

Timed Event/Level Structures *

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

Abstract

This paper presents timed event/level(TEL) structures, an extension to timed event-rule structures, which allows the general use of signal levels and timing in the specification of an asynchronous circuit. TEL structures can express true OR causality, as well as language constructs that are very difficult to describe using purely event based specification methods. This flexibility makes it possible to easily express VHDL and CSP handshaking specifications as TEL structures. Circuits can be synthesized from timed event/level structures using a modified version of the geometric timing analysis method without any significant increase in synthesis time. Therefore, timed event/level structures increase specification flexibility without impacting synthesis performance.

1 Introduction

Although asynchronous circuits have been a popular research topic in universities for a number of years, they have been slow to catch on in industry. This is partly due to the fact that until very recently, most of the theoretical advantages of asynchronous design had not been conclusively demonstrated in practice. However, industry's reluctance to use this design style is largely due to the lack of asynchronous design tools that are capable of meeting their needs. The existing tool suites for synchronous design allow a designer to specify circuits in a reasonably high-level language such as Verilog or VHDL. The tools then do all the low level details of circuit synthesis. Although a number of asynchronous design tools exist, they all have weaknesses that make them unsuitable for large scale industrial designs and none of them even comes close to the flexibility and power available in synchronous CAD tools. One of the weaknesses is that nearly all existing asynchronous CAD tools lack support for explicit timing assumptions. These timing assumptions can often make the difference between an asynchronous circuit that is faster than the corresponding synchronous circuit and one that is slower. Timing assumptions can be made manually by the designer, but this is very error prone. Another weakness is that the behaviors that can be specified by the asynchronous tools are often severely limited. In particular, many asynchronous tools do not provide support for checking the level of a signal. This limits the usefulness of the tool and makes it difficult to specify any behavior where sampling the value of a signal is necessary. Simple concepts, such as a loop on a condition, often have complex or imprecise specifications if level information cannot be included. This makes asynchronous design tools harder to use and less appealing to industrial designers.

There are currently two general approaches to specifying the behavior of asynchronous circuits: language-based approaches and graph-based approaches. The two specification methods each allow a somewhat different class of circuit to be specified and require different methods for synthesis. Therefore, the specification method chosen can determine to a large extent the quality of the resulting circuit. Synthesis methods for language-based specifications directly translate a program into a circuit. One approach to this is syntax directed translation where language constructs are mapped directly to library blocks[1, 2]. In this method, signal levels and concurrency are supported, but timing information cannot be specified. Also, the circuits produced can be redundant and slow since optimizations are not seen when simply mapping program constructs to circuit blocks. In another language-based method, the specification is translated to a circuit using a series of semantic preserving transformations [3]. This approach also supports levels, but it requires a large amount of human intervention to be effective and has no support for timing.

Graph-based specification methods require a specification that is lower level than language based methods, but can make synthesis of efficient circuits easier. In one graph-based method, an interpreted Petri net or *STG* is used for specification[4, 5, 6, 7, 8]. *STGs* are very good at expressing concurrency. However, the traditional *STG* synthesis methods restrict the types of choice allowed in the net, and they have no support for level information or timing assumptions. There is an extension to *STGs* that does support levels[9], but it requires a restricted environment and synthesis algorithms for this extended specification are

*This research is supported by a grant from Intel Corporation, NSF CAREER award MIP-9625014, SRC grant 97-DJ-487, a DARPA AASERT fellowship, and an NSF Traineeship award.

not presented. Additionally, in [10] extensions to STGs that support levels and timing are presented. However, this work, like [9], does not present algorithms for synthesizing timed STGs with levels. Another graph-based method, *change diagrams* [11], is similar to STGs but removes some of the restrictions by adding different types of arcs to the specification. These additional arcs allow disjunctive behavior to be specified. However, change diagrams do not provide a way to model choice, and have no provision for timing information. Other graph-based methods specify circuits using asynchronous state-machines, and synthesis is performed using burst-mode techniques [12, 13, 14, 15]. The burst-mode method allows one purely conjunctive expression to be specified on each arc of the state machine. However, burst-mode synthesis requires the *fundamental-mode* assumption which states that when a state change occurs, all of the changing outputs are allowed to settle before any change in the input signals. This can sometimes require adding delay between the circuit and its environment so that the inputs to the circuit do not change before the outputs settle. Also, state-machine based specification is not well-suited to expressing concurrency since state machines are inherently sequential. Finally, state machines do not express causality between output and input events directly, making it difficult, if not impossible, to utilize timing assumptions to optimize the circuit.

The specification method used in the ATACS tool described in [16] is a combination of the graph-based and language-based approaches. While the tool accepts language-based specifications as input, it does not directly use them for synthesis. Instead, ATACS compiles the input program written in a timed, event-based handshaking expansion into a graph, which is then used for synthesis. ATACS uses *timed event-rule(ER) structures*, a variant of Winskel’s event structures [17] with timing, for synthesis. Since timed ER structures separate causality from conflict, they are both easier to generate from high-level descriptions, and easier to analyze. Unlike all of the previously described specification methods, timed ER structures allow the use of explicit timing assumptions in synthesis. However, like STGs, timed ER structures have no support for levels in the specification. Due to this limitation, previous versions of ATACS have limited the input languages to exclude conditional loops, true OR causality, and any other constructs that require sampling the level of a signal.

