March 1983

Report No. STAN-CS-83-963

A Hardware Semantics Based on Temporal Intervals

by

Joseph Halpern, Zohar Manna
and Ben Moszkowski

Department of Computer Science

Stanford University
Stanford, CA 94305

A Hardware Semantics Based on Temporal Intervals

Joseph Halpern[1], Zohar Manna[2,3] and Ben Moszkowski[2]

[1]IBM Research Center, 5600 Cottle Road, San Jose, CA 95193, USA
[2]Department of Computer Science, Stanford University, Stanford, CA 94305, USA
[3]Applied Mathematics Department, Weizmann Institute of Science, Rehovot, Israel

Abstract

We present an interval-based temporal logic that permits the rigorous specification of a variety of hardware components and facilitates describing properties such as correctness of implementation. Conceptual levels of circuit operation ranging from detailed quantitative timing and signal propagation up to functional behavior are integrated in a unified way.

After giving some motivation for reasoning about hardware, we present the propositional and first-order syntax and semantics of the temporal logic. In addition we illustrate techniques for describing signal transitions as well as for formally specifying and comparing a number of delay models. Throughout the discussion, the formalism provides a means for examining such concepts as device equivalence and internal states.

This work was supported in part by the National Science Foundation under a Graduate Fellowship, Grants MCS79-09495 and MCS81-11586, by DARPA under Contract N00039-82-C-0250, and by the United States Air Force Office of Scientific Research under Grant AFOSR-81-0014.

This paper will appear in the Tenth International Colloquium on Automata, Languages and Programming, Barcelona, Spain, July, 1983. A full version is in preparation.

§1 Introduction

Computer systems continue to grow in complexity and the distinctions between hardware and software keep on blurring. Out of this has come an increasing awareness of the need for behavioral models suited for specifying and reasoning about both digital devices and programs. Contemporary hardware description languages (for example [1,22,29]) are not sufficient because of various conceptual limitations:

- Most such tools are intended much more for simulation than for mathematically sound reasoning about digital systems.
- Difficulties arise in developing circuit specifications that out of necessity must refer to different levels of behavioral abstraction.
- Existing formal tools for such languages are in general too restrictive to deal with the inherent parallelism of circuits.

The logic presented in this paper overcomes these problems and unifies in a single notation digital circuit behavior that is generally described by means of the following techniques:

- Register transfer operations
- Flowgraphs and transition tables
- Tables of functions
- Timing diagrams
- Schematics and block diagrams

Using the formalism, we can describe and reason about qualitative and quantitative properties of signal stability, delay and other fundamental aspects of circuit operation.

We develop an extension of linear-time temporal logic [18,25] based on intervals. The behavior of programs and hardware devices can often be decomposed into successively smaller periods or intervals of activity. These intervals provide a convenient framework for introducing quantitative timing details. State transitions can be characterized by properties relating the initial and final values of variables over intervals of time. In fact, we feel that interval-based temporal logic provides a sufficient basis for directly describing a wide range of devices and programs. For our purposes, the distinctions made in dynamic logic [10,24] and process logic [6] between programs and propositions seem unnecessary.

The temporal logic's applicability is not limited to the goals of computer-assisted verification and synthesis of circuits. This type of notation, with appropriate "syntactic sugar," can provide a fundamental and rigorous basis for communicating, reasoning or teaching about the behavior of digital devices, computer programs and other discrete systems. Moszkowski [20,21] has applied it to describing and comparing devices ranging from delay elements up to a clocked multiplier and the Am2901 ALU bit slice developed by Advanced Micro Devices, Inc. Temporal logic also provides a basic framework for exploring the computational complexity of reasoning about time. Simulation-based languages can perhaps use such a formalism as a vehicle for describing the intended semantics of delays and other features. Manna and Moszkowski [17] show how temporal logic can itself serve as a programming language.

§2 Propositional Temporal Logic with Intervals

We first present the propositional part of the temporal logic; this provides a basis for the first-order part.

Syntax

The propositional temporal logic consists of propositional logic with the addition of modal constructs to reason about intervals of time.

Formulas are built inductively out of the following:

- Propositional variables: $P, Q, \ldots$
- Logical connectives: $\neg w$ and $w_1 \wedge w_2$, where w, w_1 and w_2 are formulas.
- Next: $\bigcirc$ w (read *"next* w"), where w is a formula.
- Semicolon: w_1; w_2 (read "w_1 *semicolon* w_2" or "w_1 *followed by* w_2"), where w_1 and w_2 are formulas.

Models

Our logic can be viewed as linear-time ternporal logic with the addition of the "chop" operator of process logic [6,11]. The truth of variables depends not on states but on intervals. A model is a pair $(\Sigma, \mathcal{M})$ consisting of a set of states $\Sigma = s, t, \ldots$ together with an interpretation $\mathcal{M}$ mapping each propositional variable

P and nonempty interval $s_0 \ldots s_n \in \Sigma^+$ to a some truth value $\mathcal{M}_{s_0 \ldots s_n}[\![P]\!]$. In what follows, we assume Σ is fixed.

The length of an interval $s_0 \ldots s_n$ is n. An interval consisting a single state has length **0.** It is possible to permit infinite intervals although for simplicity we will omit them here. An interval can also be thought of as the sequence of states of a computation. In the language of Chandra et al. [6], our logic is "non-local" with intervals corresponding to "paths."

Interpretation of Formulas

We now extend the meaning function $\mathcal{M}$ to arbitrary formulas:

- $\mathcal{M}_{s_0 \ldots s_n}[\![\neg w]\!] = true \quad iff \quad \mathcal{M}_{s_0 \ldots s_n}[\![w]\!] = false$
  The formula $\neg w$ is true in an interval iff w is false.

- $\mathcal{M}_{s_0 \ldots s_n}[\![w_1 \wedge w_2]\!] = true \quad iff \quad \mathcal{M}_{s_0 \ldots s_n}[\![w_1]\!] = true \text{ and } \mathcal{M}_{s_0 \ldots s_n}[\![w_2]\!] = true$
  The conjunction $w_1 \wedge w_2$ is true in $s_0 \ldots s_n$ iff w_1 and w_2 are both true.

- $\mathcal{M}_{s_0 \ldots s_n}[\![\bigcirc w]\!] = true \quad iff \quad n \geq 1$ and $\mathcal{M}_{s_1 \ldots s_n}[\![w]\!] = true$
  The formula 0 w is true in an interval so.. . s_n iff w is true in the subinterval $s_1 \ldots$ s,. If the original interval has length 0, then 0 w is false.

