Temporally Coarse Representation of Behavior for Model-based Troubleshooting of Digital Circuits

Walter Hamscher
Price Waterhouse Technology Centre
68 Willow Road, Menlo Park, CA 94025
pwtc !hamscher@labrea. Stanford.edu

Abstract

Model-based troubleshooting relies on having a representation of devices that can provide predictions about their behavior based on the observations that the troubleshooter makes. For complex devices such as digital circuit boards, the cost of the prediction process is a major obstacle to practical model-based troubleshooting. But increasing the efficiency of making predictions usually means sacrificing precision — thereby sacrificing diagnostie resolution so it is important to have a representation that can provide predictions that are derived from cheap observations and that are yet precise enough so that device misbehaviors will be detectable. This paper presents an implemented representation of board-scale digital circuits that emphasizes the importance of appropriate temporal abstractions for coping with behavioral complexity.

1 Motivation

Model-based troubleshooting is driven by the interaction of predictions and observations. A device model produces predictions about what ought to be observed; comparison with observations of the actual device produce discrepancies; these discrepancies are then traced to their possible underlying causes in the model and repairs of the actual device proposed. Model-based troubleshooting has been demonstrated in a variety of domains using a variety of implementation technologies Brown *et ai,* 1982] [Davis, 1984] [Genesereth, 1984] Pan, 1984] [Hamscher and Davis, 1984] [Campbell and Shapiro, 1986] [de Kleer and Williams, 1987] [Dague *et ai,* 1987] [Hamscher and Davis, 1987]. It scales well to deal with structural complexity as measured by sheer component count [First *at ai,* 1982] [Scarl *ct ai,* 1985]. However, for troubleshooting devices with complex time-dependent behavior, predicting the expected behavior of the device in response to external stimuli is complex and expensive, thus constituting a major drawback.

*This paper describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the author's research on troubleshooting was provided by DEC, Wang, Symbolics, and DARPA under ONR contract N00014-85-K-0124.

A model-based troubleshooting program that successfully diagnoses faults in behaviorally complex digital circuits is described in [Hamscher, 1988]. The troubleshooting element is XDE, a domain-independent diagnosis engine based on GDE [de Kleer and Williams, 1987]. The circuits used as examples are all from the Console Controller Board of the Symbolics 3600. About 40% of the board has been represented and several troubleshooting examples run, the largest involving 100 visible circuit nodes and 20 chips including two microprocessors. This paper focuses on the representation of circuit behavior that makes troubleshooting it possible.

There are several approaches to the problem of scaling model-based troubleshooting to deal with complex behavior. One approach that appears in some form in nearly every model-based troubleshooting program is to use a hierarchy of behavioral abstractions, using successively more detailed models as the diagnosis proceeds. The problem is that a mere commitment to using layers of behavioral abstraction says nothing about what abstractions will be *appropriate.*

To be appropriate for troubleshooting, abstractions should retain enough predictive power to detect symptoms, but should allow predictions to be made efficiently. Among the characteristics of digital circuit troubleshooting are (i) a gap of several orders of magnitude between the temporal granularity at which events occur in the machine and the temporal granularity at which observations can be easily made, and (ii) the fact that the most commen physical failures are usually manifest at coarse timescales. These characteristics mean that temporal precision can often be sacrificed without losing too much predictive force. Temporal abstractions, including familiar concepts such as *frequency, cycles, counting, sequence, duration, sampling,* and *change,* make it possible to reason about large numbers of events occurring in the circuit without having to refer explicitly to each one.

This paper first presents the temporal constraint propagator TINT. Next, some simple examples of digital component behaviors as represented with TINT are presented. Next, several temporal abstractions are presented. Finally, these temporal abstractions are used to describe the temporally coarse behavior of some components.

2 TINT

The behavior of circuit components is represented using a simple temporal reasoning system in which rules are used to derive facts about the values of functions of time. A function of time is called a *signal]* for example, the voltage at a circuit node is a signal because its value can change over time. An *event* is a change in the value of a signal.

TINT is implemented using JOSHUA [Rowley *et ai,* 1987]; the syntax will be shown as Cambridge prefix PC with [] denoting predicate terms, () denoting function terms, and the prefix ? denoting universally quantified variables. TINT provides the four-place predicate thru for making assertions about signal values, [thru ?1 ?u ?signal ?value] means that from the lower bound time ?1 to the upper bound time ?u inclusive, ?signal had value ?value. Any token can appear as the ?value of a signal.

In contrast to more sophisticated models of time (for example, the interval model in [Allen, 1984]), for simplicity time is taken to be a sparse set, the integers divisible by a temporal granularity constant δ. Granularity can be thought of as the smallest unit of time that is measurable by available instruments. Only integers, $-\infty$, and $+\infty$ can appear as time arguments to the thru and tsame predicates. This use of timestamps in TINT rather than symbolic quantities or expressions results in serious limitations as compared to other temporal reasoning systems [Williams, 1986] [Dean and McDermott, 1987], but it is adequate as a demonstration vehicle.

