Exploiting the Potential of Diagrams in Guiding Hardware Reasoning

Kathi Fisler*
Department of Computer Science
Lindley Hall 215
Indiana University
Bloomington, IN 47405
kfisler@cs.indiana.edu

August 26, 1997

Abstract

Formal methods promises designers increased assurance in and understanding of their designs. Assurance is gained via proof; understanding is gained via the construction of proof. Researchers have developed powerful proof techniques; they have not focused sufficiently on creating tools to support reasoning. As a result, formal methods fails to attain its full potential. We argue that by formalizing the notations provided by diagrammatic representations, we can build tools that provide support for both proof and reasoning, thereby making formal methods more widely applicable by designers.

1 Introduction

> Formal methods offer much more to computer science than just "proofs of correctness" for programs and digital circuits, however. Many of the problems in software and hardware design are due to imprecision, ambiguity, incompleteness, misunderstanding, and just plain mistakes in the statement of top-level requirements, in the description of intermediate designs, or in the specification of components and interfaces.
>
> John Rushby [16]

Desire for proofs of correctness of systems spawned the research area known as "formal methods". Today's systems are of sufficient complexity that testing is infeasible, both computationally and financially. As an alternative, formal methods promotes mathematical analysis of a system as a means of locating inconsistencies and other design errors. Techniques used can range from writing system descriptions in a formal notation to verification that the designed system satisfies a particular behavioral specification. A good general introduction to formal methods appears in [16].

Ideally, using formal methods increases our assurance in and understanding of our designs. Assurance results from proof, while understanding results from the process of producing the proof. Successful use of formal methods therefore requires powerful proof techniques and clear logical notations. The verification research community has paid considerable attention to the former.

*Research supported by AT&T Bell Laboratories under the PhD Fellowship program.

Current techniques, many of which can be fully automated, handle sufficiently complex systems that formal methods are now being adopted (albeit slowly) in industry. In our drive to provide powerful proof methods, however, we have overlooked the latter requirement. Research has focused on proof without paying sufficient attention to reasoning. Current tools are often criticized as too hard to use, despite their computational power. Most designers, not having been trained as logicians, find the methodologies and notations very unnatural. Industrial sites starting out with formal methods must often rely on external verification professionals to help them use these tools effectively [12]. Tools that are not supportive of reasoning therefore fail to provide the full benefits of formal methods. We can augment our current methodologies to address this problem, but we first need to understand reasoning and its role in hardware design.

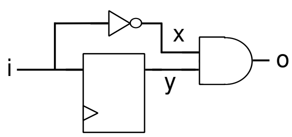
Barwise and Etchemendy [3] view valid reasoning as "the exploration of a space of possibilities" defined by the given information and the desired reasoning task. Under this definition, the more clearly a representation system allows for the exploration of this space, the more naturally reasoning can be conducted within this representation. They note that reasoning is a *heterogeneous* activity — people use multiple representations of information while reasoning, and those representations are often non-sentential forms such as diagrams. This is consistent with what occurs in hardware reasoning, in which a combination of state machines, circuit diagrams, timing diagrams, and sentential languages (such as VHDL) are often used.

We believe that diagrams naturally support such exploration, and therefore have the potential to bridge the proof-versus-reasoning gap. Diagrams are usually cast aside as informal notations, but they can be made rigorous. Some people view diagrams as too specialized to be suitable for formal methods; certainly, if we were to construct a tool out of only one style of diagram, this would be a legitimate concern. The key to using diagrammatic representations effectively is to use different styles simultaneously. Multiple representations of information interact formally in *heterogeneous logics*, an introduction to which appears in Barwise and Etchemendy's paper in this volume [4].

We have developed a heterogeneous hardware logic that encompasses diagrammatic and sentential representations. Section 2 develops a simple example to illustrate hardware reasoning with diagrams. Section 3 defines our proposed logic; it supports circuit diagrams, timing diagrams, a variant of state machines, and second-order logic. Section 4 contains additional examples, intended to illustrate how the logic supports various styles of hardware reasoning.

2 Contrasting Diagrammatic and Sentential Representations

Consider a simple physical device: a single pulser (SP). A single pulser converts each input pulse of arbitrary but finite duration into an output pulse of unit duration. There are many possible implementations of a single pulser; we propose one which generates its output pulse in the clock cycle following the fall of the input. Two views of the implementation, one diagrammatic and one sentential, can be given as follows:



$$SP(I,O) \equiv \exists x \exists y : delay(I,x) \wedge not(I,y) \wedge and(y,x,O)$$
$$delay(I,O) \equiv \forall t : O(t+1) = I(t)$$
$$not(I,O) \equiv \forall t : O(t) = 1 - I(t)$$
$$and(I_1,I_2,O) \equiv \forall t : O(t) = I_1(t) \times I_2(t)$$

Notice that, up to a level of abstraction that ignores the lengths of the lines in the diagram, both representations contain the same information. However, the presentation of that information is crucial to how easily we can reason about it. For example, suppose we want to determine whether the output of the circuit can ever be high when the input is high. Reasoning about this on the schematic is straightforward: looking at the diagram, we see that the input value is inverted and passed to an **and** gate which computes the output value. Given that a high **and** gate output requires two high inputs, it is easy to conclude that the input cannot be high when the output is to be high.

We will now attempt to reason about the same question using the sentential representation. Assuming the input is high (where high corresponds to numeric value 1 and logical true), we can rewrite (either in our heads or on paper) the definitions to determine that both the *delay* predicate and the *not* predicate have high as their first argument. This in turn produces a value for the second argument of the *not* predicate, binding y to 0. We can now replace occurrences of y with 0, from which we conclude that the output of the **and** gate is 0. Returning to the definition of SP, we see that the output of the **and** gate is the output of SP; again, we conclude that the input cannot be high when the output is high.

Although we arrive at the same conclusion using each representation, one could reasonably argue that reasoning on the diagram is clearer than reasoning on the sentences. In this case, we suggest that the qualitative difference lies in how each representation maintains the connections between the components of the circuit. The diagram maintains the connections explicitly; in the sentential representation, the user has to mentally connect the components via the common wire names and concentrate on rewriting values based on those connections. The diagram frees the user from having to reconstruct connection information, thereby allowing the user to concentrate on the reasoning rather than the representation. This is a very small example, yet it illustrates our point nicely: using diagrams for reasoning about circuit and value problems is advantageous.

Diagrams can also play a role in the specification and verification of systems. Timing diagrams are becoming a more popular notation for expressing behavioral specifications [17] [10], presumably because people find them clearer to use than formalisms such as temporal logic. To contrast methods, we present the behavioral specification of the single pulser in three representations: second-order logic, temporal logic, and timing diagrams. In each case, we want to specify that the pulser produces an output pulse for each input pulse, it produces only one output pulse for each input pulse, and all output pulses are of unit duration.

Higher-Order Logic:

$$\begin{aligned} spec1(i,O) = (\forall n, m : &Pulse(i,n,m) \supset \\ &\exists k : n \leq k \wedge k \leq m \wedge O(k) = 1 \wedge \\ &\quad (\forall j : (n \leq j \wedge j \leq m \wedge O(j) = 1 \supset j = k)) \end{aligned}$$

$$\begin{aligned} spec2(i,O) = (\forall k : O(k) = 1 \supset \\ &SinglePulse(O,k) \wedge \\ &(\exists n, m : n \leq k \wedge k \leq m \wedge Pulse(i,n,m))) \end{aligned}$$

$$\begin{aligned} Pulse(f,n,m) = (&n < m \wedge f(n-1) = 0 \ \wedge f(m) = 0 \wedge \\ &(\forall t : (n \leq t \wedge t < m \supset f(t) = 1))) \end{aligned}$$

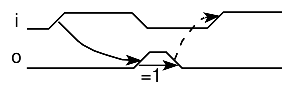
$$SinglePulse(O,k) = O(k) \wedge \neg O(k-1) \wedge \neg O(k+1)$$

Temporal Logic:

$$\text{rising\_edge} \equiv \neg i \wedge \bigcirc i$$
$$\Box\,(\text{rising\_edge} \rightarrow \bigcirc(\neg \text{rising\_edge}\ \mathcal{U}\ o))$$

$\Box\,(\text{rising\_edge} \rightarrow \Diamond o)$
$\Box\,(o \rightarrow \bigcirc(\neg o))$
$\Box\,(o \rightarrow \bigcirc(\neg o\,\mathcal{U}\ \text{rising\_edge}))$

Timing Diagrams:

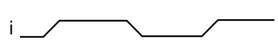


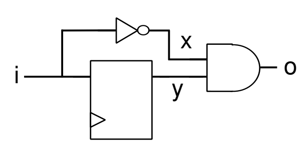
The temporal logic representation uses the usual operators: next ($\bigcirc$), henceforth($\Box$), eventually ($\Diamond$), and until ($\mathcal{U}$). The timing diagram notations will be explained in detail in section 3.4. In this timing diagram, the dashed arrow indicates that any appearance of the second event must be preceded by the first (safety). The solid arrow indicates that the edge is not only safe, but in addition, any occurrence of the first event must eventually be followed by the second event (liveness). The $= 1$ constrains the amount of time (in this case, one clock cycle) that must elapse between the two events.