- $\mathcal{M}_{s_0 \ldots s_n}[\![w_1; w_2]\!] = true$ *iff* $\mathcal{M}_{s_0 \ldots s_i}[\![w_1]\!] = true$ and $\mathcal{M}_{s_i \ldots s_n}[\![w_2]\!] = true$, for some i, $0 \leq i \leq n$.
  Given an interval $s_0 \ldots$ s,, the formula $w_1; w_2$ is true if there is at least one way to divide the interval into two adjacent subintervals $s_0 \ldots s_i$ and s;. . . s_n such that the formula w_1 is true in the first one, $s_0 \ldots$ s;, and the formula w_2 is true in the second, s;. . . s,.

A formula *w* is *satisfied* by a pair (M, $s_0 \ldots s_n$) iff

$$\mathcal{M}_{s_0 \ldots s_n}[\![w]\!] = true$$

This is denoted as follows:

$$(\mathcal{M}, s_0 \ldots s_n) \models w.$$

If all pairs of $\mathcal{M}$ and $s_0 \ldots s_n$ satisfy w then w is *valid,* written $\models$ w.

§3 Expressing Temporal Concepts in the Propositional Logic

We illustrate the temporal logic's descriptive power by giving a variety of useful temporal concepts. The connectives $\neg$ and $\wedge$ clearly suffice to express other basic logical operators such as $\vee$ and $\equiv$.

Examining Subintervals

For a formula w and an interval $s_0 \ldots$ s,, the construct $\diamondsuit_a w$ is true if w is true in at least one subinterval s;. . . s_j contained within se.. . s_n and possibly the entire interval $s_0 \ldots s_n$ itself. Note that the "a" in $\diamondsuit_a$ simply stands for "any" and is not a variable.

$$\mathcal{M}_{s_0 \ldots s_n}[\![\diamondsuit_a w]\!] = \textit{true} \quad \textit{iff} \quad \mathcal{M}_{s_i \ldots s_j}[\![w]\!] = \textit{true}, \text{ for some } 0 \leq i \leq j \leq \text{n}$$

Similarly, the formula El w is true if the formula w itself is true in all subintervals of $s_0 \ldots s_n$:

$$\mathcal{M}_{s_0 \ldots s_n}[\![\Box_a w]\!] = \textit{true} \quad \textit{iff} \quad \mathcal{M}_{s_i \ldots s_j}[\![w]\!] = \textit{true}, \text{ for all } 0 \leq i \leq j \leq n$$

These constructs can be expressed as follows:

$$\diamondsuit_a w \quad \equiv \quad (\textit{true};\ w;\ \textit{true})$$

$$\Box_a w \quad \equiv \quad \neg \diamondsuit_a \neg w$$

Because semicolon is associative, the definition of $\diamondsuit_a$ is unambiguous. Together, $\diamondsuit_a$ and $\Box_a$ fulfill all the axioms of the modal system S4 [12], with $\diamondsuit_a$ interpreted as *possibly* and El as *necessarily.*

Initial and Terminal Subintervals

For a given interval $s_0 \ldots s_n$ the operators $\diamondsuit_i$ and $\Box_i$ are similar to $\diamondsuit_a$ and El but only look at *initial* subintervals of the form so.. . s_i for $i \leq n$. We can express $\diamondsuit_i$ w and El w as shown below:

$$\diamondsuit_i w \quad \equiv \quad (w;\ \textit{true})$$

$$\Box_i w \quad \equiv \quad \neg \diamondsuit_i \neg w$$

For example, the formula $\Box_i(P \wedge Q)$ is true on an interval if *P* and Q are both true in all initial subintervals. The connectives $\diamondsuit_t$ and $\Box_t$ refer to *terminal* subintervals of the form $s_i \ldots s_n$ and are expressed as follows:

$$\diamondsuit_t w \quad \equiv \quad (\textit{true};\ w)$$

$$\Box_t w \quad \equiv \quad \neg \diamondsuit_t \neg w$$

Both pairs of operators satisfy the axioms of S4. The operators $\diamondsuit_t$ and $\Box_t$ correspond directly to 0 and $\Box$ in linear-time temporal logic [18].

The *Yields* Operator

It is often desirable to say that within an interval $s_0 \ldots s_n$ whenever some formula w_1 is true in any initial subinterval $s_0 \ldots s_i$, then another formula w_2 is true in the corresponding terminal interval $s_i \ldots s_n$ for any i, $0 \leq i \leq$ n. We say that w_1 *yields* w_2 and denote this by the formula $w_1 \rightsquigarrow w_2$:

$$\mathcal{M}_{s_0 \ldots s_n}[\![w_1 \rightsquigarrow w_2]\!] = \text{true}$$
$$\textit{iff} \quad \mathcal{M}_{s_0 \ldots s_i}[\![w_1]\!] = \textit{true} \text{ implies } \mathcal{M}_{s_i \ldots s_n}[\![w_2]\!] = \textit{true}, \quad \text{for al.1 } 0 \leq i \leq \text{n}$$

The yields operator can be viewed as ensuring that no counterexample of the form $w_1; \neg w_2$ exists in the interval:

$$(w_1 \rightsquigarrow w_2) \equiv \neg(w_1; \neg w_2)$$

This is similar to interpreting the implication $w_1 \supset w_2$ as the formula $\neg(w_1 \text{ A } \neg w_2)$.

Temporal Length

The construct *empty* checks whether an interval has length 0:

$$\mathcal{M}_{s_0 \ldots s_n}[\![empty]\!] \equiv true \textit{ iff } n = 0$$

Similarly, the construct *skip* checks whether the interval's length is exactly 1:

$$\mathcal{M}_{SO \ldots s_n}\ [skip] \equiv true \textit{ iff } n = 1$$

These operators are expressible as shown below:

$$empty \equiv \neg \bigcirc true$$

$$skip \equiv 0\ empty$$

Combinations of the operators *skip* and *semicolon* can be used to test for intervals of some fixed length. For example, the formula

$$skip;\ skip;\ skip$$

is true exactly for intervals of length 3. Alternatively, the connective *nest* suffices:

$$0\ 0\ 0\ empty$$

Initial and Final States

The construct beg w tests if a formula w is true in an interval's starting state:

$$\mathcal{M}_{s_0 \ldots s_n}[\![beg\ w]\!] \equiv \mathcal{M}_{s_0}[\![w]\!]$$

The connective beg can be expressed as follows:

$$beg\ w \equiv \Diamond(empty\ A\ w)$$

This checks that w holds for an initial subinterval of length 0, i.e., at the interval's first state. By analogy, the final state can be examined by the operator fin w:

$$fin\ w \equiv \Diamond(empty\ A\ w)$$

This checks that w holds for a terminal subinterval of length 0, i.e., at the interval's final state.