The ?signal argument of the thru predicate is normally a function term. For example, the term (voltage (in a u32a)) denotes the voltage signal at node (in a u32a). The voltage function maps a node to a real-valued signal. *Abstractions* and *behaviors* are functions from signals to signals. Abstractions describe relationships between signals at different levels of detail. Behaviors describe the relationships that components enforce between their input and output signals. TINT manipulates assertions with predicate ground terms containing composite terms built up from primitive signals and abstractions, **[thru $-\infty$ $+\infty$ (ll (in a u32a)) 1]**, for example, means that the logic-level at node (in a u32a) was always 1. [thru $-\infty$ $+\infty$ (change S) nil] means (literally) that the signal resulting from the application of the change abstraction to some signal S was always nil.

TINT provides rules that are used in forward chaining fashion to propagate the consequences of assertions about signal values. The following (nonsense) rule says that if ?x is of type thing, signal ?s has some value ?v from time ?1 to time ?u, and ?1 is strictly less than ?u, then the signal resulting from the application of abstraction to ?s is the result of applying fun to ?v:

```
   If [isa ?x thing]
  and [thru ?1 ?u ?s ?v]
  and (< ?1 ?u)
 Then [thru ?1 ?u (abstraction ?s) (fun ?v)]
```

The set of all thru predications (predicate ground terms) referring to the same signal is called the *history* of the signal. TINT combines overlapping intervals of the same history having the same value into *maximal intervals* and records a contradiction if a signal has more than one value at a given time.

A hybrid truth maintenance system maintains Boolean constraints among predications along with minimal environments for each assertion [McAHester, 1980] [de Kleer, 1986].

TINT provides predicates, rules, and a framework of signals and abstractions that together are used to describe circuit behavior. However, the main issue is the vocabulary of signal types and abstractions and the specific rules that the program will use to reason about them. The next sections discuss these.

3 Behaviors

Circuit components have intended behaviors that are functions from signals to signals, and these behaviors can be translated into rules. The intended behavior of a component depends on some collection of background conditions — for example, that the component in question is "working" (not physically damaged), that it is connected to a power source, and so forth. By convention, the background conditions for a component are collected and summarized as a mode signal whose value is normal during the intervals that all the conditions are satisfied. For example, the following rule says that if a two-input AND gate ?a has the status working and is getting power, then its mode is normal:

```
   If [isa ?a and2]
  and [status-of ?a working]
  and [thru ?1 ?u (power (in power ?a)) t]
 Then [thru ?1 ?u (mode ?a) normal]
```

The principal behavior rules for AND gates depends on the mode signal having the value normal. In the following rule the signals (11 ...) denote the digital signals appearing at the input ports (in 0 ?a), (in 1 ?a) and at the output port (out 0 ?a). If any input of a binary AND gate is 0 then the output is 0:

```
   If [isa ?x and2]
  and [thru ?11 ?u1 (mode ?x) normal]
  and [thru ?12 ?u2 (ll (in ?n ?x)) 0]
  and (overlap (?11 ?u1) (?12 ?u2))
 Then [thru (max ?11 ?12) (min ?u1 ?u2)
           (ll (out y ?x)) 0]
```

The function overlap tests whether the mentioned intervals have any point in common.

Another rule for the AND gate says that with all but one of its inputs held to 1, it acts as a buffer. In the two-input case, this means that as long as input ?n is I the output is the same as input (- 1 ?n) mod 2. [tsame ?1 ?u ?signall ?signal2] means that at every time between the lower bound ?1 and the upper bound ?u inclusive, ?signall has the same value as ?signal2:

```
   If [isa ?x and2]
  and [thru ?10 ?u0 (mode ?x) normal]
  and [thru ?11 ?u1 (ll (in ?n ?x)) 1]
  and (overlap (?10 ?u0) (?11 ?u1))
 Then [tsame (max ?10 ?11) (min ?u0 ?u1)
            (ll (in (- 1 ?n) ?x)) (ll (out y ?x))]
```

Other forward-chaining rules for describing the behavior of components do not correspond to their input/output directionality. For example, there is a rule that if the output of an an AND gate is 1 then all of the inputs must be 1.

The previous examples of behavior rules involved only combinational behaviors. Sequential behaviors require introducing signals to explicitly represent the internal states of components.

As with any program for reasoning about change, TINT encounters the *frame problem* [McCarthy and Hayes, 1969] [Shoham, 1986] [Lifschitz, 1987]. The approach used in TINT is not general, since it requires that each component interacts with few enough other components and in few enough ways that they can all be listed explicitly. The result is a rule — a frame axiom — for every state signal that mentions every kind of event that could change that state.