This paper presents *timed event/level(TEL) structures*, an extension to timed ER structures which allows the general use of levels in the specification of an asynchronous circuit. TEL structures allow information about levels to be included in the ER structure in the form of an arbitrary boolean expression. This makes it possible to extend the specification languages accepted by ATACS to allow the specification of conditional loops and true OR causality, as well as all other constructs that require waits on boolean expressions. TEL structures can be synthesized using a modified version of the geometric timing analysis method presented in [18], without any significant increase in synthesis time. Therefore, TEL structures facilitate more general specifications without decreasing synthesis performance.

2 Motivating example

One of the important specification constructs that is much easier to express with levels is a loop on a condition. This, or any construct that requires sampling the value of a signal and then making a decision based on the result, is very difficult to specify in a purely event-based semantics. One specification where a conditional loop is required is the *sbuf-send-pk2* controller from the HP Post Office [15] benchmark suite. This example is cited in [14] as a motivation for the level extension to burst-mode circuits and had to be modified to be expressed as an STG for the SIS benchmark suite. It is also an interesting example of the expressiveness of TEL structures.

The purpose of this controller is to manage the transfer of packets between a sender and a receiver. First, the receiver asserts *req*, which requests a line to be sent from the sender. Then the sender sends the line and raises *sendline*. When the receiver reads the line, it acknowledges the sender by raising *ackline*. Then the sender lowers *sendline*, and the receiver responds by lowering *ackline*. This protocol will continue until the receiver chooses to terminate it. To terminate the packet transfer, the receiver asserts *done* sometime after the falling transition of *sendline* but before it raises *ackline* again. When the sender detects that *done* has risen, it lowers *sendline* and also raises *ack*, indicating it has detected that the packet transfer is over. The receiver then lowers *req*, *ackline*, and *done* in parallel and the sender, in response to this, lowers *ack*.

Figure 1 shows the TEL structures that represent the circuit and environment for the *sbuf-send-pk2* controller. These TEL structures are produced by compilation of the following handshaking level description of the circuit:

```

process circuit
*[[req]; sendline↑; [-done ∧ ackline → sendline↓; [-ackline]; sendline↑; *
  | done ∧ ackline → (ack↑ || sendline↓); [-req ∧ -ackline]; ack↓; ]]
process environment
*[[req↑; [sendline]; ackline↑;
  [-sendline → (done↑ || ackline↓); [sendline]; ackline↑ [-sendline ∧ ack]; (req↓ || ackline↓ || done↓); [-ack]
  | -sendline → ackline↓ [sendline]; ackline↑; *]]

```

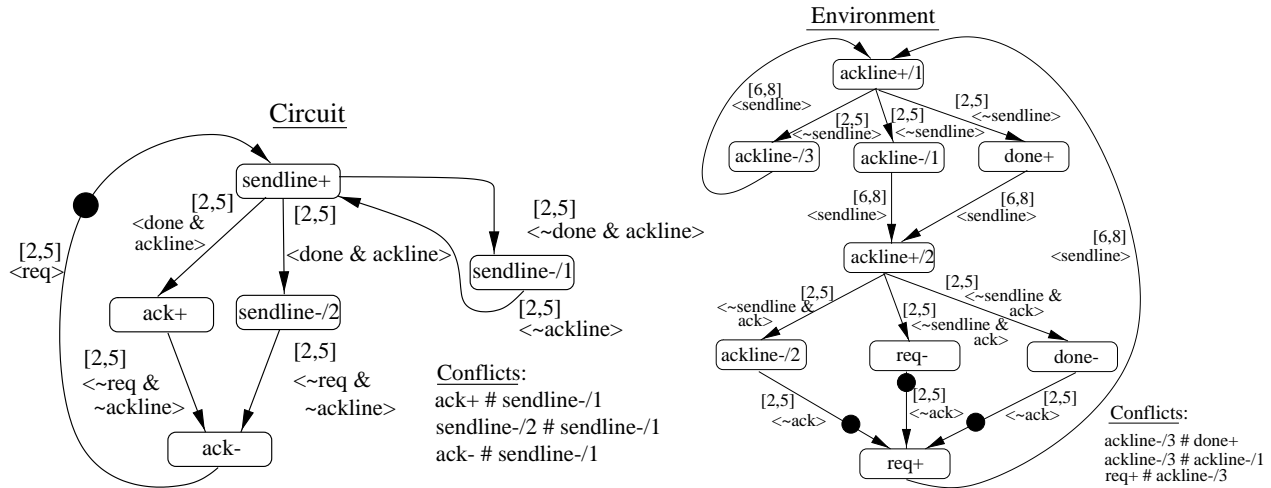


Figure 1: TEL structure *sbuf-send-pk2* controller.