§4 Some Complexity Results

We prove that satisfiability for arbitrary propositional formulas is undecidable but demonstrate the decidability of a useful subset.

Theorem: Satisfiability for propositional temporal logic with semicolon is undecidable.

Chandra et al. [6] show that satisfiability for process logic with an operator called *chop* is undecidable. Our *semicolon* construct acts like *chop* and therefore our theorem strengthens their result since we do not require programs in order to obtain undecidability.

If we restrict all propositional variables to be *local* (that is, each propositional variable P is true of an interval $s_0 \ldots s_n$ iff P is true of the first state s_0), then we get a decidable logic:

Theorem: Local temporal logic with *semicolon* has a decision procedure that is elementary in the depth of the operators $\neg$ and semicolon.

This is the best we can do since Kozen (private communication) has shown that the validity problem for local temporal logic with *semicolon* is nonelementary. The proofs of these theorems will appear in the full paper.

§5 First-Order Temporal Logic with Intervals

We now give the syntax and semantics of the first-order temporal logic. Expressions and formulas are built inductively as follows:

Syntax of Expressions

- Individual variables: $U, V, \ldots$
- Functions: $f(e_1, \ldots, e_k)$, where $k \geq 0$ and $e_1, \ldots, e_k$ are expressions. In practice, we use functions such as + and $\vee$ (bit-or). Constants like 0 and 1 are treated as zero-place functions.

Syntax of Formulas

- Predicates: $p(e_1, \ldots, e_k)$, where $k \geq 0$ and $e_1, \ldots, e_k$ are expressions. Predicates include $\leq$ and other basic relations.
- Equality: $e_1 = e_2$, where e_1 and e_2 are expressions.
- Logical connectives: $\neg w$ and $w_1 \wedge w_2$, where w, w_1 and w_2 are formulas.
- Universal quantification: $\forall V.$ w, where V is a variable and w is a formula.
- Next: 0 w, where w is a formula.
- Semicolon: $w_1; w_2$, where w_1 and w_2 are formulas.

Models

A model consists of a set of states $\Sigma = s, t, \ldots$ and domain D together with an interpretation $\mathcal{M}$ mapping each variable V and interval $s_0 \ldots s_n$ to some value $\mathcal{M}_{S0 \ldots s_n}[\![V]\!]$ in D. Furthermore, each function and predicate symbol is given some meaning. Each k-place function symbol f has an interpretation $\mathcal{M}[\![f]\!]$ which is a -function mapping k elements in D to a single value:

$$\mathcal{M}[\![f]\!] \in (D^k \rightarrow D)$$

Interpretations of predicate symbols are similar but map to truth values:

$$\mathcal{M}[\![p]\!] \in (D^k \rightarrow \{true, false\})$$

The semantics given here keeps the interpretations of function and predicate symbols independent of intervals and thus time-invariant. The semantics can however be extended to take into account the dynamic behavior of parameters.

Interpretation of Expressions and Formulas

We now extend the interpretation $\mathcal{M}$ to arbitrary expressions and formulas:

- $\mathcal{M}_{s_0 \ldots s_n}[\![f(e_1, \ldots, e_k)]\!] = \mathcal{M}[\![f]\!](\mathcal{M}_{s_0 \ldots s_n}[\![e_1]\!], \ldots, \mathcal{M}_{s_0 \ldots s_n}[\![e_k]\!])$,
  The interpretation of the function symbol f is applied to the interpretations of $e_1, \ldots, e_k$.
- $\mathcal{M}_{s_0 \ldots s_n}[\![p(e_1, \ldots, e_k)]\!] = \mathcal{M}[\![p]\!](\mathcal{M}_{s_0 \ldots s_n}[\![e_1]\!], \ldots, \mathcal{M}_{s_0 \ldots s_n}[\![e_k]\!])$
- $\mathcal{M}_{s_0 \ldots s_n}[\![e_1 = e_2]\!] = true$ *iff* $\mathcal{M}_{s_0 \ldots s_n}[\![e_1]\!] = \mathcal{M}_{s_0 \ldots s_n}[\![e_2]\!]$
- $\mathcal{M}_{s_0 \ldots s_n}[\![\neg w]\!] =$ $true$ *iff* $\mathcal{M}_{s_0 \ldots s_n}[\![w]\!] = false$
- $\mathcal{M}_{s_0 \ldots s_n}[\![w_1 \wedge w_2]\!] = true$ *iff* $\mathcal{M}_{s_0 \ldots s_n}[\![w_1]\!] = \mathcal{M}_{s_0 \ldots s_n}[\![w_2]\!] = true$
- $\mathcal{M}_{s_0 \ldots s_n}[\![\forall V.\, w]\!] = true$ *iff* $\mathcal{M}'_{s_0 \ldots s_n}[\![w]\!] = true$,
  for every interpretation $\mathcal{M}'$ that agrees with $\mathcal{M}$ on the assignments to all variables, function and predicate symbols except possibly the variable V.
- $\mathcal{M}_{s_0 \ldots s_n}[\![\bigcirc w]\!]$ ▪ ♦□◆ℳ *iff* n ≥ 1 and $\mathcal{M}_{s_1 \ldots s_n}[\![w]\!] = true$
- $\mathcal{M}_{s_0 \ldots s_n}[\![w_1; w_2]\!] = true$ *iff* $\mathcal{M}_{s_0 \ldots s_i}[\![w_1]\!] = true$ and $\mathcal{M}_{s_i \ldots s_n}[\![w_2]\!] = true$, for some i, $0 \leq i \leq$ n.

Satisfiability and validity of formulas are as in the propositional case.

All the other temporal operators mentioned earlier are expressible as before. In addition, existential quantification can be introduced as the dual of universal quantification:

$$3\,\mathrm{v}\,.\,\mathrm{w} \equiv \neg\forall V.\neg w$$

Values in the Data Domain

It is sufficient for our purposes that the data domain D contain natural numbers and nested finite tuples. Both 0 and 1 serve as numbers and bits, with 0 standing for low voltage and 1 standing for high voltage. The data domain does not contain any intermediate voltages or "undefined" values?

The following are sample values:

$$0, \quad 3, \quad \langle 0 \rangle, \quad \langle 1, 2 \rangle, \quad \langle\rangle, \quad \langle 6, 3, \langle\rangle, 9 \rangle$$

*The approach taken in Moszkowski [20] includes undefined values. However, their omission results in no loss of generality and somewhat simplifies the underlying logic.

We adapt the convention that an n-element tuple has subscripts ranging from 0 on the left to $n-1$ on the right.