We claim that the timing diagram is the clearest of these behavioral specifications for purposes of human reasoning. Specifications need to be easily understandable since systems are often built based on a designer's interpretation of them. The meaning of neither sentential specification is immediately clear, despite the fact that each is written in a well-known logical notation. In fact, the average person might construct a diagrammatic depiction of the sentential specifications in the process of understanding their full meanings. As in the case of the circuit diagram example, the important information — here, the relationship between timing events — is made more explicit by the diagram than by the other two representations. Timing diagrams therefore seem a good candidate for expressing event-based behavioral specifications.

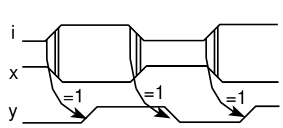
To illustrate heterogeneous reasoning, we complete the single pulser example by proving diagrammatically that the proposed circuit diagram satisfies the specification expressed in the timing diagram. The steps taken here, although appearing informal, are consequences of inference rules in the logic; these rules are presented formally in section 3.6.

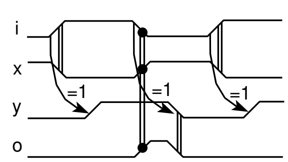
We start by assuming the implementation shown above and an input pulse of unspecified duration on i:



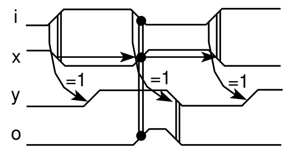


The circuit indicates the functional relationship between signals i, y, and x; the timing diagram for i can be extended to reflect these relationships; the resulting diagram appears below on the left. Given the waveforms for y and x, the diagram can be further extended to display information about signal o as in the diagram on the right:

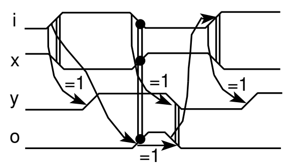




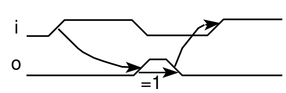
Looking at the specification, one of our goals is to relate the events on i and o. We can relate the rising edge on i to the rising edge on o based on an implicit edge between each pair of events on signal x. Another implicit edge on x allows us to relate the falling edge on o and the second rising edge on i. The top diagram shows the implicit edges and the lower two diagrams the inferences on signals i and o.



Comparing the specification we are trying to prove with our current derived timing diagram, we note that we are missing constraints between the rising and falling events on o. We note that each of these events is synchronous with another event and that an edge denoting duration one exists between the other two events. We therefore add an edge to obtain the following diagram:



The diagram now contains quite a bit of information, much of which has been subsumed by inference steps. We can always remove information from the timing diagram (though doing so might weaken the information content):



Note that this diagram differs from the desired diagram only in the style of the arrow relating the rising edge on o to the rising edge on i. Our informal definition of the edge types indicated that the solid arrow subsumes the dashed arrow; we can weaken the existing arrow to a dashed arrow, thus completing the proof.

The purpose of this example was not to advocate this style of deductive hardware verification; automatic verification techniques are capable of handling large classes of circuits without the need for such low level human intervention. Our intent was to demonstrate that there is a formal structure to how we reason with diagrams and that that structure lends a certain degree of natural clarity to the reasoning process.

What remains is to formalize diagrammatic reasoning methods like the ones used above in a logic suitable for hardware design. That task is the focus of the next section. Although the focus of this section was on the merits of diagrammatic representations, we do not mean to suggest that traditional, sentential representations have no role to play. Indeed, we believe that for a hardware reasoning system to be suitably flexible, sentential as well as diagrammatic representations need to be included. We suspect that sentential representations will be particularly crucial for mathematical reasoning, although we save exploration of that issue for later research.

3 Heterogeneous Hardware Logic

Our logic supports four representations: circuit diagrams, timing diagrams, algorithmic state machines (ASMs), and higher-order sentential logic. The diagrammatic portion of heterogeneous hardware logic was first presented in [8]. This paper redefines the syntax and semantics for timing diagrams originally presented in [8]; in addition, the sentential portion of the logic is formally presented for the first time. The circuit diagram and ASM portions of the logic are unchanged from [8], but are provided here for reference. We first define a model of physical devices, which will serve as a common semantic basis for the four syntactic representations. The syntax and semantics of each representation is presented in turn in subsections 3.2 through 3.5. A discussion of logical consequence and presentation of inference rules are provided in section 3.6.

3.1 Physical Devices

Our model of physical hardware is defined in two stages. First, we capture the structural aspects of a device along with its interface with the external environment. Later, we augment this model so that it can express how a device behaves over time while interacting with the environment. We assume that wires in devices can carry values in $\{0, 1\}$. Ports are primitive objects providing connection points for wires. We choose to associate voltage values with ports rather than wires; the term *assignment* will refer to a total function from a set of ports to the set $\{0, 1\}$.

We assume that devices are composed only of wires and *basic components*: binary and and or gates, inverters, and unit delay elements. Specifically, a basic component is a tuple $\langle I, O, F, D\rangle$ where I and O are disjoint sets of input ports and output ports, F is a function which assigns each $p \in O$ a function F_p from assignments on I into $\{0, 1\}$; and D is a function from O to the non-negative integers indicating the delay of the component. We assume that delay elements have $d = 1$ and all other basic components have $d = 0$.

An *abstract device* is a 5-tuple $D = \langle I, O, B, W, c\rangle$ capturing the structure of a component where I and O are disjoint sets of ports providing the external input and output interface to D, B is a set of basic components, W is a set of wires and c is a wiring function from W to sets of ports. We assume that all ports in a device are distinct and that the sets $c(w)$ partition the ports of D. Any ports in $I \cup O$ are called *interface* ports; all other ports are *internal*.

Although any abstract device corresponds to a piece of physical hardware, we are only interested in considering those that meet certain well-formedness conditions. In defining those conditions, we need to be able to talk about paths between ports within a device. Given an abstract device D, there is a *connecting step* from port p_i to port p_j, denoted $p_i \leadsto p_j$, iff either p_i and p_j are respectively an input and an output port to some basic component, or p_i is an internal output port or input interface port, p_j is an internal input port or output interface port, and $\{p_i, p_j\} \subseteq c(w)$ for some wire w. A finite transitive chain of connecting steps forms a *connecting path*, denoted $p_0 \leadsto^* p_n$; a device contains a connecting cycle if $p \leadsto^* p$ for some port p. We call a device *well-connected* if treating the basic components as nodes and the wiring function as giving rise to edges yields a connected graph, if for each internal port p in D there exists a connecting path from p to an element of O, and if every connecting cycle in D passes through a delay element.

In order to consider an abstract device in mid-computation, we need to know the values on the ports of the device. A tuple $\langle D, i\rangle$ consisting of an abstract device and an assignment for its ports will be called a *concrete device*. We require that all assignments included as part of a concrete device are consistent with the structure of the device. That is, given $\langle D, i\rangle$, for all gates g in D, the value in i on the output port of g is consistent with the values in i for the input ports of g and the function associated with g. The set of all possible assignments to the delay output ports in a device forms its *possible states*. A concrete device is well-connected if the abstract device it contains is well-connected.

We are interested in determining when two devices exhibit the same external behavior. This requires that we be able to operate our devices over time. Given a well-connected concrete device $C = \langle D, i\rangle$ and an assignment a to the input ports of D, there is a unique *derived assignment* i' such that $\langle D, i'\rangle$ is a concrete device where i' is defined as follows: if p is an interface input port of D, then $i'(p) = a(p)$, if p is an output port to a delay element with input port p_{in} then $i'(p) = i(p_{\text{in}})$, if p is an output port of some gate g then $i'(p)$ is F_p applied to the restriction of i' to the input ports of g, and if p is any other port $i'(p) = i'(q)$ where q is the unique internal output port or input interface port wired to p.[1] Concrete device $\langle D, i'\rangle$ is said to *follow from* C given a, where i' is the derived assignment from $\langle D, i\rangle$ and a.

The term *assignment sequence* refers to a sequence of assignments $i_1, i_2, \ldots, i_k$ to the interface input ports of a device. Given concrete device $C = \langle D, i\rangle$ and assignment sequence $\langle i_1, i_2, \ldots, i_k\rangle$, a *run* of C is a sequence $r = \langle C_0, C_1, \ldots, C_k\rangle$ of concrete devices such that $C_0 = C$ and for each $1 \le j \le k$, C_j is the concrete device that follows from C_{j-1} given i_j. The *output* of a concrete device $\langle D, i\rangle$ is the restriction of i to the interface output ports O of D. The output of a run $r = \langle C_0, C_1, \ldots, C_k\rangle$ of a concrete device is a sequence $\langle O_0, O_1, \ldots, O_k\rangle$ where each O_i is the output of C_i. The state of devices and runs are similarly defined by restricting assignments to the delay output ports. For port p, the *run-valuation* of p is the restriction of these assignments to p.

Concrete devices $C_1 = \langle D_1, i_1\rangle$ and $C_2 = \langle D_2, i_2\rangle$ are *behaviorally equivalent* if D_1 and D_2 have the same sets of input and output interface ports and for every assignment sequence a for D_1, the output of the run of C_1 under a is the same as the output of the run of C_2 under a. Abstract devices D_1 and D_2 are behaviorally equivalent if for every $\langle D_1, i_1\rangle$ there exists an i_2 such that $\langle D_1, i_1\rangle$ and $\langle D_2, i_2\rangle$ are behaviorally equivalent, and vice versa.

3.2 Circuit Diagrams

In this logic, we consider circuit diagrams composed of icons representing binary and and or gates, unit delay elements, inverters, and wires. We use a set-theoretic model to capture the syntactic information contained in a given circuit diagram. In this model, we represent a wire line as an ordered pair of the points it connects; a binary gate icon is modelled as an ordered triple $\langle x, y, z\rangle$, indicating that the icon connects x and y on the left, in that order, from top to bottom, with z on the right. Unary icons are similarly represented with an ordered pair.