The first thing to notice about the TEL structure representation is that each process in the specification is represented with a separate TEL structure. In this case, there is one TEL structure for the circuit, and a second for its environment. This makes TEL structures both easier to compile to and easier to read. When this particular specification is broken up into processes, it is clear that the circuit itself is fairly simple, while the environment is more complex. TEL structures will be defined formally in the next section, however, intuitively they can be thought of as graphical representations of timed handshaking expansions. Each transition in a process corresponds to an *event* in the TEL structure. In the figure, events are shown as boxes connected by arrows. If the same transition occurs multiple times in a handshaking expansion it may occur multiple times in the TEL structure, however an optimizing step can often remove multiple occurrences of events. The arrows that connect events are referred to as *rules*, and are annotated with both a boolean expression and a lower and upper timing bound. Rules represent the causality between events in a process. When two events occur sequentially in the handshaking expansion, a rule connects them. If there is a wait on a condition between these two events, the rule is annotated with that wait, indicating that the second event cannot occur until both the first event has occurred and the condition has been satisfied. The timing bound, which distinguishes TEL structures from the previously described specification methods, allows the designer to specify a range on the delay between the firings of events in both the circuit and its environment. The compilation of both CSP and VHDL specifications to TEL structures is addressed more formally in [19].

The behavior of TEL structures can be illustrated by examining how the structure for the *sbuf-send-pk2* makes a choice based on the value of signal *done*. If *done* is low when sampled, the handshaking indicates that only the event *sendline-* can fire, otherwise events *sendline-* and *ack+* occur in parallel. This choice is represented in the TEL structure by the conflict relation (defined formally in the next section). If two events e_1 and e_2 conflict, indicated $e_1 \# e_2$, either e_1 or e_2 can fire, but not both. In this circuit, there are two *sendline-* events, *sendline-/1* and *sendline-/2*, both of which cause the signal *sendline* to fall. However, the conflict relation states that only one of them can occur. Additionally, *ack+* conflicts with *sendline-/1*. Both the rules *sendline+* → *ack+* and *sendline+* → *sendline-/2* are annotated with the expression $\langle done \wedge ackline \rangle$, indicating that these rules cannot fire unless both *done* and *ackline* are high. This corresponds to the condition $done \wedge ackline$ in the handshaking expansion. The rule *sendline+* → *sendline-/1* is annotated with the expression $\langle \neg done \wedge ackline \rangle$, corresponding to the other choice in the handshaking expansion. If *done* is low when *ackline* rises, event *sendline-/1* will fire. If *done* is high when sampled, the TEL structure allows both *ack+* and *sendline-/2* to occur in parallel, just as specified in the handshaking expansion. This example shows how choices based on signal levels in handshaking expansions can be directly represented by TEL structures.

This example shows how TEL structures can be used to represent specifications that are quite difficult to express with purely event based specification methods. Although they are no more expressive than general Petri nets, they are more expressive than the free choice Petri nets which are required by most STG synthesis methods. Since they allow processes to be separated, they significantly simplify compilation, increase readability and make it possible to compile language constructs that involve levels. They also allow the designer to make timing assumptions in both the circuit and the environment which are not possible with the other specification methods.

3 The semantics of TEL structures

Event structures were introduced by Winskel[17] and timing has been added to them in several ways. Subrahmanyam added timing to event structures using temporal assertions [20]. Burns introduced timing in a deterministic version, the event-rule system, in which causality is represented using a set of rules, and a single delay value is associated with each rule[21]. Timed ER structures, introduced by Myers in [16], allow a delay range to be associated with each rule. They can represent a set of specifications equivalent to those represented by both time and timed Petri nets and can often express specification in a much more concise way than a Petri net. TEL structures, described formally below, extend timed ER structures by allowing a boolean expression to be associated with each rule.

3.1 Timed event/level structures

A TEL structure is a 6-tuple $T = \langle N, s_0, A, E, R, \# \rangle$ where:

1. N is the set of signals;
2. $s_0 = \{0, 1\}^N$ is the initial state;
3. $A \subseteq N \times \{+, -\} \cup \$$ is the set of actions;
4. $E \subseteq A \times (\mathcal{N} = \{0, 1, 2, \dots\})$ is the set of events;
5. $R \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times (b : \{0, 1\}^N \rightarrow \{0, 1\})$ is the set of rules;
6. $\# \subseteq E \times E$ is the conflict relation.

The signal set, N , contains the wires in the circuit specification. The state s_0 contains the initial value of each signal in N . The action set, A , contains for each signal, x , in N , a rising transition, $x+$, and a falling transition, $x-$, along with the dummy event $\$$, which is used to indicate an action that does not cause a signal transition. The event set, E , contains actions paired with occurrence indices (i.e., $\langle a, i \rangle$). Rules represent causality between events. Each rule, r , is of the form $\langle e, f, l, u, b \rangle$ where:

1. e = enabling event,
2. f = enabled event,
3. $\langle l, u \rangle$ = bounded timing constraint, and
4. b = a sum-of-products boolean function over the signals in N .

A rule is *enabled* if its enabling event has occurred and its boolean function is true in the current state. There are two possible semantics concerning the enabling of a rule. In one semantics, referred to as *non-disabling semantics*, once a rule becomes enabled, it cannot lose its enabling due to a change in the state. In the other semantics, referred to as *disabling semantics*, a rule can become enabled and then lose its enabling. This can occur when another event fires, resulting in a state where the boolean function is no longer true. In the non-disabling case, a rule is *satisfied* if it has been at least l time units since it was enabled and *expired* if it has been at least u time units since it was enabled. In the disabling case, a rule is satisfied if it has been continuously enabled for a time greater than or equal to l and expired if it has been continuously enabled for a time greater than or equal to u . The difference here is that in the non-disabling case, a change in the state after the rule is enabled does not effect the time at which it becomes satisfied or expired. Excluding conflicts, an event cannot occur until every rule enabling it is satisfied, and it must occur before every rule enabling it has expired.