A falling-edge triggered register provides the simplest example of sequential behavior, involving only three rules. The first rule says that (a) the number appearing at the output of the register is identical to its state, and that (b) changes from 1 to 0 on the clock input are "interesting:"

```
  If [isa ?r register]
Then [tsame -∞ +∞ (state ?r) (num (out 0 ?r))]
 and [interesting-event (ll (in clk ?r)) (1 0)]
```

The value of the abstract signal (event ? from ?to ?s) is t whenever there has been a change from the value ?from to ?to. The value of this abstract signal is recorded explicitly only when that event type is marked as "interesting."

The second rule is a state-transition rule. Any change from 1 to 0 on the clock input causes the register to enter the state selected by its data input signal (num (input 0 ?r)). The previous state of the register is irrelevant. The rule below concludes that during (at least) the single moment succeeding the transition, state had the value ?input:

```
   If [isa ?r register]
  and [thru ?l1 ?u1 (mode ?r) normal]
  and [thru ?l2 ?u2 (event 1 0 (ll (in clk ?r))) t]
  and [thru ?l3 ?u3 (num (in 0 ?r)) ?input]
  and (overlap (?l1 ?u1) (?l2 ?u2) (?l3 ?u3))
 Then [thru (+ δ ?u2) (+ δ ?u2) (state ?r) ?input]
```

The third rule is a persistence rule. The register stays in whatever state it is in so long as there has been no change of the clock from 1 to 0:

```
   If [isa ?r register]
  and [thru ?l1 ?u1 (mode ?r) normal]
  and [thru ?l2 ?u2
            (event 1 0 (ll (in clk ?r))) nil]
  and [thru ?l3 ?u3 (state ?r) ?state]
  and (<= (max ?l1 ?l2) ?l3 (min ?u1 ?u2))
 Then [thru (max ?l1 ?l2) (+ δ (min ?u1 ?u2))
            (state ?r) ?state]
```

In general, transition rules deduce that a component must have been in a state for just one moment, and the persistence rules subsequently deduce how long that state must have lasted.

4 Temporal Abstractions

The notion of an "abstraction" takes on a specific meaning in TINT as a function from signals to signals. For example, suppose the function sign maps real numbers into $\{-, 0, +\}$. An example of an abstraction would be a function tsign that maps a real-valued signal into a $\{-, 0, +\}$-valued signal for each point in time. Temporal abstractions are abstractions whose pointwise definitions require reference to signal values over multiple times. The important temporal abstractions presented below include *change, counting, duration, sequences, cycles, frequency,* and *sampling.*

Change The function change is t only at moments when the underlying signal has just changed its value, otherwise it is nil. *Stay* is the obvious negation. An example showing the values of these signals over time is shown below (this and subsequent examples follow the convention that $\delta = 1$, and that the more abstract the signal the closer it appears to the top line):

(change X)	?	t	nil	nil	t
(stay X)	?	nil	t	t	nil
X	3	4	4	4	5
time	0	1	2	3	4

Similarly, the abstract signal (event ?from ?to ?S) is t whenever the underlying signal ?S has just changed from ? from to ?to. For example, (event 500 700 S) is t where S has just changed from 50 to 70:

(event :any 50 S)	?	?	nil	nil	nil	nil	t
(event 50 70 S)	.	?	nil	t	nil	nil	nil
S	?	50	50	70	30	70	50
time	0	1	2	3	4	5	6

A ? from argument of :any denotes the special case of any transition to ?to, which is useful for marking the known beginning of an interval, (event :any 50 S) is t at time 6. However, it is not known to be t at time 1 since the value of S could have been 50 at 0.

In the domain of troubleshooting circuit boards, it is much easier to observe whether a given single-bit signal changed or not during an interval of several seconds than it is to observe each individual change. The abstraction *changing with-rcspect-to* is specifically tailored to making statements about whether a given logic level signal ever changed, statements that typically arise from observations of the circuit, (changing-wrt ?1 ?u ?S) is t only at the upper bound time ?u and only when ?S changed at least once during the interval from ?1 to ?u inclusive:

(changing-wrt 1 6 S)	nil	nil	nil	nil	nil	nil	t	nil
(changing-wrt 1 3 S)	nil	nil	nil	nil	nil	nil	nil	nil
(change S)	?	nil	nil	nil	t	nil	nil	nil
S	0	0	0	0	1	1	1	1
time	0	1	2	3	4	5	6	7

For example, if [thru 6 6 (changing-wrt 1 6 S) t] is true it means that S changed at least once between times 1 and 6 inclusive.

Counting The function count-ww counts the number of events that have occurred with respect to a window of fixed width. It takes an argument n that is the width of the window in units of *6,* and a signal argument S:

(count-ww 3 S)	.	?	1	1	2	2	1
S	t	nil	nil	t	t	nil	nil
time	0	1	2	3	4	5	6

Sequences The abstraction sequence indicates when a particular string of (possibly repeated) values has appeared contiguously on a signal. Given a sequence like (0 1) it can be thought of as a finite string recognizer for occurrences of the regular expression 0+ 1"+".