Specifically, a *circuit sketch* s is a tuple $\langle P, I, O, W, N, D, A, R\rangle$ where P is a set of objects called the *connection points* of s, I and O are disjoint subsets of P called the input points and output points of s, W, N, and D are disjoint subsets of $P \times P$ called the *wire lines*, *negation icons*, and *delay icons*, and A and R are disjoint subsets of $P \times P \times P$ called the and *gate icons* and or *gate icons*. A wire $w = \langle p_a, p_b\rangle$ is *branch-free* iff w is the unique wire with p_a as its first component;

[1] The proof of uniqueness appears in [8].

otherwise w is a *branching* wire. We use the term circuit *sketch* as opposed to circuit *diagram* so that we may reserve the latter term for only those diagrams which are well-formed. A formal definition of this term will be presented shortly.

Definition 1 Let s be a circuit sketch and D be an abstract device.

1. A *depiction map* from s to D is an injective function ϕ from the connection points of s into the ports of D such that for all $p \in s_P$;
   (a) $p \in s_I \to \phi(p) \in D_I$.
   (b) $p \in s_O \to \phi(p) \in D_O$.
   (c) If $l = \langle p_a, p_b \rangle$ is a wire line of s then $\phi(p_a)$ and $\phi(p_b)$ are wired together by some wire $w \in D_W$; $\phi(p_a)$ must be an input interface port or internal output port and $\phi(p_b)$ must be an output interface port or internal input port.
   (d) If $n = \langle p_a, p_b \rangle$ is a negation icon of s then $\phi(p_a)$ is connected to the input port and $\phi(p_b)$ connected to the output port of some inverter in D_G.
   (e) If $d = \langle p_a, p_b \rangle$ is a delay icon of s then $\phi(p_a)$ is connected to the input port and $\phi(p_b)$ connected to the output port of some delay element in D_R.
   (f) If $g = \langle p_a, p_b, p_c \rangle$ is an and-gate icon of s then $\phi(p_a)$ and $\phi(p_b)$ are connected to the input ports and $\phi(p_b)$ connected to the output port of some and-gate in D_G.
   (g) If $g = \langle p_a, p_b, p_c \rangle$ is an or-gate icon of s then $\phi(p_a)$ and $\phi(p_b)$ are connected to the input ports and $\phi(p_b)$ connected to the output port of some or-gate in D_G.
2. D is a *structural implementation of* s if there is a surjective depiction map from s to D and the converse of requirements 1c through 1g in the definition of a depiction map holds. This is written $D \models_s s$.
3. D is a *behavioral implementation of* s if D is behaviorally equivalent to some device D' which is a structural implementation of s. This is written $D \models_b s$.

Lemma 1 *For any device D there exists a circuit sketch C that is unique up to isomorphism on circuit sketches such that $D \models_s C$. For any circuit sketch C there exists a device D that is unique up to isomorphism on devices such that $D \models_s C$.*

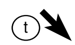
We use this lemma to make a deferred definition: a *circuit diagram* is any circuit sketch s for which the device D such that $D \models_s s$ is well-connected. For the remainder of this work, we assume we are dealing only with circuit diagrams, as opposed to circuit sketches.

3.3 ASM Charts

ASM charts are a variant of state machines that combine the traditional Mealy and Moore machines. They have an appearance reminiscent of flow-charts: rectangles denote states, diamonds represent conditional branches, and ovals represent conditional (Mealy) outputs. Moore outputs are designated by assigning a variable a value (either T or F) within a state rectangle. Each conditional branch diamond contains the name of a single signal to be tested and has two paths leaving it, one labeled T and one labeled F (where T and F are relative to the value of the signal tested in the diamond). Each conditional oval contains one or more variable names to be assigned T when control reaches the oval. Examples of ASM charts appear in figure 4 (page 17) and more extensive discussion appears in [14]. As in the section on circuit diagrams, we will reserve the term *ASM chart* for what we wish to consider well-formed diagrams, using the term *ASM graph* for the general case.

An *ASM graph* g is a tuple $\langle S, B, O, N, R, P \rangle$ where S is a set of state objects, B is a set of conditional branch objects and O is a set of conditional output objects such that S, B, and O are all pairwise disjoint. We will refer to the union of these sets as the *objects* of g. N is a set of signal names. R is a subset of $S \times S \times \mathcal{P}(N) \times \mathcal{P}(N)$ called the *next state transitions* of g.[2] The first and second elements of these tuples are called the *source state* and *target state*, respectively. The third and fourth elements are called the *true conditions* and the *false conditions*, respectively. Finally, P is a subset of $S \times N \times U \times \mathcal{P}(N) \times \mathcal{P}(N)$ called the *output conditions* of g. The first two elements are called the *asserting state* and *asserted variable*, respectively. The third element is called the *assignment value* and is a member $\{T, F\}$. The last two elements are called the *true conditions* and the *false conditions*, respectively. We will use the term *external signal* for those elements of N that appear in the true or false conditions of some element of $R \cup P$ but are not the asserted variable for any output condition; non-external signals are classified as *internal*.

In order to relate ASM graphs to devices, we need to be able to talk about their computational behavior. A *signal-value assignment* for ASM graph is a function from the names N to the set $\{T, F\}$. An *external signal-value assignment* is the restriction of a signal-value assignment to the external signals. $\langle g, s \rangle$ is an *executing ASM graph* where g is an ASM graph and s is some state in g. The following symbol, when placed near state s in ASM graph g, denotes that $\langle g, s \rangle$ is executing at time t.



Given executing ASM graph $\langle g, s \rangle$ and a signal-value assignment i for g, a next-state transition $\langle t_s, t_t, c_t, c_f \rangle \in R$ is *satisfied by s and i* if $t_s = s$, $i(n) = T$ for all names in c_t, and $i(n) = F$ for all names in c_f. If there is exactly one such transition satisfied by s and i, this transition is called the *next state of g under s and i*. We say that output condition $\langle t_s, n, u, c_t, c_f \rangle \in P$ is *satisfied by s and i* if $t_s = s$, $i(v) = T$ for all names v in c_t, and $i(v) = F$ for all names v in c_f. Let i' be the unique signal-value assignment such that for all output conditions $\langle t_s, n, u, c_t, c_f \rangle$ satisfied by s and i, $i'(n) = u$, $i'(x) = i(x)$ if x is an external signal and $i'(x) = F$ for all other signals x. i' is called the *signal update of g under s and i*.[3]

An ASM graph g is *deterministic* if no two next state transitions are satisfied by the same state s and signal-value assignment i; it is called *transitionally complete* if for all states s and all signal-value assignments i there exists a next-state transition that is satisfied by s and i. An ASM graph that is both deterministic and transitionally complete will be called an *ASM chart*. This corresponds to a well-formedness definition on ASM graphs.

We have established sufficient framework to discuss when a given ASM chart describes a given physical device and when a given device implements the algorithm depicted in an ASM chart. Given ASM chart g and abstract device D, a *state map* from g to D is a function from the states S of g to the possible states of D. Function ϕ from the signal names of g to the ports of D is called a *signal map* iff ϕ maps each external signal in g to an input interface port of D and each internal signal in g to an internal output port in D. ϕ is called a *complete signal map* iff it is a signal map with every interface port (input and output) of D in its co-domain.

[2] The notation $\mathcal{P}(N)$ represents the powerset of N.

[3] We are using F as the global default value for signals, though defaults could be assigned in various other ways.

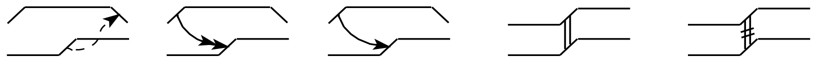


Figure 1: Types of edges that can be drawn on timing diagrams. Safe edges require the source event to occur before or simultaneously with the target event. Live edges require the target event to occur after the source event. Combined edges are used when safe and live edges are needed between the same two events. Simultaneous edges require the events to happen concurrently, while conflict edges do not permit the events to occur simultaneously. These notations and definitions are taken from the work of Schlör and Damm[17]. In order to draw simultaneous edges, it is often necessary to cross events that should not be synchronized. In the event that a synchronization line applies to only some of the events it crosses, those events it relates will be attached to the line using a dark circle.

We will establish relationships between ASM graphs and devices by simulating each on the same inputs and seeing how closely the state transitions and output behaviors correspond. Doing this requires that we know when a signal-value assignment and a port assignment are reflecting the same values. Given a signal map ϕ and signal-value assignment i, assignment a for D is *compatible* with ϕ and i iff $a(p) = i(\phi^{-1}(p))$ for all ports p in the co-domain of ϕ. A state map ϕ_s and a signal map ϕ_n are said to be *feasible for g and D* if for all signal-value assignments i for g and all states s in g there exists an assignment a for D which is compatible with ϕ_n and i and reflects state $\phi_s(s)$ such that:

1. If s' is the next state of g under s and i, then $\phi_s(s')$ is the next state of D under $\phi_s(s)$ and a.
2. If i' is the signal update of g under s and i, then assignment a' derived from $\langle D, a\rangle$ is compatible with i'.

If ϕ_s and ϕ_n are feasible for g and D and the converse of requirement 1 holds for all i and s, we say that ϕ_s and ϕ_n *capture* g and D.