The conflict relation, $\#$, is used 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. 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.

Figure 2(a) shows an example of a specification expressed as a TEL structure with non-disabling semantics. It has one conflict, $b+ \# c+$, which indicates that either the event $b+$ or the event $c+$ can occur, but not both. It also implies that only one of the signals $b+$ or $c+$ is necessary to fire $a-$. The rules $a+ \rightarrow b+$, and $a+ \rightarrow c+$ do not have level annotations. These rules function exactly the same as rules in standard ER structures and are enabled as soon as their enabling event, $a+$, fires. Since they have a bounded timing constraint of $[2,5]$, each of them becomes satisfied 2 time units after $a+$ fires and expired 5 time units after $a+$ fires. The rule $b+ \rightarrow a-$ has a level annotation, $\langle e \rangle$, and does not become enabled until both $b+$ has fired and the signal e is true. It becomes satisfied 3 time units after it becomes enabled and expired 6 time units after it becomes enabled. The rule $c+ \rightarrow a-$ also has a level annotation, $\langle f \vee g \rangle$, and becomes enabled after $c+$ has fired and $f \vee g$ is true. Since the semantics is non-disabling, once the expression has become true, the rule will become satisfied after 6 time units, even if the expression later becomes false. In general, non-disabling semantics are used for CSP or VHDL specifications. Figure 2(b) shows an **or** gate represented as a TEL structure with disabling semantics. The rule $z- \rightarrow z+$ becomes enabled when $z-$ has fired and $x \vee y$

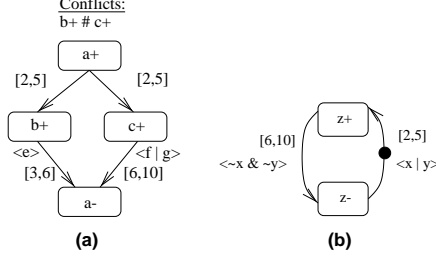


Figure 2: Examples of TEL structures.

is true. It will become satisfied 2 time units later. If both x and y become false before $z+$ fires, the rule is disabled and it is not satisfied again until 2 time units after $x \vee y$ becomes true again. Disabling TEL structures are a very intuitive and compact way to model gates. A combination of the semantics where non-disabling semantics are used for the specification and disabling semantics are used for the implementation is useful for verification.

3.2 Timed configurations

We define the behaviors specified by TEL structures as *timed configurations* [16]. Winskel defined the allowed behaviors of event structures as subsets of events, or *configurations* [17]. In order to describe the timing behavior of a TEL structure, timed configurations include the time at which each event occurred. Also, since the concept of current state is necessary for TEL structures, timed configurations are defined as sequences of events, rather than as sets, so the state that results from firing the events can be computed.

The first requirement for a sequence 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. The *con* set is the set of finite conflict-free sequences in E^* , i.e. $con \subseteq E^*$ is defined as follows:

$$con = \{X = x_0 \dots x_n \mid X \in E^* \wedge \forall x_i, x_j : \neg(x_i \# x_j)\}$$

In order to add timing, the *Tcon* set is derived from the *con* set by pairing each event with the real-valued time at which it occurred (i.e., $Tcon \subseteq (E \times \mathbb{R})^*$). The function $untime : Tcon \rightarrow con$ generates an untimed event sequence from a timed event sequence 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 sequence, all the events needed to enable the event precede it in the sequence. It is useful in the following discussion to be able to determine the set of events that occur in a sequence $X \in con$ or $Z \in Tcon$. These sets are found using the functions $set : con \rightarrow 2^E$ and $Tset : Tcon \rightarrow 2^{E \times \mathbb{R}}$:

$$\begin{aligned} set(x_0 \dots x_n) &= \{x : \exists i : x_i = x\} \\ Tset(z_0 \dots z_n) &= \{z : \exists i : z_i = z\} \end{aligned}$$

In order to determine whether a rule is enabled in a TEL structure, it is necessary to determine if the boolean expression associated with the rule is satisfied in the current state. Given that a sequence of events X has occurred, the current state is the result of firing all of the events in the sequence in order starting from the initial state s_0 . The function $s : con \rightarrow \{1, 0\}^N$ takes a sequence of events and computes the final value for each signal in the current state as follows:

$$s(X)(i) = \begin{cases} 1 & \text{if } (\exists x_j \in set(X) : x_j = \langle u_i+, m \rangle) \wedge (\neg \exists x_k \in set(X) : x_k = \langle u_i-, m' \rangle \wedge k > j) \\ 0 & \text{if } (\exists x_j \in set(X) : x_j = \langle u_i-, m \rangle) \wedge (\neg \exists x_k \in set(X) : x_k = \langle u_i+, m' \rangle \wedge k > j) \\ s_0(i) & \text{otherwise} \end{cases}$$