It is assumed that $\mathcal{M}$ contains standard interpretations of function and predicate symbols such as +, $\leq$ and $\vee$ (bit-or). We also include conditional expressions and conventional operators for constructing, subscripting and determining the length of tuples.

Naming Conventions of Variables

Within an interpretation $\mathcal{M}$, a variable's values can differ from interval to interval. For convenience, we will use naming conventions to distinguish certain types of dynamic behavior.

- General variables: $\mathcal{A}, \mathcal{N}$, X, . . .
  These can vary in value from interval to interval and are also known as *non-local, path* or *interval* variables.

- Signal variables: *A, N, X,* . . .
  The value of such a variable in an interval so.. . s_n depends solely on the initial state s_0:
  $$\mathcal{M}_{s_0 \ldots s_n}[\![A]\!] = \mathcal{M}_{s_0}[\![A]\!]$$
  Thus, signals can change from state to state and are a special case of general variables. Signals can also be referred to as local or *state* variables.

- Static variables: *a, n,* x, . . .
  A static variable a has a single interpretation $\mathcal{M}[\![a]\!]$, independent of any particular interval:
  $$\mathcal{M}_{s_0 \ldots s_n}[\![a]\!] = \mathcal{M}_{t_0 \ldots t_m}[\![a]\!]$$
  All static variables are signals and are often called *global* or *frame* variables.

In general, variables such as $\mathcal{A}$, B and c range over all elements of the data -domain *D.* On the other hand, *J,* K and *n* range over natural numbers. The variables $\mathcal{X}$, Y and z always equal one of the bit values 0 and 1. If desired, the naming style suggested here can also be used in the propositional logic.

§6 Some First-Order Temporal Concepts

Within the framework of first-order temporal logic, we can explore a variety of qualitative and quantitative timing issues. The constructs given below are useful for describing and reasoning about circuits.

Temporal Assignment

The formula $A \rightarrow B$ is true for an interval if the signal A's initial value equals B's final value:

$$A \rightarrow B \quad \equiv_{\mathbf{def}} \quad \forall c.\ [beg(A = c) \supset fin(B = c)]$$

We call this *temporal assignment.* Unlike in conventional programming languages, it is perfectly acceptable to have an arbitrary expression on the receiving end of the arrow.

Properties:

$$\models \quad (A \rightarrow B) \supset [f(A) \rightarrow f(B)]$$

If A is assigned to B, then any time-invariant function application $f(A)$ is passed to $f(B)$*

$$\models \quad [(\neg Z \rightarrow Z); (\neg Z \rightarrow Z)] \supset (Z \rightarrow Z)$$

If a bit signal is twice complemented, it ends up with its original value.

Temporal Equality

Two signals A and B are *temporally equal* in an interval if they have the same values in all states. This is written $A \approx B$ and differs from constructs for initial and terminal equality, which only examine signals' values at the extremes of the interval:

$$A \approx B \quad \equiv_{\mathbf{def}} \quad \boxdot(A = B)$$

Properties:

$$\models \quad [A \approx B] \supset [f(A) \approx f(B)]$$

If A temporally equals B, then $f(A)$ temporally equals $f(B)$.

$$\models \quad [(A, B) \approx (A', B')] \equiv (A \approx A' \wedge B \approx B')$$

The pair (A, B) temporally equals (A', B') exactly if the signal A temporally equals A' and B temporally equals B'.

Temporal Stability

A signal A is *stable* if it has a fixed value. The notation used is $stb\ A$ and can be expressed as shown below:

$$stb\ A \quad \equiv_{\mathbf{def}} \quad \exists b.\ (A \approx b)$$

It follows from this that every static variable is stable.

The Temporal Function *len*

Quantitative timing properties are handled by a O-place temporal function *Zen* whose value for any interval so.. . s_n equals the length n:

$$\mathcal{M}_{s_0 \ldots s_n} [\![len]\!] = n$$

Examples

Concept	*Formula*
The signal *A* is stable and the interval has at least m + n units	$stb\, A \wedge len \geq m + n$
In some subinterval of length $\geq m$, *X* is stable	$@([Zen \geq m] \wedge stb\ X$

Blocking

It is useful to specify that as long as a signal *A* remains stable, so does another signal *B*. We say that *A blocks B* and write this as *A blk B*. The predicate *blk* can be expressed using the temporal formula

$$A\ blk\ B \quad \equiv_{\text{def}} \quad \Box(stb\ A \supset stb\ B)$$

The predicate *A blk B* can be extended to allow for quantitative timing. When describing the behavior of digital circuits, it is often useful to express that in any initial interval where *A* remains stable up to within the last m units of time, *B* is stable throughout:

$$A\ blk^m\ B \quad \equiv_{\text{def}} \quad \Box[(stb\ A;\ Zen \leq m) \supset stb\ B]$$

This modification has utility in situations where *B* is known to be slow in responding to changes in *A*.

Initial and Terminal Stability

The predicate $istb^m\ A$ is true for an interval so. . . s_n if the signal *A* is stable in the initial states $s_0 \ldots s_m$. The next definition has this meaning:

$$istb^m\ A \quad \equiv_{\text{def}} \quad \Diamond(stb\ A \wedge len = m)$$

Note that the formula is false on an interval of length less than *m*. By analogy, $tstb^m\ A$ is true if *A* ends up stable for at least *m* units of time.

Rising and Falling Signals

A rising bit signal can be described by the predicate $\uparrow X$:

$$\uparrow X \quad \equiv_{\mathbf{def}} \quad [(X \approx 0);\ skip;\ (X \approx I)]$$

This says that X is 0 for a while and then jumps to 1. The gap of quantum length represented by the test *skip* is necessary here since a signal cannot be 0 and 1 at the same instant. Falling signals are analogously described by the construct $\downarrow X$:

$$\downarrow X \quad \equiv_{\mathbf{def}} \quad [(X \approx 1);\ skip;\ (X \approx 0)]$$

These operators can be extended to include quantitative information specifying minimum periods of stability before and after the transitions. For example, timing details can be added to the operator $\uparrow$:

$$\uparrow^{m,n} X \quad \equiv_{\mathbf{def}} \quad [(X \approx 0 \ \mathrm{A}\ len \geq m);\ skip;\ (X \approx \mathbf{1} \wedge len \geq n)]$$

This can also be expressed as shown below:

$$\vDash \uparrow^{m,n} X \equiv (\uparrow X \ \mathrm{A}\ istb^{m}\ X \ \mathrm{A}\ tstb^{n}\ X)$$

Thus, the extended form of $\uparrow$ can be reduced to the original one with separate details concerning initial and terminal stability.