As in the section on circuit diagrams, we now define three relationships between ASM graphs and devices that capture the various granularities of relationships between them.

Definition 2

1. g *describes* D if there exists a state map and a signal map that are feasible for g and D.
2. D is a *structural implementation* of g iff there exists a surjective state map and a complete signal map that capture g and D. This is written $D \models_s g$.
3. D is a *behavioral implementation of g* if D is behaviorally equivalent to some device D' which is a structural implementation of g. This is written $D \models_b g$.

3.4 Timing Diagrams

The timing diagram syntax and semantics originally presented in [8] is too restricted to be able to represent general timing relationships. We have updated our syntax and semantics to follow the much more flexible system of Schlör and Damm [17]. A timing diagram is a collection of individual waveforms whose events are related by a series of edges between them. The types and notations for edges are given in figure 1 and include notations for safety requirements, liveness requirements, and coincidence requirements. A timing level is an element of the set {high, low, don't-care, rising, falling}. We define a *timing pattern* as a pair $\langle s, c\rangle$ where s is a sequence of timing levels and c is a color used to indicate the role of the signal in the system as one of input, output, or internal. A timing pattern is *well-formed* if high is never immediately followed by low or rising, low is never immediately followed by high or falling, rising is never immediately followed by rising or falling, and falling is never immediately followed by rising or falling. A *timing event* is defined as a tuple $\langle p, i\rangle$ where p is a timing pattern and i is an index into p_s such that i is no larger than the length of p_s. We will write $p(i)$ to refer to the timing level in p_s at time i.

Given a collection C of timing patterns, an *edge* on those patterns is a pair of the form $\langle e_1, e_2\rangle$ where e_1 and e_2 are timing events on patterns in C. An edge may be annotated with a duration marker consisting of a positive integer or integer variable and one of the symbols $+, -, =$; these markers specify bounds on the amount of time that may pass between the two events. A partial function mapping edges to duration markers is a *duration mapping*. We consider a *timing diagram* to be a tuple $\langle N, P, E_S, E_L, E_M, E_C, \phi\rangle$ in which N is a set of names, P is a function from N to timing patterns, E_S is a set of safe edges, E_L is a set of live edges, E_M is a set of simultaneous edges, E_C is a set of conflict edges, and ϕ is a duration mapping on $E_S \cup E_L$.

In comparing devices and timing diagrams, a timing value v_t and a numeric device value v_d are said to *correspond* if v_t is high and $v_d = 1$ or if v_t is low and $v_d = 0$. If v_t is don't-care, then it corresponds to any value of v_d. We relate the signal names in a timing diagram T to the ports of a device D using an injective function called a *waveform map* in which input signals in T map to input interface ports of D, output signals in T map output interface ports of D, and internal signals in T map to internal output ports of D. The *defining indices* of a timing pattern are those that map to values in {high, low, don't-care}.

Given a run $R = \langle D, a_0\rangle, \langle D, a_1\rangle, \ldots, \langle D, a_n\rangle$ of a concrete device, a timing event $\langle p, i\rangle$, and a port s in D, $\langle p, i\rangle$ *matches* s in C_j iff if i is a defining index of p then $p(i)$ corresponds to $a_j(s)$, if $p(i) =$ rising then $a_j(s) = 0$ and $a_{j+1}(s) = 1$, and if $p(i) =$ falling then $a_j(s) = 1$ and $a_{j+1}(s) = 0$. A timing diagram T is *valid* for R under waveform map ϕ between T and D iff

1. For every signal $s \in T$, there exists a monotonic function f from the indices of s to $\{0, \ldots, n\}$ such that for all defining indices i, the value of $\phi(s)$ in $R_{f(i)}$ corresponds to $s(i)$.[4]

2. For every $\langle\langle s_1, i_1\rangle, \langle s_2, i_2\rangle\rangle \in E_S$ and for all j, $0 \le j \le n$, if $\langle s_2, i_2\rangle$ matches $\phi(s_2)$ in R_j, then there must exist k, $0 \le k \le j$, such that $\langle s_1, i_1\rangle$ matches $\phi(s_1)$ in R_k.

3. For every $\langle\langle s_1, i_1\rangle, \langle s_2, i_2\rangle\rangle \in E_L$ and for all j, $0 \le j \le n$, if $\langle s_1, i_1\rangle$ matches $\phi(s_1)$ in R_j, then there must exist k, $j \le k \le n$, such that $\langle s_2, i_2\rangle$ matches $\phi(s_2)$ in R_k.

4. For every $\langle\langle s_1, i_1\rangle, \langle s_2, i_2\rangle\rangle \in E_M$, if $\langle s_1, i_1\rangle$ matches $\phi(s_1)$ in R_j, then $\langle s_2, i_2\rangle$ matches $\phi(s_2)$ in R_j.

5. For every $\langle\langle s_1, i_1\rangle, \langle s_2, i_2\rangle\rangle \in E_C$, if $\langle s_1, i_1\rangle$ matches $\phi(s_1)$ in R_j, then $\langle s_2, i_2\rangle$ does not match $\phi(s_2)$ in R_j.

Definition 3 Let T be a timing diagram and let D be a device.

1. T *describes* D, written $D \models T$, if there is a waveform map ϕ from T to D such that for all runs R of D, T is valid for R under ϕ.

[4] This function f is not necessarily unique.

2. D is a *structural implementation of* T if T describes D using a surjective waveform map. This is written $D \models_s T$.

3. D is a *behavioral implementation of* T if D is behaviorally equivalent to some device D' which is a structural implementation of T. This is written $D \models_b T$.

3.5 Sentential Logic

Our sentential logic is second-order logic augmented with arithmetic operations. We assume the existence of a sort N of natural numbers and a sort B of booleans with boolean constants $\mathbf{f}$ and $\mathbf{t}$. We also assume we have temporal variables $t_1, t_2, \ldots$, temporal constants $\overline{0}, \overline{1}, \ldots$ for each number k in N, and function constants $\overline{f}$ for every function from N to N. Given a concrete device C and an assignment sequence A, we associate with each port p in C a *port function constant* $\overline{p}$ from N to B; for each i up to the length of A, $\overline{p}(i)$ returns the value on p in device C_i in the run of C on A, and for all other i, $\overline{p}(i)$ returns $\mathbf{f}$. Port value 0 is equivalent to boolean constant $\mathbf{f}$ and port value 1 is equivalent to boolean constant $\mathbf{t}$. We will assume the existence of port function variables W, X, Y.

The class of *temporal terms* is the smallest class containing the temporal variables and constants and closed under the following operation: if t is a temporal term and $\overline{f}$ is a function from N to N, then $\overline{f}(t)$ is a temporal term. Given a function ϕ mapping temporal variables into N, we extend ϕ to a function mapping all temporal terms t into N in the obvious way by recursion on terms:

$$\phi(\overline{k}) = k$$
$$\phi(\overline{f}(t)) = f(\phi(t))$$

We also have a set L of *value terms*, defined as the smallest set containing $\mathbf{t}$, $\mathbf{f}$, $F(t)$ for each temporal term t and $F \in N \to B$, $X(t)$ for each temporal term t and port function variable X, and closed under the following: if $l_1, l_2 \in L$, then $\neg l_1$ and $(l_1 \wedge l_2)$ are in L. We assume that $\vee$ is defined as usual from $\neg$ and $\wedge$.

Given any function ϕ from temporal variables into N and port function variables into $N \to B$, and any value term l, we define $l[\phi]$ to be the boolean value that l takes on under the mapping ϕ. This is defined by recursion as follows:

$$\begin{aligned}
&\mathbf{t}[\phi] = \text{true}\\
&\mathbf{f}[\phi] = \text{false}\\
&F(t)[\phi] = F(\phi(t))\\
&X(t)[\phi] = \phi(X)(\phi(t))\\
&(\neg l)[\phi] = \neg(l[\phi])\\
&(l_1 \wedge l_2)[\phi] = (l_1[\phi] \wedge l_2[\phi])
\end{aligned}$$

The *atomic formulae* are the set of expressions of the form $t_1 \leq t_2$, $t_1 = t_2$ for temporal terms t_1 and t_2, and $l_1 = l_2$ for value terms l_1 and l_2. The set of formulae of our language is the smallest set containing the atomic formulae and closed under the following: if B_1 and B_2 are formulae, then so are $\neg B_1$, $(B_1 \wedge B_2)$, $\forall X B_1$, and $\forall t B_1$; $\vee$ and $\exists$ are assumed to be defined as usual from these connectives.

We define the semantics of formulae in terms of concrete devices and assignments to variables, as one would expect. For any formula E, we define $C, A \models E[\phi]$ recursively on E, where $C = \langle D, i \rangle$ is a concrete device and A is an assignment sequence of length k for D. ϕ maps temporal variables into temporal constants $\overline{0} \ldots \overline{k}$ and port function variables into port function constants. The notation $B_1[X/\overline{p}]$ denotes the substitution of $\overline{p}$ for X in B_1; $B_1[t/t_i]$ is defined analogously.