This equation simply states that if the last event to occur on a signal is a rising transition, its final value is 1. If the last event is a falling transition, its final value is 0. And if there have been no events on the signal it has its initial value. The current state function allows a rule enabling relation to be formally defined. As described before, there are two different semantics concerning rule enabling: one for the disabling case, and one for the non-disabling case. In order to clarify the differences, two separate rule enabling relations ($\models_{\subseteq} con \times R$) are defined:

$$\begin{aligned} X \models_{\subseteq} ((e, f, l, u, b)) &\Leftrightarrow [\exists Y \preceq X : e \in set(Y) \wedge b(s(Y))] && \text{non-disabling} \\ X \models_{\subseteq} ((e, f, l, u, b)) &\Leftrightarrow [e \in set(X) \wedge b(s(X))] && \text{disabling} \end{aligned}$$

In the non-disabling case, this means that a rule is enabled by the sequence X if there is some prefix Y of X where its enabling event has fired and its boolean condition is satisfied. A prefix of X is used for the non-disabling case since an event firing after the rule becomes enabled may cause the boolean expression to become false, and in the non-disabling case, this should not disable the rule. In the disabling case, the rule is enabled by a sequence X if its enabling event is in X and its boolean condition is satisfied by the current state. Therefore, with this definition, events firing after the rule becomes enabled may cause it to become disabled. The rule enabling relations are used in the definition of the untimed enabling relation for events, ($\vdash \subseteq con$):

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

This says that if the events in the sequence X have occurred, the event f is *untimed enabled*. This is true when all of the rules that enable f are either enabled or have an enabling event that conflicts with the enabling event of another rule enabling f that is enabled. Either the disabling or non-disabling rule enabled relation can be used when determining if an event is enabled. This choice can be made on a global basis for the entire TEL structure or can be made on a rule by rule basis by adding an extra type field to each rule. The untimed relation makes it possible to define the function, $secured \subseteq con$ which determines, given a sequence of events, whether each event was enabled when it fired:

$$secured(x_0 \dots x_n) \Leftrightarrow [\forall i \leq n : \{x_0, \dots, x_{i-1}\} \vdash x_i]$$

The *secured* relation defined above does not consider timing. In order to get a *time-secured* relation, a few more functions that deal with timing and event sequences are needed. The first one, $L : Tcon \times \mathfrak{R} \rightarrow \mathcal{N}$, returns the index of the last event in the sequence Z that fired at or before time t :

$$L(z_0 \dots z_n, t) = \max\{i : z_i = \langle x_i, t_i \rangle \wedge t_i \leq t\}$$

The next function, $time_pref : Tcon \times \mathfrak{R} \rightarrow Tcon$, uses this index to compute the prefix of the sequence Z that contains all events that fired at or before time t :

$$time_pref(Z, t) = z_0 \dots z_{L(Z, t)}$$

The function $Ts : Tcon \times \mathfrak{R} \rightarrow \{0, 1\}^N$, then uses $time_pref$ to determine the state at a given time:

$$Ts(Z, t) = s(untime(time_pref(Z, t)))$$

Finally, the relation $ConSat \subseteq (b : \{0, 1\}^N \rightarrow \{0, 1\}) \times Tcon \times \mathfrak{R} \times \mathfrak{R}$, uses Ts to determine whether a boolean expression is continuously satisfied in the interval from t_1 to t_2 :

$$ConSat(b, Z, t_1, t_2) \Leftrightarrow (\forall t : t_1 \leq t \leq t_2 \Rightarrow b(Ts(Z, t)))$$

These functions make it possible to reason about whether a rule is satisfied. The rule satisfied relation, $sat \subseteq Tcon \times R \times \mathfrak{R} \times \mathfrak{R}$, determines whether a rule is satisfied at time t given that its enabling event fired at time t' :

$$\begin{aligned} sat(Z, \langle e, f, l, u, b \rangle, t, t') &\Leftrightarrow [\exists t'' : b(Ts(Z, t'')) \wedge (t'' \geq t') \wedge (t \geq t'' + l)] && \text{non-disabling} \\ sat(Z, \langle e, f, l, u, b \rangle, t, t') &\Leftrightarrow [\exists t'' : b(Ts(Z, t'')) \wedge (t'' \geq t') \wedge (t \geq t'' + l) \wedge ConSat(b, Z, t'', t)] && \text{disabling} \end{aligned}$$

These equations state that a rule is satisfied if it has been at least l time units since its enabling event fired and its boolean equation became satisfied. In the disabling case, there is the added requirement that the boolean expression must remain continuously satisfied until time t . The *timed enabling relation*, ($\vdash_t \subseteq Tcon \times \mathfrak{R} \times E$), uses the sat relation to define whether an event f is enabled at time t by a sequence Z :

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

This says that the event f is timed enabled by a sequence Z if it is untimed enabled by the untimed version of Z and all of the rules that are necessary to make f enabled are satisfied. The *time-secured* $\subseteq Tcon$ function can now be defined as follows:

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

This says the sequence of event-time pairs in Z is time secured if and only if each event-time pair (e_i, t_i) in Z is timed enabled by the prefix of Z ending in (e_{i-1}, t_{i-1}) .