A. negative pulse with quantitative information can be described as shown below:

$$\downarrow\uparrow^{l,m,n} X \equiv$$
$$[(X \approx 1 \ \mathrm{A}\ len \geq 1);\ skip;$$
$$(X \approx 0 \ \mathrm{A}\ len \geq m);\ skip;\ (X \approx 1 \ \mathrm{A}\ len \geq n)]$$

These constructs can be further modified to provide for noninstantaneous rise and fall times.

Smoothness

A bit signal X is smooth if it is either stable or has a single transition. The following illustrates one way to express smoothness:

$$sm\ X \quad \equiv_{\mathbf{def}} \quad (stb\ X \ v \ \uparrow X \ v \ \downarrow X)$$

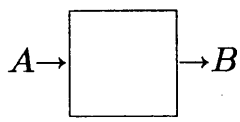
Since digital devices generally require clock inputs to be smooth, it is sometimes important to ensure that a signal has this property.

§7 Delays and Combinational Elements

Delay is a fundamental phenomenon in dynamic systems and an examination of it touches upon basic issues ranging from feedback and parallelism to implementation and internal device states. Such concepts also come into play in descriptions of more complicated devices. In addition, a key design decision in building any hardware simulator centers around the treatment of delay (see, for example, Breuer and Friedman [5]). For these and other reasons, it is worth taking a detailed look at various models of signal propagation.

Unit Delay

One of the simplest and most important types of delay elements can modeled as having the following structure:



Here A is the input signal and B is the associated output. The following statement uses intervals to characterize the desired behavior:

> *In every subinterval* ***of*** *length exactly one unit, the initial value* ***of*** *the input A equals the final value* ***of*** *the output B.*

The next predicate *del* formalizes this:

$$A \; del \; B \quad \equiv_{\text{def}} \quad \Box[(len = 1) \supset (A \rightarrow B)]$$

Property:

$$\models \; (A \; del \; A) \equiv stb \, A$$

A signal is fed back to itself iff it is stable.

Transport Delay

It is natural to extend the predicate *del* to cover delays over m-unit intervals:

$$A \; del^{m} \; B \quad \equiv_{\text{def}} \quad \Box \quad (len = m \supset [A \rightarrow B])$$

Breuer and Friedman [5] refer to this as *transport delay.*

Properties:

$$t = A\ del^0\ B \equiv A \approx B$$

Zero delay is equivalent to temporal equality.

$$\models (A\ del^m\ B \wedge B\ del^n\ C) \supset A\ del^{m+n}\ C$$

Delay is cumulative.

$$\models \langle A1, A2\rangle\ del^m\ (BS, B2\rangle \equiv (\mathrm{Al}\ del^m\ \mathrm{Bl} \wedge A2\ del^m\ B2)$$

Delay between pairs is equivalent to component-wise delay. This generalizes to tuples of arbitrary length.

Functional Delay

Often, one signal receives a delayed function of another. The following examples illustrate this and are based on the predicate *del* although other delay models can be used.

Examples

Concept	*Formula*
X keeps on being complemented	$(\neg X)\ del\ X$
. *B* either accepts *A* or itself, depending on X	[if (X = 1) then *A* else *B]* $del\ B$

Properties:

$$\models A\ del^m\ B \supset f(A)\ del^m\ f(B)$$

If *A* has a delay to *B* then it follows that *f(A)* is delayed to *f(B).*

$$\models [f(A)\ del^m\ B \wedge g(B)\ del^n\ C] \supset g(f(A))\ del^{m+n}\ C$$

Composition applies.

$$\models [(\neg X)del^m\ Y \wedge (\neg Y)\ del^n\ Z] \supset X\ del^{m+n}\ Z$$

Two inverters cancel.

$$\models (I + 1)\ del\ I \supset [(I + Zen) \rightarrow I]$$

If the variable I keeps incrementing by 1, its final value is greater than its initial value by the length of the interval.

Delay Based on Shift Register

A shift register R storing m + 1 values can be specified as follows:

$$R[0] \; del \; R[1] \; \wedge \; \ldots \; \wedge \; R[m-1] \; del \; R[m]$$

Over each unit of time, the contents of R shift right by one element. That is, the value of $R[0]$ is passed to $R[1]$ and so forth. This description is more formally expressed by means of quantification:

$$\forall i \in [0, m-1[. \; (R[i] \; del \; R[i+1])$$

The next formula has the same meaning but is more concise:

$$R[0 \text{ to } m-1] \; del \; R[1 \text{ tom}]$$

The following property shows how to achieve an m-unit delay by means of such a shift register:

$$\models \quad RIO \text{ to m-l}] \; delR[1 \text{ to } m] \supset R[0] \; del^m R[m] \qquad (*)$$

This suggests an implementation of $A \; del^m \; B$ of the form $A \; shdel_R^m \; B$:

$$A \; shdel_R^m \; B \quad \equiv_{\text{def}} \quad (A \approx R[0] \wedge R[m] \approx B \wedge RIO \text{ tom} - 1] \; del \; \text{R[l tom])}$$

Here, the yalue of A is fed into $R[0]$ and B receives the value $R[m]$. The correctness of this implementation is given by the following property:

$$\models \quad A \; shdel_R^m \; B \supset A \; del^m \; B$$

We can localize R in the formula $A \; shdel_R^m \; B$ by defining a variant $A \; shdel" \; B$ which existentially quantifies over R:

$$A \; shdel" \; B \quad \equiv_{\text{def}} \quad \exists R. \; (A \; shdel_R^m \; B)$$

The register is assumed to exist without being externally visible to an observer. The quantifier's effect on scoping is similar to that of a begin-block in a conventional block-structured programming language. We call $A \; shdel" \; B$ an *external*

specification of the implementation. In fact, this is logically equivalent to the basic delay predicate $A\ del^m\ B$ as the next property demonstrates:

$$\models \quad A\ shdel"\ B \equiv A\ del^m\ B$$

The proof that *shdel* implies *del* follows from the implementation theorem $(*)$ given above. The converse requires demonstrating that some R exists. Perhaps the easiest way to do this is by direct construction. At each instant of time, the values of the m + **1** elements of R can be those of the next m + **1** values of B in appropriate order:

$$R[\dot{\mathbf{I}} \approx \bigcirc^{m-i} B, \quad \text{for } 0 \leq i \leq m$$

The output value $R[m]$ always equals the expression $\bigcirc^0 B$, which is defined to be B's current value. Similarly, $R[0]$ always equals $0"\ B$, that is, the value B will have m units later. This technique works even if the interval has length less than m.