$C, A \not\models \mathbf{f}[\phi]$
$C, A \models \mathbf{t}[\phi]$
$C, A \models (t_1 = t_2)[\phi]$ iff $\phi(t_1) = \phi(t_2)$
$C, A \models (t_1 \leq t_2)[\phi]$ iff $\phi(t_1) \leq \phi(t_2)$
$C, A \models (l_1 = l_2)[\phi]$ iff $\phi(l_1) = \phi(l_2)$
$C, A \models \neg B_1[\phi]$ iff $C, A \not\models B_1[\phi]$
$C, A \models (B_1 \wedge B_2)[\phi]$ iff $C, A \models B_1[\phi]$ and $C, A \models B_2[\phi]$
$C, A \models (\forall X B_1)[\phi]$ iff $C, A \models (B_1[X/\overline{p}])[\phi]$
for each port function constant associated with a port in C
$C, A \models (\forall t B_1)[\phi]$ iff $C, A \models (B_1[t/t_c])[\phi]$
for each temporal constant $t_c \in \overline{0} \ldots \overline{k}$

Definition 4 Let $C = \langle D, i \rangle$ be a concrete device and E be a formula in our sentential language. E *describes* C, written $C \models E$, if for all assignment sequences A for D, there exists a function ϕ mapping port function variables into port function constants for C and temporal variables into temporal constants $\overline{0} \ldots \overline{k}$, where k is the length of A, such that $C, A \models E[\phi]$. We say that E *describes* D, written $D \models E$ if for all assignments i, $\langle D, i \rangle \models E$.

3.6 Rules of Inference and Methods of Proof

Although a general discussion of the theory of heterogeneous inference is out of the scope of this paper, our presentation is motivated by the work of Barwise and Etchemendy [1]. When stating these rules, we use the term *representation* as opposed to the more traditional term *formula* to avoid the sentential connotations associated with the latter. In general, a rule can be formulated to infer representation G from a set of representations S if it is the case that whenever a device D models every element of S, D also models G; in this case G is said to be a logical consequence of S. While this requirement does not dictate which of the modeling relationships (structural or behavioral) should be used in defining rules of inference, we use behavioral modeling to create rules between diagrams of the same type and structural modeling to create rules between diagrams of different types.

There are a number of inference rules for the full logic; only a subset are relevant to the examples presented in this paper. Rules are presented with their diagrammatic depictions where feasible. We do not present the standard inference rules of higher-order logic or the rules of arithmetic, but assume they are defined within the system.

We start by defining some general methods of proof. In these definitions, we consider a *proof* to be a sequence of representations such that each element of the sequence is either a single representation or another proof; each proof appearing as a complete element of a proof P is a *direct subproof* of P. Each proof starts with a series of representations called the *initial assumptions* of the proof. The *context* for an element of a proof is the set of elements that have preceded it in the proof. We will consider a proof to be *valid* if every element that is not an initial assumption is the logical consequence of its context. A *goal* is a representation that we wish to prove from the set of initial assumptions. The goal is satisfied if it is the last representation in a valid proof.

In the course of producing a proof, it is often the case that there are many possible cases that need to be considered in order to complete the next proof step; a simple example of this from sentential logic arises when working with disjunctions. In this case, it is common to consider each possibility in turn and then to prove that the desired goal is satisfied in each case. The following rule generalizes this notion of breaking into a set of cases:

Rule 1 (**Condition Exhaustive**) Let $C_1, \ldots, C_n$ be a set of direct subproofs of a proof P, let R be some representation, and let A be some property that can be associated with R. $C_1, \ldots, C_n$ are *exhaustive with respect to* A iff every context in which R has property A is subsumed by some C_i.

As an example of how this rule can be instantiated for a particular property, we define a rule for breaking into cases based on the states of an ASM chart that assert a particular variable.

Rule 2 (**Asserting States Exhaustive**) Let A be an ASM chart, let v be any signal in A, and let u be a truth value. Let S be the set of all states s such that $\langle s, v, u, c_t, c_f \rangle$ is an output condition in A. Direct subproofs $C_1, \ldots, C_n$ are *Asserting States Exhaustive* if each element of S is the state of A in the initial assumptions of some C_i.

Usually, our goal in splitting into cases is to derive some particular representation from each case so that the representation may be extracted from the cases and asserted at the level containing the subproofs. We express this in our logic using the following rule:

Rule 3 (**Merge**) Let P be a proof containing an exhaustive set of cases $C_1, \ldots, C_n$ and let R be a representation that is satisfied in every C_i, $1 \leq i \leq n$. R is true in P by the *Merge* rule.

We now turn to rules that are particular to the different types of representations. We begin with four rules related to ASM charts.

Rule 4 (**Asserting State**) Let A be an ASM chart, let v be any signal in A, and let u be a truth value such that there exists a unique state $s \in A$ that can be the asserting state for v with value u in the current context. If t is a time variable such that $v(t) = u$, then it follows by the rule of *Asserting State* that $\langle A, s \rangle$ is executing at time t.

Rule 5 (**Value in State**) Let $\langle A, s \rangle$ be executing at time t. Given output condition $\langle s, n, u, c_t, c_f \rangle$, if every element of c_t is true in the current context and every element of c_f is false in the current context, then $n(t) = u$ follows by *Value in State*.

Rule 6 (**State Transition**) Let $\langle A, s \rangle$ be executing at time t. Given next state transition $\langle s, s', c_t, c_f \rangle$, if every element of c_t is true in the current context and every element of c_f is false in the current context, then $\langle A, s' \rangle$ is executing at time $t+1$ by the rule of *State Transition*.

Rule 7 (**Looping**) Let $\langle A, s \rangle$ be executing at time t with a next-state transition $\langle s, s, c_t, c_f \rangle$ such that there is exactly one signal in $c_t \cup c_f$. Let v name this signal.

1. If $v \in c_t$, $\exists t_1 \; t_1 > t \wedge \neg v(t_1)$ is true in the current context, and t_1 is the smallest such time, then $\langle A, s' \rangle$ is executing at time $t_1 + 1$ under the *looping* rule, where $\langle s, s', \{\}, \{v\} \rangle$ is a next-state transition in A.

2. If $v \in c_f$, $\exists t_1 \; t_1 > t \wedge v(t_1)$ is true in the current context, and t_1 is the smallest such time, then $\langle A, s' \rangle$ is executing at time $t_1 + 1$ under the *looping* rule, where $\langle s, s', \{v\}, \{\} \rangle$ is a next-state transition in A.

We now present some rules associated with circuit diagrams. It is straightforward to define the rules of boolean algebra in terms of their associated circuit diagram representations; as a result, we give only one example of such a rule here, although the full set of boolean algebra rules are defined within the logic. We define one instance of the distributive law below; its diagrammatic representation appears in figure 2.

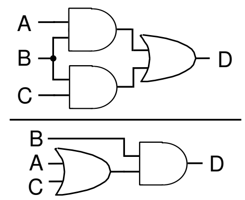


Figure 2: Inference rule corresponding to distributivity.

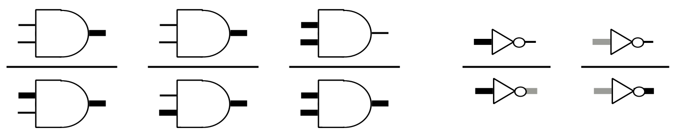


Figure 3: The simulation rules on circuit diagrams; high voltages are denoted in black and low voltages are denoted in grey. Thin wires are considered to have unknown/don't-care voltage. Other rules on and gates, such as those involving low voltages, can be derived from the existing and rules. Rules for or gates can be derived from both the and rules and the inverter rules.

Rule 8 (**Distributivity**) Given circuit diagram C with a branch-free wire w connecting the output of an or gate r to the input of an and gate a, circuit diagram C' is derived by means of *distributivity* by adding new and gates a_1 and a_2, adding new or gate r_1, wiring the output of r_1 to the output of a, wiring the non-w input to a to an input port of each of a_1 and a_2, wiring one input of r to the unused input port on a_1 and the other input of r to the unused input port on a_2, wiring the outputs of a_1 and a_2 to the input of r_1, and removing a, w, and r.

In addition to the boolean algebra rules, we define what can be thought of as simulation rules on circuit diagrams — rules that allow us to infer the propagation of voltage levels on wires across gates. These rules should be intuitively clear. A collection of these rules appears in figure 3; the formal definition of the first rule can be given as:

Rule 9 (**And High Output**) Let a be an and gate whose output voltage is known to be high. It follows that the voltage on either input must also be high.

We use a variety of timing diagram inference rules in this paper: some between timing diagrams and other timing diagrams, some between timing diagrams and circuit diagrams, and some between timing diagrams and sentential logic. The first two types of rules were used in the single pulser discussion in section 2. The rules are now presented formally; the reader is referred back to the single pulser discussion for examples of using the rules.

Rule 10 (**Edge Transitivity**) Let T be a timing diagram with edges $\langle e_1, e_2\rangle$ and $\langle e_2, e_3\rangle$ of the same type. Unless $\langle e_1, e_2\rangle$ and $\langle e_2, e_3\rangle$ are conflict edges, timing diagram T' is derived from T by means of *edge transitivity* by adding new edge $\langle e_1, e_3\rangle$ of the same type as $\langle e_1, e_2\rangle$ and $\langle e_2, e_3\rangle$.

Rule 11 (**Weakening**) Let T be a timing diagram. Timing diagram T' is derived from T by means of *weakening* by either removing an edge from E_S, E_L, E_M, or E_C, or by removing a timing pattern p from the range of P and removing all edges from E_S, E_L, E_M, and E_C containing an event on p.

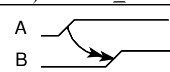
Rule 12 (**Time Equality**) Let T be a timing diagram with edges $\langle e_1, e_2\rangle, \langle e_3, e_4\rangle \in E_M$ and edge $\langle e_1, e_3\rangle$ of any type. Timing diagram T' is derived by means of *time equality* by adding edge $\langle e_2, e_4\rangle$ of the same type as $\langle e_1, e_3\rangle$. If $\langle e_1, e_3\rangle \in E_S \cup E_L$, then $\phi(\langle e_2, e_4\rangle) = \phi(\langle e_1, e_3\rangle)$.