The third requirement for a subset of events to be a configuration is that it is *non-expired*. An event is expired when for each of the rules enabling it, the time since the rule was enabled has exceeded the upper bound of the rule's timing constraint. The relation $Rexp \subseteq Tcon \times R \times \mathbb{R} \times \mathbb{R}$, determines whether a rule is expired given that its enabling event fired at time t' :

$$\begin{aligned} Rexp(Z, \langle e, f, l, u, b \rangle, t, t') &\Leftrightarrow [\exists t'' : b(Ts(Z, t'')) \wedge (t'' \geq t') \wedge (t \geq t'' + u)] && \text{non-disabling} \\ Rexp(Z, \langle e, f, l, u, b \rangle, t, t') &\Leftrightarrow [\exists t'' : b(Ts(Z, t'')) \wedge (t'' \geq t') \wedge (t \geq t'' + u) \wedge ConSat(b, Z, t'', t)] && \text{disabling} \end{aligned}$$

These equations state that a rule is expired if it has been at least u time units since a time after its enabling event fired and its boolean equation was satisfied. In the disabling case, there is the added requirement that the boolean expression must have remained continuously satisfied until time t . These equations are used in the relation $expired \subseteq Tcon \times E \times \mathbb{R}$ to determine when an event is expired:

$$expired(Z, f, t) \Leftrightarrow [(untime(Z) \vdash f) \wedge \forall (e, t') \in Tset(Z) : \langle e, f, l, u, b \rangle \in R \Rightarrow Rexp(Z, \langle e, f, l, u, b \rangle, t, t')]$$

The next relation, $non-expired \subseteq Tcon \times E$, states that a timed configuration Z is non-expired if for every event, 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 the event has not occurred and is not expired at any time before the latest time of any event occurrence in the configuration.

$$\begin{aligned} non-expired(Z) &\Leftrightarrow [\forall f \in E : [(\exists (f, t) \in Tset(Z) : \neg expired(time_pref(Z, t), f, t)) \vee \\ &\quad \exists (f', t) \in Tset(Z) (f \# f') \wedge \neg expired(time_pref(Z, t), f, t) \vee \\ &\quad \forall t \in \mathbb{R}^+ \leq \max_{(e, t) \in Tset(Z)} \{t\} : \neg expired(time_pref(Z, t), f, t)]] \end{aligned}$$

A timed configuration of a TEL structure, $\langle N, s_0, A, E, R, \# \rangle$, is a sequence of event-time pairs $Z \subseteq 2^{E \times \mathbb{R}}$ which is:

1. conflict-free: $Z \in Tcon$;
2. time-secured: $time-secured(Z)$;
3. non-expired: $non-expired(Z)$.

4 Geometric timing analysis of TEL structures

Asynchronous circuits are synthesized from TEL structures by using a depth-first search to find all of the states allowed by the specification. In order to perform this search, the algorithm must be able to determine which rules are enabled to fire in any given state. A rule is untimed enabled if its enabling event has fired and its boolean expression is satisfied. Therefore, the algorithm uses a set, R_m , which contains all rules whose enabling event has fired and a state vector s_c , which indicates the current value of each signal. The pair $R_m \times s_c$ defines an *untimed state* since it indicates which rules are enabled, but says nothing about timing. From this state, the algorithm can determine the set of enabled rules, R_{en} . R_{en} can be constructed from the untimed state by including only those members of R_m whose boolean expressions are satisfied by s_c . In order to determine which rules in R_{en} are satisfied, timing information is needed. How this set of timing information, TI , is represented depends on the specific timing analysis algorithm being used. At a minimum, this information must contain how long each rule has been enabled. A *timed state* is defined to be $R_m \times s_c \times TI$. A timed state contains all the information necessary to compute the set of satisfied rules, R_s . Only rules in R_s are allowed to fire and cause a transition to another state.

Our timing information is represented with geometric regions, first introduced in [22, 23, 24]. This approach has been shown to be efficient for timed state space exploration [25, 26, 18] and can be easily modified to analyze TEL structures without any substantial increase in synthesis time. The geometric region based timing analysis method for timed ER structures is based on keeping track of the relationships between the enabling times of a set of rules. The only change that needs to be made to extend the method to TEL structures is that the enabling condition now has an added requirement. Before a rule is considered enabled, its boolean expression must be satisfied by the current state of the signals and if the disabling semantics is used, the expression must remain satisfied until the enabled event fires. It is necessary to keep track of the signal states in order to use the result of the state space exploration in synthesis anyway, so this check adds no additional space and requires minimal computation time.

When the geometric region approach is used for timing analysis, part of TI is defined to be a constraint matrix M that specifies the maximum difference in time between the enabling times of all the rules in R_{en} . The $0th$ row and column of the matrix contain the separations between the enabling times of each rule in R_{en} and a dummy rule r_\emptyset . The enabling time of r_\emptyset is defined to be uniquely 0. Each entry m_{ij} in the matrix M has the value $\max(t(enabling(j)) - t(enabling(i)))$, which is the maximum time difference between the enabling time of rule j and the enabling time of rule i . Since the enabling time of r_\emptyset is always zero, the maximum time difference between the enabling of rule i and the enabling of rule r_\emptyset (m_{0i}) is just the maximum time since i was