Variable Transport Delay

A batch of delay elements may have varying characteristics although each individual device is rather fixed in its timing behavior. The predicate $A\ vardel^{m,n}\ B$ specifies that A's value is propagated to B by transport delay with some uncertain factor between m and n:

$$A\ vardel^{m,n}\ B \quad \equiv_{\text{def}} \quad \exists i \in [m, n].\ (A\ del^i\ B)$$

Delay with Sampling

Digital circuits often require that inputs remain stable and be sampled for some minimum amount of time in order to ensure proper device operation. The delay model *A sadel B* has this characteristic:

$$A\ sadel^m\ B \quad \equiv_{\text{def}} \quad \boxdot[(stb\ A \wedge Zen \geq m) \supset fin(A = B)]$$

Here the input A must be stable at least m units of time for the output B to equal A. Behavior during changes in A is left unspecified. The properties below illustrate two other ways of expressing *sadel*. We present them to demonstrate other possible styles:

$$\models \quad A\ sadel^m\ B \equiv \boxdot(tstb^m\ A \supset fin(A = B))$$

$$\models \quad A\ sadel^m\ B \equiv [tstb^m\ A \rightsquigarrow beg(A = B)]$$

Properties:

$$\models A\ del^m\ B \supset A\ sadel^m\ B$$

Basic delay implements sampling delay.

$$\models A\ \ sadel^m\ B \equiv (tstb^m\ A \rightsquigarrow [\ beg(A = B)\ \wedge\ A\ blk\ B])$$

Once the device stabilizes, the input *A* blocks the output *B*.

The predicate *sadel* can be extended to associate some factor with the blocking of *B* by *A:*

$$A\ sadel^{m,n}\ B \quad \equiv_{\text{def}} \quad (tstb^m A \rightsquigarrow [beg(A = B) \wedge A\ blk^n\ B])$$

In a sense, *m* is the maximum delay and *n* is the minimum delay.

An Equivalent Delay Model with an Internal State

A related delay model $A\ stdel_X^{m,n}\ B$ is based on a bit flag X that is set to 1 after the input *A* has been held stable m units. Whenever X is 1, the input *A* equals the output *B* and blocks X, which in turn blocks *B* by the factor *n:*

$$\begin{aligned} A\ stdel_X^{m,n}\ B \quad &\equiv_{\text{def}} \\ &\Box([stb\ A\ \wedge\ len \geq m] \supset fin(X = 1)) \\ &\wedge\ \Box(beg(X = 1) \supset [beg(A = B)\ \wedge\ A\ blk\ X\ \wedge\ X\ blk^n\ B]) \end{aligned}$$

In the manner described earlier, we internalize X by existentially quantifying over it:

$$A\ stdel^{m,n}\ B \equiv \exists X.\ (A\ stdel_X^{m,n}\ B)$$

This external form is in fact logically equivalent to $A\ sadel^{m,n}\ B$:

$$\models A\ stdel^{m,n}\ B \equiv A\ sadel^{m,n}\ B$$

The following construction for X can be used:

$$X \approx \text{if } [\ beg(A = B)\ \wedge\ A\ blk^n\ B] \text{ then 1 else 0}$$

There arc a variety of specifications that use different internal signals such as X and yet are externally equivalent.

Delay with Separate Propagation Times for 0 and 1

Sometimes it is important to distinguish between the propagation times for 0 and 1. The following variant of *sadel* does this by having separate timing values fpr the two cases:

$$A\ sadel01^{m,n}\ B \quad \equiv_{\mathbf{def}}$$
$$\quad \boxdot([A \approx 0\ \mathbf{A}\ Zen \geq m] \supset fin(A = B))$$
$$\mathbf{A}\ \boxdot([A \approx \mathbf{1}\ \mathbf{A}\ Zen \geq n] \supset fin(A = B))$$

Smooth Delay Elements

It is possible to specify that between times when the delay element is stable, if the input changes smoothly, then so does the output. We call *such* a device a *smooth* delay element. This type of delay has utility in systems which must propagate clock signals without distortion. Here is a predicate based on the earlier specification *stdel:*

$$A\ smdel_X^{m,n}\ B \quad \equiv_{\mathbf{def}}$$
$$A\ stdel_X^{m,n}\ B$$
$$\mathbf{A}\ \boxdot([beg(X = 1)\ \mathbf{A}\ fin(X = \mathbf{1})\ \mathbf{A}\ sm\ A] \supset sm\ B)$$

The external form quantifies over X:

$$A\ smdel^{m,n}\ B \quad \equiv_{\mathbf{def}} \quad 3X.\ (A\ smdel_X^{m,n}\ B)$$

Delay with Tolerance to Noise

Sometimes it is important to consider the affects of transient noise during signal changes. A signal *A* is *almost smooth* with factor l if *A* is continuously stable all but at most l contiguous units of time:

$$stb\ A;\ (len \leq l);\ stb\ A$$

The delay model *toldel* is similar to *smdel* but has an additional timing coefficient l for showing how almost smooth input changes result in smooth output transitions:

$$A\ toldel^{m,n,l}\ B \quad \equiv_{\mathbf{def}}$$
$$A\ stdel_X^{m,n}\ B$$
$$\mathbf{A}\ \boxdot[(beg(X = \mathbf{1})\ \mathbf{A}\ fin(X = \mathbf{1})\ \mathbf{A}\ [stb\ A;\ (len \leq l);\ stb\ A]) \supset sm\ B]$$

From this we can obtain the external form

$$A \ toldel^{m,n,l} \ B$$

The predicate *smdel* is a special case of *toldel* with a noise tolerance of 1 time unit:

$$\models \ A \ smdel^{m,n} \ B \equiv A \ toldel^{m,n,1} \ B$$

Gates with Input and Output Delays

One might specify an and-gate with both input and output delays as follows:

$$(X, X')saand^{m,n}Y \quad \equiv_{\text{def}} \quad \exists Z, Z'.\left[X\, sadel^{m}\, Z \ \wedge \ X' sadel^{m}\, Z' \ \wedge \ (Z \ \wedge \ Z')sadel^{n}Y\right]$$

Here a delay exists from the input X to an internal signal Z and another delay occurs from X' to Z'. The bit-and of Z and Z' is propagated to Y. The input delays are given by *m* and the output one by *n*. If we choose to ignore input delays, the model reduces to a single occurrence of *sadel*:

$$\models (X, X') \ saand^{0,n} \equiv (X \ \wedge \ X') \ sadel^{n} \ Y$$

If the internal propagation is modeled by transport delay, things are even simpler. Here is an and-gate specified in this manner:

$$(X, X') \ tand^{m,n} \ Y \quad \equiv_{\text{def}} \quad \exists Z, Z'. \ [X \ del^{m} \ Z \ \wedge \ X' \ del^{m} \ Z' \ \wedge \ (Z \ \wedge \ Z') \ del^{n} \ Y]$$