The next definition gives an example of how we infer additional timing diagram information from a timing diagram and a circuit diagram. In this definition, we define the *flip* s' of a sequence s of timing levels such that $s'(i) =$ high if $s(i) =$ low, $s'(i) =$ low if $s(i) =$ high, $s'(i) =$ rising if $s(i) =$ falling, and $s'(i) =$ falling if $s(i) =$ rising.

Rule 13 (**Waveform Negation**) Let T be a timing diagram, let C be a circuit diagram, and let D be a device such that $D \models T$ and $D \models_s C$. Let $\langle p_a, p_b\rangle$ be an inverter in C such that T contains a timing pattern $p = \langle s, c\rangle$ for the signal name corresponding to p_a. Timing diagram T' follows by means of *waveform negation* by adding a new timing pattern $p' = \langle s', c'\rangle$ such that s' is the flip of s and c' is the color associated with the function of p_b in C, and for each i up to the length of s, $\langle\langle p, i\rangle, \langle p', i\rangle\rangle \in E_M$.

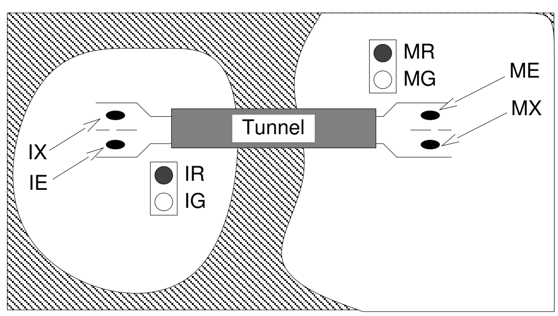
In the next section, we will use rules between timing diagrams and sentential logic. Unlike most of the other rules in the system, these are essentially translation rules. One example is the rule that produces a live edge from a sentential formula:

$$\frac{\forall t\ (\neg A(t) \wedge A(t+1) \rightarrow \exists t'\ t' \geq t \wedge \neg B(t') \wedge B(t'+1))}{}$$



4 The Island Traffic Light Controller

We are now ready to demonstrate the flexibility of the logic on an example. Assume that we want to design a controller for the traffic lights at a one lane tunnel connecting the mainland to a small island as pictured below. There is a traffic light at each end of the tunnel; there are also four sensors for detecting the presence of vehicles: one at tunnel entrance on the island side (IE), one at tunnel exit on the island side (IX), one at tunnel entrance on the mainland side (ME), and one at tunnel exit on the mainland side (MX).



In addition, there is a constraint that at most sixteen cars may be on the island at any time. We make the environmental assumptions that all cars are finite in length, that no car gets stuck in the tunnel, that cars do not exit the tunnel before entering the tunnel, and that cars do not leave the tunnel entrance without traveling through the tunnel.

The solution discussed here consists of three communicating controllers: one for the island lights, one for the mainland lights, and one tunnel controller that processes the requests for access issued

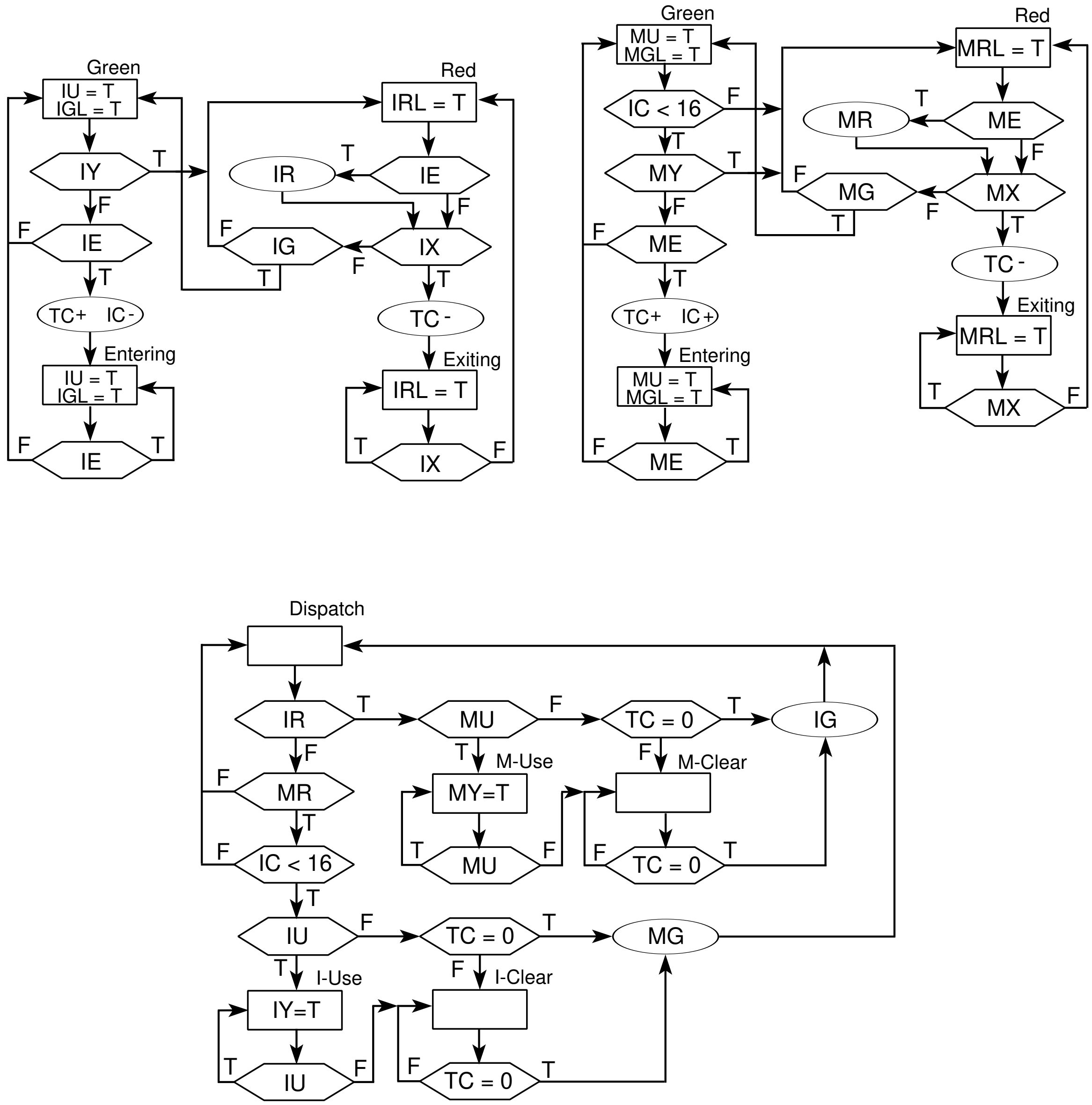


Figure 4: ASM charts for the Island Traffic Light Controller; the island light controller is on the top left, mainland light controller on the top right, and tunnel controller on the bottom. IGL and IRL are the green and red lights for the island, IU indicates that the island is using the tunnel, IR indicates that the island is requesting the tunnel, IY indicates that the island is being instructed to release control of the tunnel, and IG indicates that the island has been granted control of the tunnel; a similar set of signals has been defined for the mainland. TC is a count of the number of cars presently inside the tunnel and IC is a count of the number of cars presently on the island.

$$\forall t \exists t_1 \; t_1 > t \wedge \neg IE(t_1) \wedge \forall t_2 \; t_1 > t_2 \geq t \rightarrow IE(t_2)$$

$$\forall t \exists t_1 \; t_1 > t \wedge IE(t_1) \wedge \forall t_2 \; t_1 > t_2 \geq t \rightarrow \neg IE(t_2)$$

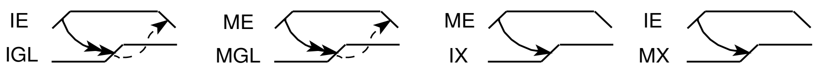
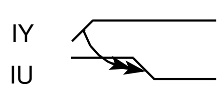


Figure 5: Representations of the environmental assumptions.

by the other two controllers. State machines depicting each of the three controllers are provided in figure 4. We would like to establish that our solution has at least the following properties:

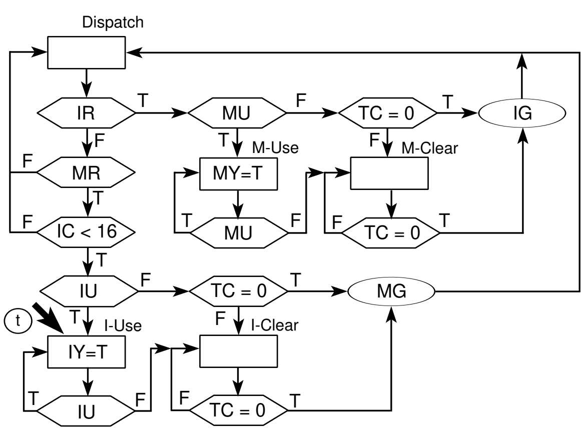
1. Cars never travel both directions in the tunnel at the same time.
2. Access to the tunnel is not granted until the tunnel is empty.
3. Lights don't turn green until access is granted.
4. The tunnel is used once granted.
5. Requests for the tunnel are eventually granted.
6. Once a car arrives at an entrance, the light at that entrance eventually turns green.
7. All commands to yield the tunnel are acknowledged by the island and mainland controllers.
8. There are never more than 16 cars on the island.
9. Counters are only changed once per car.
10. Counters do not move if increment and decrement signals are asserted simultaneously.