Algorithm 4.1 (Fire a rule)

```

timed state fire_rule(rule  $\langle e, f, l, u, b \rangle$ , timed state  $\langle R_m, s_c, M, R_f \rangle$ , TEL  $\langle N, s_0, A, E, R, \# \rangle$ , bool disabling){
   $M[m\_index(\langle e, f, l, u, b \rangle)[0] = -l$ ;
  recanonicalize( $M$ );
  project( $M, \{\langle e, f, l, u, b \rangle\}$ );
   $R_f = R_f \cup \{\langle e, f, l, u, b \rangle\}$ ;
   $R_m = R_m - \{\langle e, f, l, u, b \rangle\}$ ;
  if( $\forall \langle e_i, f, l_i, u_i, b_i \rangle \in R : ((r_i \in R_f) \vee (\exists r_j = \langle e_j, f, l_j, u_j, b_i \rangle \in R_f : e_i \# e_j))$ ){
    if( $f = \langle u_i +, m \rangle$ )  $s_c[s\_index(u_i)] = 1$ ;
    elseif ( $f = \langle u_i -, m \rangle$ )  $s_c[s\_index(u_i)] = 0$ ;
    if(disabling)
      foreach( $r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_m \cup R_f$ )
        if ( $\neg b_i(s_c) \wedge r_i \in R_m$ ) project( $M, \{\langle e_i, f_i, l_i, u_i, b_i \rangle\}$ );
        elseif( $\neg b_i(s_c) \wedge r_i \in R_f$ ){
           $R_f = R_f - r_i$ ;
           $R_m = R_m + r_i$ ;
        }
      project( $M, \{r_i = \langle e_i, f_i, l_i, u_i, b_i \rangle \in R_m : f_i \# f\}$ );
       $R_m = R_m - \{\langle e_i, f_i, l_i, u_i, b_i \rangle \in R_m : f_i \# f\}$ ;
       $R_f = R_f - \{\langle e_i, f_i, l_i, u_i, b_i \rangle \in R_f : f_i = f \vee f_i \# f\}$ ;
       $R_m = R_m \cup \{\langle e_i, f_i, l_i, u_i, b_i \rangle \in R : e_i = f\}$ ;
      add_rules( $M, R_m$ );
      advance_time( $M$ );
      recanonicalize( $M$ );
    }
  }
  return( $\langle R_m, s_c, M, R_f \rangle$ );
}

```

Figure 3: Procedure for firing a rule.

enabled. The maximum time difference between the enabling time of r_0 and the enabling time of rule i (m_{i0}) is the negation of the minimum time since i was enabled. 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 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.

When an event fires and causes new rules to be 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 used to do this is *recanonicalization*. Recanonicalization takes a matrix M where some of the m_{ij} 's are greater than $\max(t(\text{enabling}(j)) - t(\text{enabling}(i)))$ 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 the algorithm 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. When the algorithm is used for maintaining a region matrix, 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 [27].

In our version of the geometric regions algorithm [18], timing information is updated whenever a rule fires, and rules are allowed to fire independently of events. This approach is a generalization of the geometric regions technique presented in [27], where timing information only changes when an event fires. Our algorithm eliminates the *single behavioral rule restriction*, which requires that each event has only a single rule that controls its firing time. In our algorithm, 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 R_m , R_{en} , and R_s . 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.

A depth first search is used to find the state space of a TEL structure. From a timed state, $R_m \times s_c \times M \times R_f$, the search algorithm calculates the R_s set. It then chooses a rule from R_s to fire, places the rest of the rules in R_s on the stack and calls the *fire_rule* function shown in Figure 3 to actually fire the rule. If the timed state returned by *fire_rule* has been seen before,

the algorithm pops an unexplored timed state off the stack. The *fire_rule* function takes as input the rule chosen to fire and a timed state, and returns the timed state that results from firing the rule. The *m_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 rows and columns corresponding to a set of rules 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_m . 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 no event can fire, the algorithm is done. Otherwise, the algorithm updates the state vector, using the *s_index* function to find the index of the signal in the state vector. If the event is a signal firing, the appropriate bit of the state vector is updated. If the event is \$, the state vector remains unchanged. The next step is only necessary if the semantics is disabling. In this step, the algorithm checks to see if any rules have become disabled by the firing of the event. This would occur if the event firing caused the boolean function associated with the rule to become false. Timing information for any disabled rules in R_m is projected out of the constraint matrix and any disabled rules in R_f are removed and added back to R_m . This is the only difference in the algorithm between disabling and non-disabling semantics. Next, the algorithm removes any rules that enable an event that conflicts with the firing event from constraint matrix M and sets R_m and R_f . Additionally, it removes from R_f any rules that enable the firing event. Next, it adds to R_m any rules whose enabling event is the firing 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 previous rule is just the maximum time since the enabling of the previous 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. Then, 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 that are enabled in the new timed state.

This variation on the geometric regions approach allows us to analyze specifications containing levels with no increase in computational complexity. The only additional steps taken to analyze specifications with levels are checks at various points in the algorithm to see if the level is satisfied. These simple boolean checks take very little time, and the state information that they require needs to be computed for synthesis anyway. Therefore this algorithm increases the flexibility of the specification language without impacting synthesis time.