The predicate *tand* simplifies even if internal input delay is not ignored:

$$\models \ (X, X') \ tand^{m,n} \ Y \equiv (X \ \wedge X') \ del^{m+n} \ Y$$

§8 Simple Latch

A latch is a simple memory element for storing and maintaining a single bit of data. The two inputs S and R determine what value is stored with S standing for *Set* and R standing for *Reset*. When the latch is stable, the outputs Q and $\overline{Q}$ are complements. Note that the bar in "$\overline{Q}$" is part of the name and not an operator. Such elements are among the simplest storage devices that can be constructed

out of TTL gates and provide a basis for building counters and other sequential components. Here is one way to specify such a latch:

$$(S,\ R)\ latch^{m,n}\ (Q, \overline{Q}) \quad \equiv_{\mathbf{def}}$$
$$\Box[(S \approx 0 \ \wedge\ R \approx 1 \ \wedge\ len \geq m)$$
$$\rightsquigarrow (beg[Q = 0 \ \wedge\ \overline{Q} = 1] \ \wedge\ S\ \ blk^n \langle Q, \overline{Q}\rangle)]$$
$$\wedge\ [(S \approx 1 \wedge R \approx 0 \wedge len \geq m)$$
$$\rightsquigarrow (beg[Q = 1 \ \wedge\ \overline{Q} = 0] \ \wedge\ R\ \ blk^n \langle Q, \overline{Q}\rangle)]$$

For example, the specification states that after S is 1 and R is 0 for at least m units of time, Q equals 1, $\overline{Q}$ equals 0 and R blocks both with factor n. That is, the outputs are stable as long as R remains "inactive" at 0, independent of S's behavior. A logically equivalent specification based on an internal state is given in the full paper.

A latch can be constructed out of two nor-gates that feed back to one another:

$$\models\ [\neg(R \vee \overline{Q})\ sadel^{m,n}\ Q \ \wedge\ \neg(S \vee Q)\ sadel^{m,n}\ \overline{Q} \ \wedge\ n \geq 1]$$
$$\supset [(S,\ R)\ latch^{2m,n}\ (Q,\ \overline{Q})]$$

The gates' blocking factor n must be nonzero in order to achieve a feedback loop that maintains a stored value.'

§9 Some Variants of Temporal Logic

There are a variety of operators and concepts that can be added to the temporal logic. We discuss a few here.

Iteration

The logic can be generalized to include iteration. In the proposition case, this involves adding the Kleene closure of *semicolon*. This does not affect our basic complexity results. Loop operators such as *while* can be expressed by means of such a construct.

Ignoring Intervals

The concepts presented here can generally be expressed in linear-time temporal logic [18] with $\bigcirc$, $\Box$, *0* and $\mathcal{U}$. The satisfiability of propositional formulas for such a logic is PSPACE-complete [28]. I-lowever, the conciseness and clarity provided by *semicolon* and other interval-dependent constructs are' often lost.

Infinite Intervals

In the semantics already given, all intervals are restricted to being finite. It can however be advantageous to consider infinite intervals arising out of nonterminating computations. The inclusion of such intervals does not alter the complexity of satisfiability.

Projection

Sometimes it is desirable to examine to behavior of a device at certain points in time and ignore all intermediate states. This can be done using the notion of *temporal projection*. The formula w_1 II w_2 in an interval forms a subinterval consisting of those states where w_1 is true and then determines the value of w_2 in this subinterval:

$$\mathcal{M}_{s_0 \ldots s_n}[\![w_1 \ \Pi \, w_2]\!] = \mathcal{M}_{t_0 \ldots t_m}[\![w_2]\!],$$

where $t_0 \ldots t_m$ is the sequence of the states in so.. . s_n that satisfy w_1:

$$\mathcal{M}_{t_i}[\![w_1]\!] = true, \quad \text{for } 0 \leq i \leq m$$

Note that to.. . t_m need not be a contiguous subsequence of $s_0 \ldots$ s,. If no states can be found, the projection is *false.* In the semantics given here, the formula w_1 examines states, not intervals. For example, the formula

$$(X = 1) \ \Pi \ stb \ A$$

is true is A has a constant value throughout the states where X equals 1. Variables like X act as metrics for measuring time and facilitate different levels of atomicity. If two parts of a system are running as different rates, metrics can be constructed to project away the asynchrony. Other definitions of projection are also possible.

Additional Modifications

Further possible extensions include quantification over propositional variables as well as interval-oriented temporal logics based on branching or probabilistic models of time.

§10 Related Work

We now mention some related research on the semantics of hardware. Gordon's work [8] on register-transfer systems uses a denotational semantics with partial

values to provide a concise means for reasoning about clocking, feedback, instruction-set implementation and bus communication. Talantsev [30] as well as Betancourt and McCluskey [3] examine qualitative signal transition concepts corresponding to $\uparrow X$ and $\downarrow X$. Wagner [31] also uses such constructs as $\uparrow X$ in a semi-automated proof development system for reasoning about signal transitions and register transfer behavior. Malachi and Owicki [16] utilize a temporal logic to model self-timed digital systems by giving a set of axioms. Bochmann [4] uses a linear-time temporal logic to describe and verify properties of an arbiter, a device for regulating access to shared resources.

Leinwand and Lamdan [14] present a type of Boolean algebra for modeling signal transitions. Applications include systems with feedback and critical timing constraints. Patterson [23] examines the verification of firmware from the standpoint of sequential programming. Meinen [19] discusses a semantics of register transfer behavior. McWilliams [15] develops computational techniques for determining timing constraints in hardware. Eveking [7] uses predicate calculus with explicit time variables to explore verification in the hardware specification language Conlan.

A number of people have used temporal logics to describe computer communication protocols [9,13,26]. Bernstein and Harter [2] augment linear-time temporal logic with a construct for expressing that one event is followed by another within some specified time range. This facilitates the treatment of various quantitative timing issues. Recently Schwartz et al. [27] have introduced a temporal logic for reasoning about intervals. They distinguish intervals from propositions.

For our purposes, much of this work either has difficulties in treating quantitative timing, lacks rigor, is unintuitive or does not easily generalize. In particular, we believe that in many papers on applications of temporal logic, various basic aspects of discrete-time systems have be neglected in favor of more "glamorous" protocols and distributed algorithms. Furthermore, the computational models used generally interleave the executions of different processes. In the treatment of digital circuits, this approach seems inappropriate.