(sequence '(0 1) S)	nil	nil	nil	t	nil	nil	t	nil	t
S	0	1	1	0	0	1	0	1	0
time	0	1	2	3	4	5	6	7	8

Cycles The function cycles-ww is the composition of the *count* and *sequence* abstractions. It is used to count the number of endings of a particular sequence of values:

(cycles-ww 3 '(0 1) S)	?	?	?	1	2	1	2
S	0	1	0	1	0	1	0
time	0	1	2	3	4	5	6

Typically, the larger the window, the less relative fluctuation of the cycle count over time. For example, suppose A and B are signals that are just slightly out of phase, (cycles-ww n ... A) and (cycles-ww n ... B) will have the same value most of the time, and will never differ by more than l.

(cycles-ww 8 .. A)	2	2	3	2	2	3	2	2
(cycles-ww 8 .. B)	2	3	2	2	3	2	2	3
(sequence .. A)	nil	nil	t	nil	nil	t	nil	nil
(sequence .. B)	nil	t	nil	nil	t	nil	nil	t
time	0	1	2	3	4	5	6	7

The larger the window, the less the relative difference, and conversely, the easier to detect significant deviations (as for example the difference between a signal occasionally asserted and one that is running at about 20 Khz). By convention, the window size is usually taken to be 1000 times the expected period of the signal, so that the cycles-ww of a pair of signals can be judged as equal if they differ by no more than $\frac{1}{10}$%, that is, by no more than one cycle in a thousand.

Frequency Frequency is the number of cycles that occurred during a window, divided by the duration of that window. The abstraction function fww yields the frequency of a signal with respect to a window size and a particular sequence of values. With a sufficiently large window relative to the cycle time (for example, 1000 times as large), the result is an adequate approximation to the normal notion of "frequency."

(fww 3 '(0 1) S)	?	?	?	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{1}{3}$	$\frac{2}{3}$
S	0	1	0	1	0	1	0
time	0	1	2	3	4	5	6

Sampling The notion of sampling is essential to understanding behavior of synchronous systems; here, the sampling of a signal refers to the values that the signal takes on at certain (usually regularly spaced) moments. The abstraction function sample-and-hold (abbreviated samp) takes two argument signals V and S; V is t where the signal S is to sampled. The value of samp is the value of S where V was last t:

(samp V S)	?	1	1	1	1	0	0	0
V	nil	t	nil	nil	nil	t	nil	nil
S	1	1	0	1	0	0	0	1
time	0	1	2	3	4	5	6	7

Using Temporal Abstractions Abstractions define how a signal such as (11 n48) (the logic level at node 48) relates to signals "below" it such as (voltage n48), and signals "above" it such as (fww 10^6 ' (0 1) (11 n48)) (the frequency at node 48, measured at cycles starting with 0 and with a window of 10^6 *S* time units). Abstractions thus yield rules that fire "upward," "downward," or even "sideways" between different abstractions of the same base signal.

The important property of temporal abstractions is that they sacrifice precision without sacrificing the ability to detect faulty behavior. In troubleshooting the idea is to detect discrepancies between the observed behavior of the real device and an idealized model of it; thus the predictions of interest are those that can be made efficiently from what we have observed and that could be significantly violated if the device were broken. The *change* abstraction is useful because it is easy to observe whether signals in a device are changing or not, and easy to predict what the consequences of change (or lack of it) would be. Similarly, the *frequency* abstraction is useful even if frequencies are hard to observe accurately: the distinction between zero and nonzero frequencies is easy to observe and is likely to result in significantly different behavioral consequences. By summarizing (possibly very long) sequences of events, temporal abstractions make complex behaviors look simple enough for troubleshooting to be tractable.

5 Temporally Coarse Behaviors

Component behaviors can be described with respect to more than one level or kind of abstraction. Given any abstraction A and behavior B we can define a function AB that describes the abstracted behavior (Figure 1). For example, let A be the sign abstraction, and let B be real addition. The abstracted behavior AB is the qualitative addition function qplus (Figure 2).