5 Synthesis of *sbuf-send-pkt2*

We attempted to synthesize this circuit using the new timing analysis algorithm for TEL structures in the ATACS system, but found that it was initially unsynthesizable due to a complete state coding(CSC) violation [4]. The CSC violation could be resolved in three ways. The standard method of adding a state variable produced a circuit, but it is somewhat complex and slow. Another way to eliminate the violation is to use a rule to order the transitions on *ack+* and *sendline - /2* so that *ack+* transitions first. This produced a faster and more efficient circuit than the state variable solution. The final method is to make a timing assumption in the specification. If we specify that the maximum time bound on *sendline+ → ack+* is 4 while the minimum time bound on *sendline+ → sendline - /2* is 5, then *ack+* will always occur first, eliminating the state that causes the CSC violation. This timing assumption yields the smallest most efficient circuit, consisting of only one generalized C-element [3] and one three input **and** gate. The timing assumption would, of course, need to be verified after synthesis to see if it is valid.

This example demonstrates the flexibility of TEL structures in asynchronous circuit synthesis. The extended burst-mode approach to specifying level signals does not give the designer flexibility to make timing assumptions or explicitly order output transitions without intervening input signal transitions. In that approach only the state variable solution to the CSC problem is possible. TEL structures give designers more choices and allow them to produce better circuits.

6 Conclusions and future work

We have presented an extension to timed ER structures that allows the specification of level signals and a timing analysis algorithm that allows circuits to be synthesized from them. This extension facilitates the compilation of many new language constructs which make asynchronous synthesis from a standard hardware description language such as VHDL possible[19]. This

added flexibility brings asynchronous synthesis tools a step closer to the power of synchronous tools. In the future, we plan on modifying the partial order techniques presented in [18] to work on TEL structures and applying TEL structures to problems in timing verification.

7 Acknowledgments

We would like to thank Brandon Bachman and Eric Mercer of the University of Utah and Dr. Steve Burns of Intel Corporation for their helpful comments and encouragement.

References

- [1] C.H. van Berkel and R. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *International Conference on Computer Design, ICCD-1988*. IEEE Computer Society Press, 1988.
- [2] E. Brunvand and R. F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *International Conference on Computer-Aided Design, ICCAD-1989*. IEEE Computer Society Press, 1989.
- [3] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.
- [4] Tam-Anh Chu. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications*. PhD thesis, MIT Laboratory for Computer Science, June 1987.
- [5] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design*, 8(11):1185–1205, November 1989.
- [6] K.-J. Lin and C.-S. Lin. Automatic synthesis of asynchronous circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 296–301. IEEE Computer Society Press, 1991.
- [7] P. Vanbekbergen, F. Catthoor, G. Goossens, and H. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 184–187. IEEE Computer Society Press, 1990.
- [8] Luciano Lavagno, Kurt Keutzer, and Alberto Sangiovanni-Vincentelli. Algorithms for synthesis of hazard-free asynchronous circuits. In *Proc. ACM/IEEE Design Automation Conference*, pages 302–308. IEEE Computer Society Press, 1991.
- [9] Cho W. Moon, Paul R. Stephan, and Robert K. Brayton. Specification, synthesis and verification of hazard-free asynchronous circuits. *Journal of VLSI Signal Processing*, 7(1/2):85–100, February 1994.
- [10] Peter Vanbekbergen, Gert Goossens, Francky Catthoor, and Hugo J. De Man. Optimized synthesis of asynchronous control circuits from graph-theoretic specifications. *IEEE Transactions on Computer-Aided Design*, 11(11):1426–1438, November 1992.
- [11] M. Kishinevsky, A. Kondratyev, A. Taubin, and V. Varshavsky. On self-timed behavior verification. In *Proceedings of ACM TAU 92*, March 1992.
- [12] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 346–350. IEEE Computer Society Press, October 1992.
- [13] Steven M. Nowick, Kenneth Y. Yun, and David L. Dill. Practical asynchronous controller design. In *Proc. International Conf. Computer Design (ICCD)*, pages 341–345. IEEE Computer Society Press, October 1992.
- [14] Kenneth Y. Yun, David L. Dill, and Steven M. Nowick. Practical generalizations of asynchronous state machines. In *Proc. European Conference on Design Automation (EDAC)*, pages 525–530. IEEE Computer Society Press, February 1993.
- [15] Bill Coates, Al Davis, and Ken Stevens. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal*, 15(3):341–366, October 1993.
- [16] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.
- [17] 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.
- [18] Wendy Belluomini and Chris J. Myers. Efficient timing analysis algorithms for timed state space exploration. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, April 1997.
- [19] Wendy Belluomini, Hao Zheng, and Chris J. Myers. Synthesis of timed circuits from vhdl specifications using timed event/level structures. forthcoming paper.
- [20] P.A. Subrahmanyam. What's in a timing discipline? considerations in the specification and synthesis of systems with interacting asynchronous and synchronous components. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*. Springer-Verlag, 1990.
- [21] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.
- [22] 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.
- [23] H. R. Lewis. Finite-state analysis of asynchronous circuits with bounded temporal uncertainty. Technical report, Harvard University, July 1989.
- [24] 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.
- [25] 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.
- [26] 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.
- [27] T. G. Rokicki. *Representing and Modeling Circuits*. PhD thesis, Stanford University, 1993.