Some of these properties, such as 2 – 7, are natural candidates for finite-state verification techniques, others, such as 10, are easier to reason about once an implementation of the system is designed; we will demonstrate how our logic supports reasoning at each of these levels. As an example of verification at the state-machine level, consider the condition that all yields issued by the tunnel controller should eventually be acknowledged; many of the other conditions above could also be verified in a similar manner. Assuming the island controller is being asked to yield, this can be expressed using timing diagram

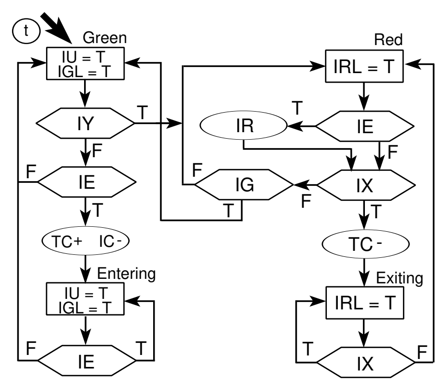


We now provide a proof, using the inference rules presented in section 3.6, that this timing diagram is a logical consequence of the three controller diagrams and the environmental assumptions. Formally, we represent the environmental assumptions as shown in figure 5. The proof is given in natural deduction style. Ideally we would develop this proof in an animated system (akin to *Hyperproof*) that updated each representation with the information from the current context as the proof progressed; lacking animation in this presentation, we provide the diagrams corresponding to each step explicitly. To keep the proof compact, rather that insert the state machine diagrams into the lines of the proof, we insert a grey icon indicating that the formula at a given step is a diagram; the diagrams corresponding to each step are numbered and are provided following the proof. We use a black rectangular icon to stand for the goal diagram on the appropriate lines.

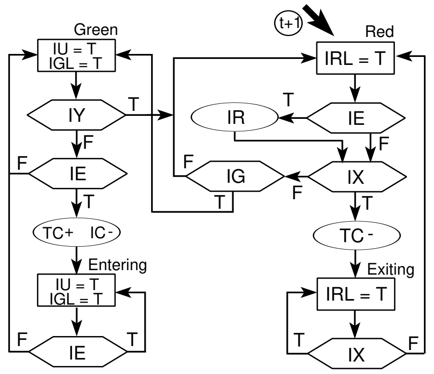
1.	⬡ Environmental Assumptions	Given (figure 5)
2.	⬡ Tunnel and Island ASMs	Given (figure 4)
3.	$IY(t)$	Given
4.	⬡ (1)	Asserting State; 3
5.	$\neg IU(t)$	Assume
6.	■	Modus Ponens, $\exists$ Intro; 3, 5
7.	$IU(t)$	Assume
8.	⬡ (2)	Assume
9.	⬡ (3)	State Transition; 3, 8
10.	$\neg IU(t+1)$	Value in State; 9
11.	■	Modus Ponens, $\exists$ Intro; 3, 10
12.	⬡ (4)	Assume
13.	$\exists t_1\; t_1 > t \wedge \neg IE(t_1) \wedge$ $\forall t_2\; t_1 > t_2 \geq t \rightarrow IE(t_2)$	Instantiate Given
14.	$\forall t_2\; t_1 \geq t_2 \geq t \rightarrow IU(t_2)$	State Trans/Val in State; 12
15.	$\forall t'\; t \leq t' \leq t_1$ ⬡ (5)	State Transition; 4, 14
16.	⬡ (6)	Looping; 12, 13
17.	$IU(t_1+1)$	Value in State; 16
18.	⬡ (7)	State Transition; 14, 15
19.	$IY(t_1+1)$	Value in State; 18
20.	⬡ (8)	State Transition; 16,19
21.	$\neg IU(t_1+2)$	Value in State; 20
22.	■	Mod Ponens, $\exists$ Intro; 3, 17, 21
23.	■	Assert States Exhaust; 8, 12
24.	■	Excluded Middle; 5, 7



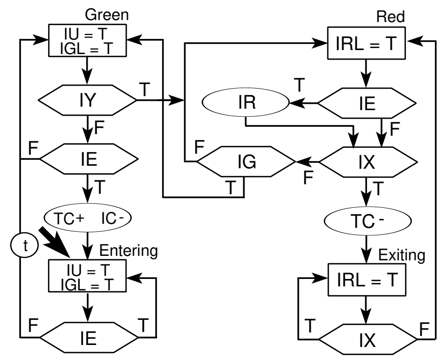
(1)



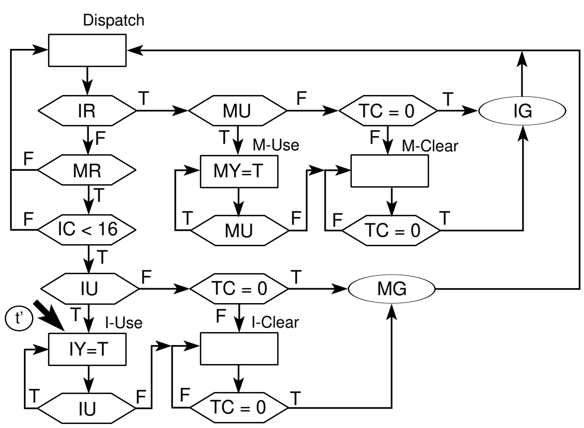
(2)



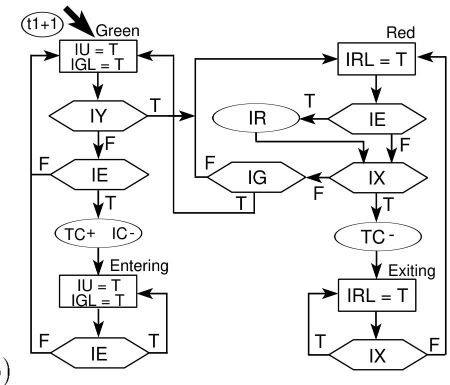
(3)



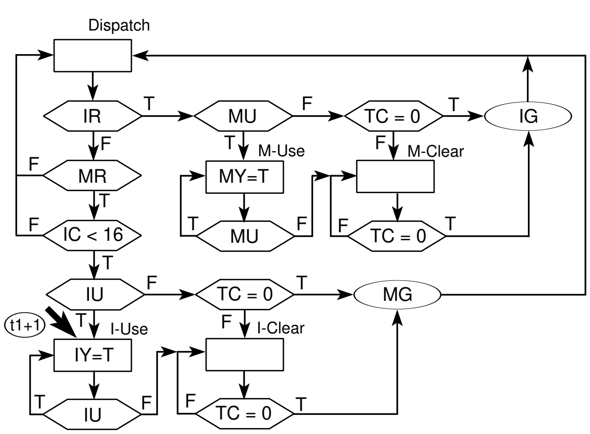
(4)



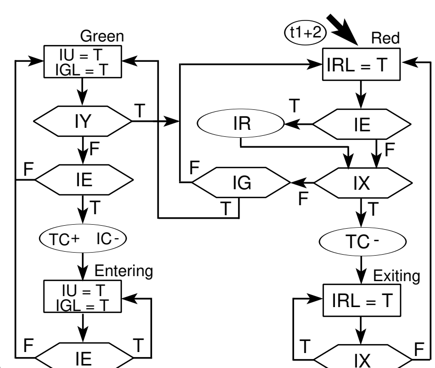
(5)



(6)



(7)



(8)

Assuming we are satisfied with the state-level design, the next step becomes designing an implementation of the system. This too can be done using the inference rules of the logic. There are several algorithms for converting a state machine into physical hardware [14]; for this example we will take a state-encoded approach to the design. The circuit diagrams provided in figure 6 for

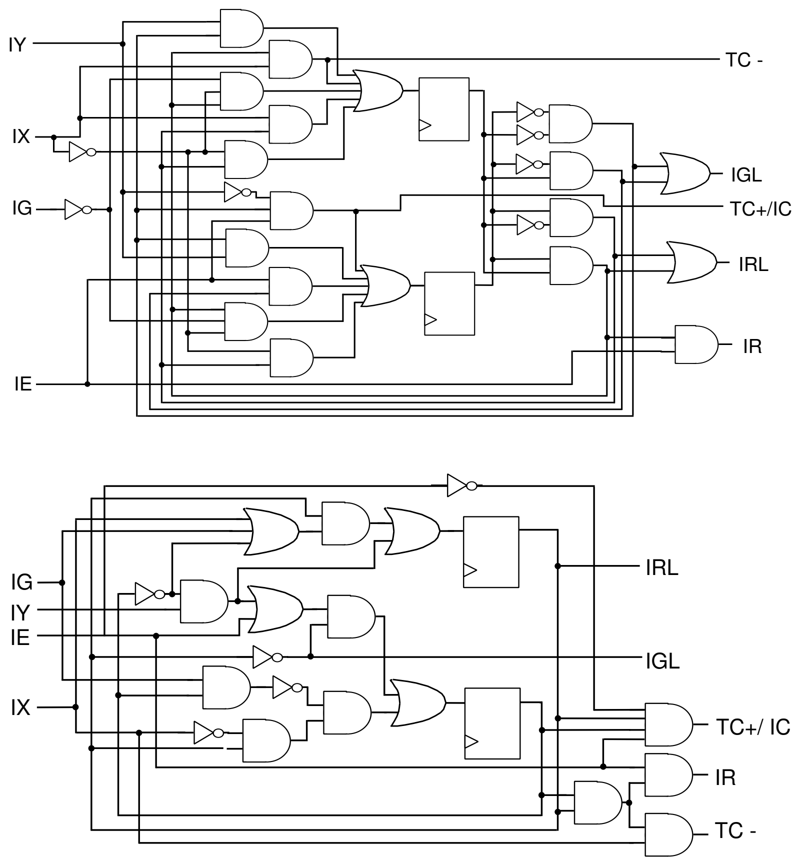


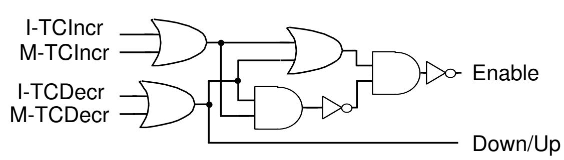
Figure 6: Circuit diagrams for the island light controller. The top diagram is naïvely produced using a state-encoding of green=00, entering=01, exiting=10, and red=11. The simplified yet behaviorally equivalent bottom diagram is derived from the first using the boolean algebra based circuit diagram inference rules .