An example involving temporal abstractions is provided by the abstracted behavior of a counter that increments on falling edges of its input (Figure 3). By temporally abstracting its input and carry-out output with respect to the count of falling edges on each signal, a 4-bit counter can be viewed as dividing the abstracted input by 16. The output frequency would thus also be 1/16 that of the input. Viewing counters as frequency dividers in this way is useful because sometimes their inputs have known frequencies that are stable over long intervals of time. For example, one way that the frequency of the input signal could be known over a long interval is if it the output of a 9.8 Mhz oscillator. This is approximated as a frequency of 10^7 cycles per second, with a window size of a thousand periods, that is, 1000 × $\frac{1}{10^7}$ seconds:

```
  If [isa ?o oscillator]
 and [thru ?l ?u (mode ?o) normal]
Then [thru ?l ?u
          (fww 10⁻⁴sec '(0 1) (ll (out 0 ?o)))
          10⁷]
```

The frequency divider behavior allows the program to predict what the output frequency will be over similarly

Figure 1: Abstractions and Behaviors

$$\begin{tikzcd}
\mathbf{(A\ x)\ (A\ y)} \arrow[r, "\mathbf{AB}"] & \mathbf{(AB\ (A\ x)\ (A\ y))} \\
 & \mathbf{(A\ z) == (A\ (B\ x\ y))} \\
\mathbf{x\ \ y} \arrow[uu, "\mathbf{A}"] \arrow[r, "\mathbf{B}"] & \mathbf{z} \arrow[u, "\mathbf{A}"']
\end{tikzcd}$$

Figure 2: Example of Abstractions and Behaviors

$$\begin{tikzcd}
\mathbf{(sign\ x)\ \ (sign\ y)} \arrow[r, "\mathbf{qplus}"] & \mathbf{(qplus\ (sign\ x)\ (sign\ y))} \\
 & \mathbf{(sign\ z) == (sign\ (plus\ x\ y))} \\
\mathbf{x\ \ y} \arrow[uu, "\mathbf{sign}"] \arrow[r] & \mathbf{z} \arrow[u, "\mathbf{sign}"']
\end{tikzcd}$$

Figure 3: Counter Behavior with Respect to the *Counting* Abstraction

$$\begin{tikzcd}
\mathbf{(A\ x)} \arrow[r, "\mathbf{divide\ by\ 16}"] & \mathbf{(/\ (A\ x)\ 16)} \\
 & \mathbf{(A\ z)} \\
\mathbf{x} \arrow[uu, "\mathbf{A\ ==\ count\ of\ falling\ edges}"'] \arrow[r, "\mathbf{4\text{-}bit\ counter}"] & \mathbf{z} \arrow[u, "\mathbf{A\ ==\ count\ of\ falling\ edges}"]
\end{tikzcd}$$

long intervals. Frequency dividers can have multiple outputs, which by convention are numbered from 0 upwards. The frequency at the nth output is $\frac{1}{2^{n+1}}$ that of the input:

```
  If [isa ?d frequency-divider]
 and [has-port ?d (out ?n ?d)]
 and [thru ?l0 ?u0 normal (mode ?d)]
 and [thru ?l1 ?u1 ?f (fww ?w ?cyc (l1 (in a ?d)))]
 and (overlap (?l0 ?u0) (?l1 ?u1))
Then [thru (max ?l1 ?l2) (min ?u1 ?u2)
           (fww (truncate (* ?w (expt 2 (+ ?n 1))))[1]
                ?cyc (l1 (out ?n ?d)))
           (/ ?f (expt 2 (+ 1 ?n)))]
```

Signals at lower frequencies have longer periods and hence require a longer duration to go through 1000 cycles; the effect is that the window size at the nth output of a frequency divider scales by 2^{n+1}

Any behavior can be abstracted using any abstraction. Moreover, there is no reason that the same abstraction A need be applied to all the signals x, y, and z. Ideally, any prediction made by (A (B...)) will also be made by AB. However, AB will rarely be able to do so for an arbitrary combination of behavior and abstractions, even when A and B are total functions. Abstracting real addition with respect to sign, for example, yields the partial function qplus (Figure 2). The *strengthofkh* can be characterized by the degree to which it is a total function. One way to strengthen a weak function is to make assumptions about the relationship between x and y such that AB is stronger over the resulting restricted domains. In the case of sign addition, one might assume that (sign x) and (sign y) are never -, so that the resulting restriction of qualitative addition became a total function.

Given a particular abstraction function A, one should ask: for what class of behaviors B it is possible to formulate easily computable and strong abstract behaviors AB, or, failing that, what reasonable assumptions can be made to strengthen AB. In the case of temporal abstractions the answer is that they are appropriate for event-preserving behaviors. Behaviors are *even-preserving* to the extent that changes on their input signals are reflected as changes on their outputs (event-preserving behaviors include all one-to-one functions). This is such a small class that is tempting to conclude that the corresponding class of digital components is so small as to be worthless. This is not so, because it is possible to structurally compose groups of digital components and define abstract signals in such a way that the behaviors of the resulting aggregate components are event-preserving. Given that freedom, the relevant class of digital circuit *structures* is so diverse as to defy definition; it is only possible to present examples within that space.