It has been argued by some that temporal logic is simply a subset of dynamic logic. However, once interval-dependent constructs are added, this is no longer the case. Operators such as *semicolon* and *yields* are not directly expressible in dynamic logic. Furthermore, the descriptive styles used in dynamic logic and temporal logic differ rather greatly. Dynamic logic and process logic stress the interaction between programs and propositions. Temporal logic is expressive enough to conveniently and directly specify a variety of useful programs. Our current view is that the addition of program variables would be redundant.

§11 Conclusion

Standard temporal logics and other such notations are not designed to concisely handle the kinds of quantitative timing properties and signal transitions that occur in the examples considered. Temporal intervals provide a unifying means for presenting the various features. Even without intervals, some of the dynamic concepts discussed here have utility in specifications and properties about discrete-time systems.

Moszkowski [21] uses the logic for describing and comparing a variety of digital devices. Manna and Moszkowski [17] show how to program directly in temporal logic. Future work will explore microprocessors, buses and protocols, DMA, firmware and instruction sets, as well as the combined semantics of hardware and software. We also plan to examine compilers and other systems that transmit and manipulate commands and programs.

References

1. M. R. Barbacci. "Instruction Set Processor Specifications (ISPS): The notation and its applications." *IEEE Transactions on Computers* C-30, 1 (Jan. 1981), 24-40.
2. A. Bernstein and P. Harter. "Proving real-time properties of programs with temporal logic." Proceedings of the 8-th Symposium on Operating Systems Principles, Pacific Grove, California, Dec., 1981, pages l-11.
3. R. Betancourt and E. J. McCluskey. Analysis of sequential circuits using clocked flip-flops. Technical Report 82, Digital Systems Laboratory, Stanford University, Aug., 1975.
4. G. V. Bochmann. "Hardware specification with temporal logic: An example." *IEEE Transactions on Computers* C-31, 3 (March 1982), 223-231.
5. M. A. Breuer and A. D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, Inc., Woodland Hills, California, 1976.
6. A. Chandra, J. Halpern, A. Meyer, and R. Parikh. Equations between regular terms and an application to process logic. Proceedings of the 13-th Annual ACM Symposium on Theory of Computing, Milwaukee, Wisconsin, May, 1981, pages 384-390.
7. H. Eveking. The application of Conlan assertions to the correct description of hardware. Proceedings of the IFIP 5-th International Conference on Computer Hardware Description Languages and their Applications, Kaiserslautern, West Germany, Sept., 1981, pages 37-50.

8. M. Gordon. Register transfer systems and their behavior. Proceedings of the IFIP 5-th International Conference on Computer Hardware Description Languages and their Applications, Kaiserslautern, West Germany, Sept., 1981, pages 23-36.

9. B. T. Hailpern and S. Owicki. Verifying network protocols using temporal logic. Technical Report 192, Computer Systems Laboratory, Stanford University, June, 1980.

10. D. Harel. *First-Order Dynamic Logic.* Springer-Verlag, Berlin, 1979. No. 68 of *Lecture Notes in Computer Science.*

11. D. Harel, D. Kozen, and R. Parikh. Process logic: Expressiveness, decidability, completeness. 21-th Annual Symposium on Foundations of Computer Science, Syracuse, New York, Oct., 1980, pages 129-142.

12. G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen and Co., Ltd., London, 1968.

13. L. Lamport. Specifying concurrent program modules. Opus 60, Computer Science Laboratory, SRI International, June, 1981.

14. S. Leinwand and T. Lamdan. Algebraic analysis of nondeterministic behavior. Proceedings of the 17-th Design Automation Conference, Minneapolis, June, 1980, pages 483-493.

15. T. M. McWilliams. Verification of timing constraints on large digital systems. Proceedings of the 17-th Design Automation Conference, Minneapolis, June, 1980, pages 139–147.

16. Y. Malachi and S. S. Owicki. Temporal specifications of self-timed systems. In H.T. Kung, B. Sproul, and G. Steele, editors, *VLSI Systems and Computations,* pages 203-212, Computer Science Press, Inc., Rockville, Maryland, 1981.

17. Z. Manna and B. Moszkowski. Temporal logic as a programming language, forthcoming.

18. Z. Manna and A. Pnueli. Verification of concurrent programs: The temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science,* pages 215-273, Academic Press, New York, 1981.

19. P. Meinen. Formal semantic description of register transfer language elements and mechanized simulator construction. Proceedings of the IFIP 4-th International Symposium on Computer Hardware Description Languages and their Applications, Palo Alto, California, Oct., 1979, pages 69-74.

20. B. Moszkowski. A temporal logic for multi-level reasoning about hardware. Proceedings of the IFIP 6- th International Symposium on Computer Hardware Description Languages and their Applications, Pittsburgh, Pennsylvania, May, 1983.

21. B. Moszkowski. *Reasoning about Digital Circuits.* Ph.D. Thesis, Department of Computer Science, Stanford University, forthcoming.
22. A. C. Parker and J. J. Wallace. "SLIDE: An I/O hardware description language." *IEEE Transactions on Computers* C-30, 6 (June 1981), 423-439.
23. D. A. Patterson. "Strum: Structured microprogram development system for correct firmware." *IEEE Transactions on Computers C-25,* 10 (Oct. 1976), 974-985.
24. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. 17-th IEEE Symposium on Foundations of Computer Science, Houston, Texas, Oct., 1976, pages 109-121.
25. N. Rescher and A. Urquart. *Temporal Logic.* Springer-Verlag, New York, 1971.
26. R. L. Schwartz and P. M. Melliar-Smith. Temporal logic specification of distributed systems. Proceedings of the 2-nd International Conference on Distributed Computing Systems, Paris, France, April, 1981, pages 446-454.
27. R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher-level temporal reasoning: language definition and examples. Technical Report CSL-138, Computer Science Laboratory, SRI International, Feb., 1983.
28. A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. Proceedings of the 14-th Annual ACM Symposium on Theory of Computing, San Francisco, California, May, 1982, pages 159-168.
29. S. Y. H. Su, C. Huang, and P. Y. K. Fu. A new multi-level hardware design language (LALSD II) and translator. Proceedings of the IFIP 5-th International Conference on Computer Hardware Description Languages and their Applications, Kaiserslautern, West Germany, Sept., 1981, pages 155–169.
30. A. D. Talantsev, "On the analysis and synthesis of certain electrical circuits by means of special logical operators." *Automation and Remote Control 20,* 1959, pages 874-883.
31. T. Wagner. *Hardware Verification.* Ph.D. Thesis, Department of Computer Science, Stanford University, 1977.