Design* (Nov., 1992).

[15] Alan R. Martello, "Temporal Specification Verification," Dept. of Elect. Eng., University of Pittsburgh, PhD. Dissertation (in preparation).

This article was processed using the LaTeX macro package with LLNCS style