Faced with a specific digital circuit and the above collection of temporal abstractions, principles are needed by which a person can decide how to describe its behavior. This model-building process is not automated, but can be metaphorically understood as "parsing" the circuit schematic: grouping components into composite structures and abstracting signals. The three basic principles by which behaviors are temporally abstracted are *reduction, synchronization,* and *encapsulation,* each discussed briefly below.

Reduction Any function of n inputs with one of its inputs held constant yields a new function of n — 1 inputs, and this fact can be used to form a temporally abstracted behavior for a multiple input behavior under the special case of its having one or more constant inputs. The resulting behavior is incomplete, of course, in the sense that it does not cover cases in which the inputs are not constant. It is nevertheless worthwhile because it provides an alternative to the undesirable option of predicting all behavior at a temporally detailed level. Weak temporally abstract predictions are better than none.

The simplest example is the behavior of a two-input AND gate during an interval when one of its inputs is a constant 1. By a rule shown earlier, this results in an assertion that the output and free input are the same at each moment during that interval. This assertion will have consequences for any temporal abstraction of either signal. For example, if the frequency of the free input is known then rules will fire to deduce the output frequency as well. There are similar rules for the behavior of any Boolean gate with all but one of its inputs held constant.

Synchronization Many digital circuits have signals that provide timing information, and the *sampling* abstraction can simplify the behavior of components to which they are connected. By representing the behavior of a component in terms of its inputs and outputs sampled with respect to a common clock, it may lend itself to temporal abstraction. In particular it may turn out to be nearly event-preserving.

The simplest example is a falling-edge triggered register. Its basic behavior is not event preserving, because events on its data input will not change the register state unless the latching input falls too. From another point of view, however, the register is just a delay element; its behavior is a one-to-one function, except for the delay of one clock cycle. In terms of the temporal abstractions given above, sampling the input and output with respect to falling edges on its clock reveals that an event on the (abstracted) input must be followed one clock cycle later by the same event on the (abstracted) output. A rule that takes advantage of this says that if the input is known to be changing (with respect to a clock) then the output is changing (with respect to the same clock), provided that the clock frequency was nonzero during a window falling within the time that the data signal was changing:

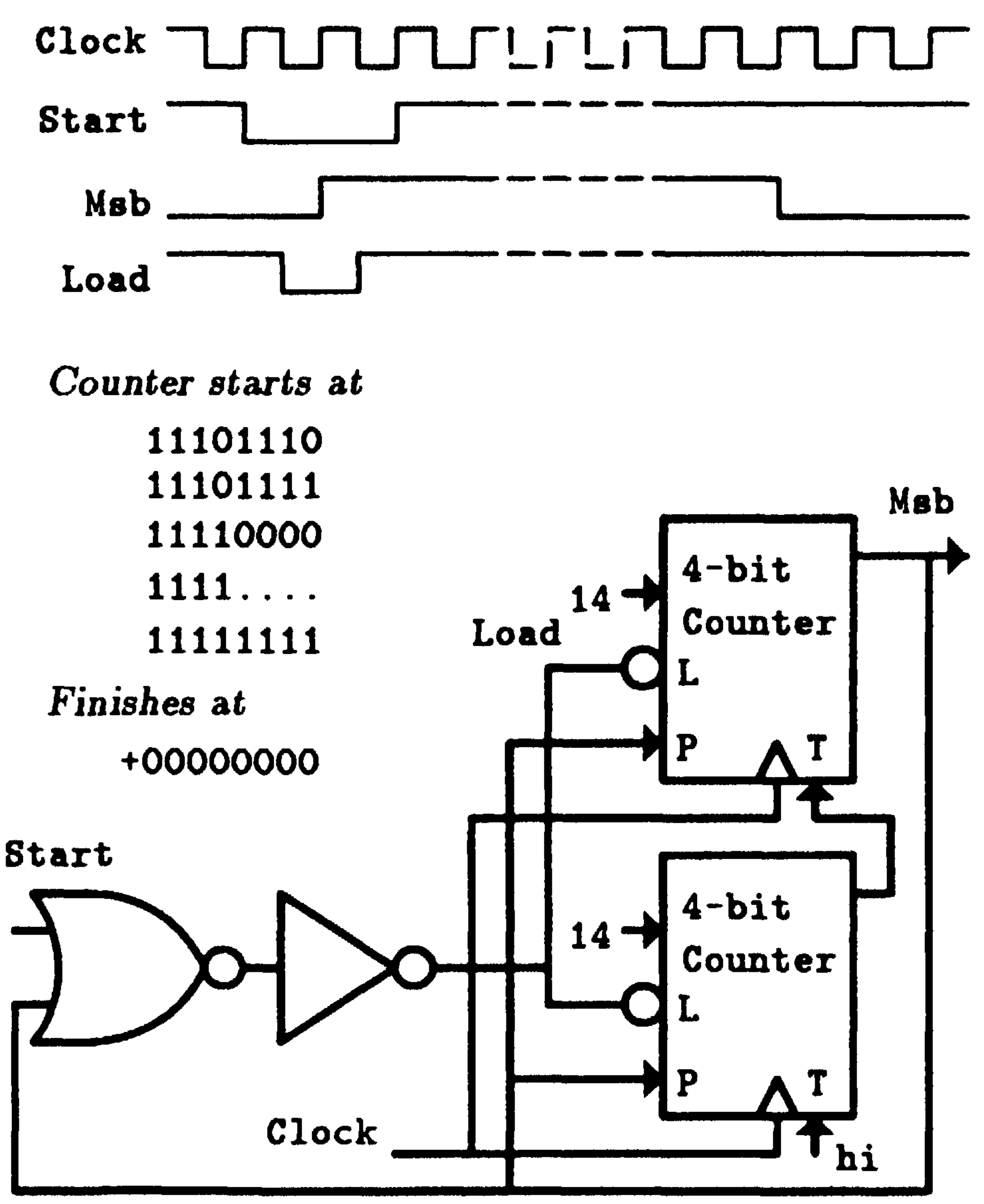
```
  If [isa ?r register]
 and [thru ?l1 ?u1 (mode ?r) normal]
 and [thru ?l2 ?u2
          (fww ?w '(0 1) (ll (in clk ?r))) ?f]
 and [thru ?z ?z
          (changing-wrt ?a ?z
            (samp (ll (in 0 ?r))
                  (fall (ll (in clk ?r))))) ?v]
 and (<= (max ?l1 ?l2) ?a ?z (min ?u1 ?u2))
 and (and (> ?f 0) (< ?w (- ?z ?a)))
Then [thru ?z ?z
          (changing-wrt ?a ?z
            (samp (ll (out 0 ?r))
                  (fall (ll (in clk ?r))))) ?v]
```