the island light controller can be shown to be logical consequences of the associated state machine in figure 4. Although the logic includes a rule for inferring implementations from ASM charts, that rule is not presented here.

The diagrams in figure 6 suggest how the logic can be used for design. Assuming we derive the top diagram under an inference rule for state-encoded implementations of state machines, there are a number of optimizations we could make to minimize the number of gates in the circuit. The bottom diagram reflects one possible minimized circuit obtained from the original by means of the boolean algebra based circuit diagram inference rules. The soundness of these rules, established but not proven here, is sufficient to assure us that the two circuits are behaviorally equivalent. We are in the process of proving a completeness result based on a canonical form for circuit diagrams [7]; this would enable us to transform any two behaviorally equivalent circuit diagrams into one another using the inference rules.

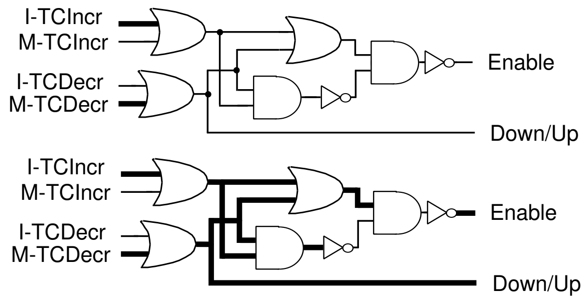
Given implementations of the three controllers, all that remains is to design the components necessary to interface the three implementations. While some of the interface consists only of wires, additional logic is required to integrate the counters and the needed comparator into the final design. This brings us back to the issue of verification, as we would like to formally establish the correctness of the interface logic.

As an example, consider the logic required to interface to the counter TC that records how many cars are currently in the tunnel. Assume we have chosen to use a LS191 up-down counter [13] in our implementation. This counter has an enable signal and a single signal for indicating whether the counter should count up or count down. When the enable signal is low, a low voltage on the up/down line causes counting upwards and a high voltage on the up/down line causes counting downwards; no counting occurs when the enable line voltage is high. Once we interface the controller implementations with the counter, we must verify that our interface logic routes signals properly to the LS191. Assume that we used the following interface logic, where I-TCIncr is the signal TC+ from the island light controller and the remaining signals are analogously defined.



We can use the simulation style inference rules on circuit diagrams to verify that our interface logic behaves as desired. As an example proof, consider the case when the island controller issues a tunnel counter increment and the mainland controller issues a tunnel counter decrement.[5] In this case the counter should hold its current value. The following proof consists of two diagrams. In the first, we assume that both I-TCIncr and M-TCDecr are asserted simultaneously. The second diagram shows the resulting asserted signals once the simulation rules are applied to the first diagram.

[5]This combination is possible in our proposed state machines.



5 Conclusions

Diagrams are a powerful — and underutilized — notation. They are good representations for hardware reasoning because they are specialized to particular properties of systems. Combining specialized representations within a heterogeneous logic provides a powerful paradigm for supporting reasoning in addition to proof. A heterogeneous hardware logic that includes diagrams therefore suggests a possible solution to the usability problem in formal methods.

Our heterogeneous logic approach has been criticized as being unnecessarily complex [15]; traditional, sentential logics are argued to be simpler and more flexible because they can model many properties using a single notation [9]. Such generality comes at a cost with respect to natural reasoning. We feel that complexity in the underlying system is a suitable tradeoff for greater usability.

Other researchers have explored formal usage of diagrams in limited situations in hardware reasoning. Timing diagrams have received the most attention, being cited as a more natural formalism for use in place of temporal logic [17] [10]. Other systems have employed more general usage of diagrammatic representations [5] [18] [6]. All of these systems formalize diagrams by translating them into known sentential logics; proofs in these systems are carried out in the sentential logic, with the diagrams serving as interface tools.

The translation approach is reasonable when using a single diagrammatic notation. In order to use this approach to define inference rules in our heterogeneous framework, we would have to either translate all of the representations into a common sentential logic or establish formal connections between multiple existing sentential logics. Rather than risk adding logical errors via the translation process, we define our rules directly on the diagrams.

There are additional advantages to our approach. Diagrams often encode substantial amounts of detailed information that may or may not be relevant to a verification effort. Translation, unable to filter what information to capture and what to ignore, might produce a considerably larger specification than is actually necessary. Our approach is also highly modular; adding a new representation does not involve integration of additional underlying logics.

Additional research is required before we can construct a tool based upon our logic. We need to extend it with support for making abstractions [11] and with better support for automatic verification methods. Automatic methods are required for handling large examples. One might question why, if we intend to automate the logic, we are concerned with how well the logic supports human reasoning. Automatic methods cannot fully handle many industrial size examples due to complexity bounds. Human reasoning is often needed in order to decompose a problem into pieces that are sufficiently small for automatic methods. In addition, the decidable logics underlying automatic methods are necessarily limited in expressibility; human reasoning is required for problems falling outside of these limitations. Addressing these two issues would yield a prototype system geared towards proof and reasoning that will help bring formal methods closer to designers.

Acknowledgements

The author thanks Jon Barwise, Shriram Krishnamurthi, Steve Johnson, Gerry Allwein, and several anonymous reviewers for useful discussions and comments on this project.

References

[1] Jon Barwise and John Etchemendy. Information, Infons, and Inference. In Robin Cooper, Kuniaki Mukai, and John Perry, editors. *Situation Theory and Its Applications*. Stanford University Press, 1990.

[2] Jon Barwise and John Etchemendy. Hyperproof. CSLI Lecture Notes, University of Chicago Press, 1994.

[3] Jon Barwise and John Etchemendy. Logic, Proof, and Reasoning. In Alan Makinowski, editor, *Companion to Logic*. Blackwell, to appear.

[4] Jon Barwise and John Etchemendy. Visual Information and Valid Reasoning. This volume.

[5] L.K. Dillon, G. Kutty, L.E. Moser, P.M. Melliar-Smith, and Y.S. Ramakrishna. A Graphical Interval Logic for Specifying Concurrent Systems. Technical Report, UCSB, 1993.

[6] Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal System Design — Interactive Synthesis on Computer-Assisted Formal Reasoning. In Luc Claesen, editor, *Formal VLSI Specification and Synthesis: VLSI Design-Methods-I*. North-Holland, 1990.

[7] Kathi Fisler. A Canonical Form for Circuit Diagrams. Indiana University Department of Computer Science Technical Report TR432, May 1995.

[8] Kathi Fisler. A Logical Formalization of Hardware Design Diagrams. Indiana University Department of Computer Science Technical Report TR416, September 1994.

[9] M.J.C. Gordon. Why higher order logic is a good formalism for specifying and verifying hardware. In G.J. Milne and P.A. Subrahmanyam, editors. *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Conference on VLSI*. North Holland, 1986.

[10] K. Khordoc, M. Dufresne, E. Cerny, P. Babkine, and A. Silburt. Integrating Behavior and Timing in Executable Specifications. In *Proceedings, Computer Hardware Description Languages and their Applications*, pp. 385–402, April 1993.

[11] Thomas Frederick Melham. Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic. University of Cambridge Computer Laboratory, Technical Report 201, August 1990.

[12] NASA Langley Formal Methods Workshop. Panel Sessions and Discussions. May, 1995.

[13] National Semiconductor Corporation. *LS/S/TTL Logic Databook*, 1987.

[14] Franklin P. Prosser and David E. Winkel. *The Art of Digital Design*, 2nd edition. Prentice-Hall, 1987.

[15] Reviewers' Comments on papers submitted to *Theorem Provers and Circuit Design 1994* and *Computer Hardware Description Languages and Their Applications*, 1995.

[16] John Rushby. Formal Methods and Digital Systems Validation for Airborne Systems. NASA Langley Contractor Report 4551, December 1993.

[17] Rainer Schlör and Werner Damm. Specification and Verification of System-Level Hardware Designs using Timing Diagrams. In *Proceedings of the European Conference on Design and Automation*. February 1993.

[18] Mandayam Srivas and Mark Bickford. SPECTOOL: A Computer-Aided Verification Tool for Hardware Designs, Vol I. Rome Laboratory Technical Report RL-TR-91-339, Griffiss Air Force Base, NY, December 1991.