This can be extended to describe the behavior of a shift register, which can be viewed as a cascade of these delay elements. If enough changes are observed at the shift register input, a lower bound can be derived on the number of changes that should be observed at its output.

Encapsulation After grouping components together, their combined behavior may lend itself to temporal abstraction using reduction or synchronization. Figure 4 shows a circuit that is part of a serial-to-parallel converter; it detects falling edges on the Start signal and asserts its Msb eighteen cycles of the Clock later. Any subsequent falling edges on the Start signal that occur before Msb has been asserted are ignored.

Encapsulation alone does not usually simplify reasoning about the behavior of the loop. In this example, the behavior of this group of components has just as many states as the individual components. However, the whole circuit acts much like a counter (and hence much like a

Figure 4: Audio Counter



frequency divider) with respect to the Start line. The number of falling edges on Msb sampled with respect to falling edges of the Clock input is bounded from below by $\lfloor \frac{n}{18} \rfloor$ where *n* is the number of falling edges on the Start signal. The following rule says that if the frequency of the Start signal is high enough over a long enough interval, then the Msb output must have changed at least once:

```
  If [isa ?c clocked-serial-burst-detector]
 and [thru ?l1 ?u1 (mode ?c) normal]
 and [thru ?l2 ?u2
          (fww ?w '(0 1)
             (samp (fall (ll (in clock ?c)))
                   (ll (in start ?c)))) ?f]
 and (< (/ 1 ?w) ?f)
 and (overlap (?l1 ?u1) (?l2 ?u2))
 and[2][thru ?a ?z GR t]
 and (< 18 (/ (- (min ?u1 ?u2) (max ?l1 ?l2)) ?w))
Then [thru ?z ?z
          (changing-wrt ?a ?z (ll (out y ?c))) t]
```

This rule is useful because it can use information about temporally coarse signals to make predictions about other, easily observed signals. Suppose that the only information about the input is that it is a stream of 1200 bytes per second. That is enough information for this rule to fire and predict that the Msb output ought to be

[2] [thru ?a ?z GR t] means that diagnostic observations are made with respect to the time interval ?a to ?z inclusive. The interval ?a to ?z is referred to as the "observation interval. " The pattern [thru ?a ?z GR t] appears in a rule to ensure that it makes its deductions only during the current observation interval.

changing, without having to reason about the step-by-step counter behavior.

6 Conclusion

Model-based troubleshooting has not previously scaled up to deal with complex devices such as digital circuit boards. This is because traditional analytic models of complex devices do not explicitly represent aspects of the device that are important for troubleshooting. This paper has presented an overview of a digital circuit representation that was constructed with troubleshooting explicitly in mind, a representation that enables a general model-based troubleshooting engine to successfully diagnose failures in circuits that are more complex than any previously attempted. The circuit representation that makes this possible is currently embodied in the temporal constraint propagation system TINT.

Much remains to be done. First, on an engineering level, TINT is merely a demonstration vehicle; it is too slow and its timestamp-oriented ontology is not sufficiently expressive. Second, the fact that the behavior rules are ail hand-crafted is a cause for concern; the temporally coarse models ought to be derived from more basic (temporally detailed) models. Third, the approach needs to be generalized. Some preliminary work has been done in extending it to the domains of computer net works, automobile engines, and physiology.

References

[Allen, 1984] J. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence,* 23(2): 123 154, July 1984.

[Brown *et ai,* 1982] J. S. B rown, R. Burton, and J. de Kleer. Pedagogical, Natural Language, and Knowledge Engineering Issues in SOPHIE I, II, and III. In D. Sleeman and J. S. Brown, editors, *Intelligent Tutoring Systems,* pages 227-282. Academic Press, New York, 1982.

[Campbell and Shapiro, 1986] S. S. Campbell and S. C Shapiro. Using Belief Revision to Detect Faults in Circuits. Research report, Department of Computer Science, SUNY Buffalo, 1986.

[Dague *et ai,* 1987] P. Dague, 0. Rairnan, and P. Deves. Troubleshooting: When Modeling is the Difficulty. In *AAAI-87,* pages 600-605, Seattle, WA, August 1987.

[Davis, 1984] R. Davis. Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence,* 24(1):347-410, 1984. Also in *Qualitative Reasoning about Physical Systems,* Bobrow (ed.), MIT Press, Cambridge, MA 1985.

[de Kleer, 1986] J. de Kleer. An Assumption-Based TMS. *Artificial Intelligence,* 28(2):127-162, 1986.

[de Kleer and Williams, 1987] J. de Kleer and B. C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence,* 32(1):97-130, April 1987.

[Dean and McDermott, 1987] T. Dean arid 1). McDermott. Temporal Data Base Management. *Artificial Intelligence,* 32(l):l-56, April 1987.

[First *et ai,* 1982] M. B. First, B. J. Weimer, S. McLinden, and R. A. Miller. LOCALIZE: Computer-Assisted Localization of Peripheral Nervous System Lesions. *Computers and Biomedical Research,* 15(6):525-543, December 1982.

[Genesereth, 1984] M. Genesereth. The Use of Design Descriptions in Automated Diagnosis. *Artificial Intelligence,* 24(1):411-436, 1984. Also in *Qualitative Reasoning about Physical Systems,* Bobrow (ed.), MIT Press, Cambridge MA 1985.

[Hamscher and Davis, 1984] W. C. Hamscher and R. Davis. Diagnosing Circuits with State: An Inherently Underconstrained Problem. In *AAAI-84,* pages 142-147, Austin, TX, August 1984.

[Hamscher and Davis, 1987] W. C. Hamscher and R. Davis. Issues in Model-Based Troubleshooting. Memo 893, MIT Artificial Intelligence Lab, March 1987.

[Hamscher, 1988] W. C. Hamscher. Model-based Troubleshooting of Digital Systems. Technical Report 1074, MIT Artificial Intelligence Lab, August 1988.

[Lifschitz, 1987] V. Lifschitz. Formal Theories of Action (Preliminary Report). In *IJCAI-87,* pages 966-972, Milan, Italy, August 1987.

[McAllester, 1980] D. A. McAllester. An Outlook on Truth Maintenance. Memo 551, MIT Artificial Intelligence Lab, August 1980.

[McCarthy and Hayes, 1969] J. M. McCarthy and P. J. Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In D. Michie and B. Meltzer, editors, *Machine Intelligence 4,* pages 463-502. Edinburgh University Press, Scotland, 1969. Also in *Readings in Artificial Intelligence,* B. L. Webber and N. J. Nilsson (eds.), Tioga Press, 1981.

[Pan, 1984] J. Pan. Qualitative reasoning with Deep-level Mechanism Models for Diagnoses of Mechanism Failures. In *Proc. 1st IEEE Conf, on A.I. Applications,* pages 295 301, Denver, CO, 1984.

[Rowley *et ai,* 1987] S. Rowley, H. Shrobe, R. Cassels, and W. C. Hamscher. Joshua: Uniform Access to Heterogeneous Knowledge Structures, or, Why Joshing is Better than Conniving or Planning. In *AAAI-87,* pages 45 52, Seattle, WA, 1987.

[Scarl *et ai,* 1985] E. Scarl, J. R. Jamieson, and C. I. Delaune. A Fault Detection and Isolation Method Applied to Liquid Oxygen Loading for the Space Shuttle. In *IJCAI-85,* pages 414 416, Los Angeles, CA, 1985.

[Shoham, 1986] Y. Shoh am. Chronological Ignorance: Time, Nonmonotonicity, Necessity, and Causal Theories. In *AAAI-86,* pages 389-393, Philadelphia, PA, August 1986.

[Williams, 1986] B. C. Williams. Doing Time: Putting Qualitative Reasoning on Firmer Ground. In *AAAI-86,* pages 105 112, Philadelphia, PA, August 1986.