A Logical Formalization of Hardware Design Diagrams*

Kathi Fisler
Department of Computer Science
Lindley Hall 215
Indiana University
Bloomington, IN 47401
kfisler@cs.indiana.edu

Abstract

Diagrams have been left as an informal tool in hardware reasoning, thus rendering them unacceptable representations within formal reasoning systems. We demonstrate some advantages of formally supporting diagrams in hardware verification systems via a simple example and provide a logical formalization of hardware diagrams upon which we are constructing a verification tool.

1 Introduction

The increased use of formal methods for verifying hardware specifications has generated a wealth of research into the formal models and representations of hardware that best facilitate the verification task. Most such models are based on combinations of temporal and higher-order logic which, while effective, do not necessarily reflect the models used during the design process. The hardware design process involves the use of a collection of diagrammatic forms, such as circuit diagrams and timing diagrams, which depict certain characteristics of hardware components more naturally than purely sentential representations. Given the relationship between good representations and ease and clarity of proof, it then seems natural to ask why formal verification does not support these representations.

One answer is that the lack of formalization of diagrammatic representations has precluded their use in formal proof and verification. Another is that diagrammatic representations are often seen as too specialized to support general reasoning about hardware components. However, the argument could be made that certain hardware characteristics are better represented diagrammatically, therefore reflecting limitations to sentential representations. Given this, it would seem that the development of logical systems in which sentential and diagrammatic representations could interact formally would be the best choice for hardware formal methods. Barwise and Etchemendy have demonstrated the feasibility of such *heterogeneous* logics via their *Hyperproof* system [1] [2]. Our goal in this project is to apply the logical underpinnings of *Hyperproof* to the domain of hardware design and verification.

The first step in developing a heterogeneous logic for the hardware domain is the formalization of the diagrammatic notations; this portion of the project is presented here. In this paper we are going

*To appear in *Diagrammatic Reasoning*, edited by Gerard Allwein and Jon Barwise.

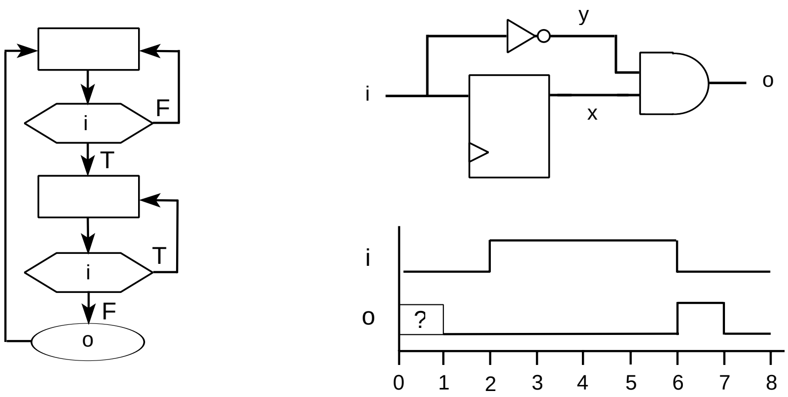


Figure 1: Three diagrams from the logic corresponding to an implementation of a single-pulser component.

to be studying several logical relationships between four kinds of objects: physical hardware devices, circuit diagrams, algorithmic state machine (ASM) charts, and timing diagrams. These three diagrammatic representations have been chosen because they address three of the four principle aspects of hardware design: control, architecture, and timing [11].[1] Examples of these diagrammatic forms are given in Fig. 1. We are interested in exploring the following key relationships between these diagrammatic representations and physical devices:

- What does it mean for some device to be described by, or to be an implementation of, some diagram?
- What does it mean for two diagrams to describe the same device or devices?
- What does it mean for two devices to be observationally (*i.e.*, behaviorally) equivalent?
- What does it mean for two diagrams to describe observationally equivalent devices?
- What does it mean for one diagram to be a structural consequence, or a behavioral consequence, of other diagrams?
- What are the valid methods of inference between diagrams?

In order to give a rigorous mathematical account that begins to answer these questions, we need to give mathematical models for each of these four kinds of objects. This requires that we isolate those features of physical devices and diagrams that are crucial to answering these questions. On the other hand, we want to abstract away from other features which may be important for other considerations (layout, or readability, for example) but are not relevant to these particular questions. We do this by giving a model of physical hardware devices in Sect. 3.1. In Sects. 3.2, 3.3, and 3.4 we present the models of the diagrammatic representations and give each representation a semantics in terms of the model of physical hardware. Examples of inference rules can be found in Sect. 4.

[1] The fourth aspect, data hierarchy, is less developed in hardware practice.

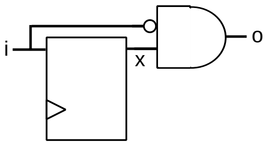
2 Verification and Diagrammatic Representation

Before presenting our formalization, we provide a brief demonstration in support of formally using diagrams in hardware verification. We believe that diagrammatic reasoning offers two main advantages over purely sentential hardware verification: clarity of representation and conciseness of proof. We will use a simple component called a *single-pulser* to demonstrate our arguments. A single-pulser contains one input and one output; for each pulse on the input, a unit duration pulse is emitted on the output. The diagrams corresponding to a single-pulser appear in Fig. 1; the timing diagram in Fig. 1 provides a visual description of the component's behavior.

Various sentential verifications of the single-pulser have been studied by [9]; for sake of comparison, we will contrast their sentential theorem-prover verification with our intended diagrammatic verification. We have chosen to use their theorem-prover verification because one of our goals is to develop a proof-checker with support for diagrammatic representations. PVS is the theorem prover used in [9], which we take as representative of sentential theorem provers.[2]

Verifying hardware using a theorem prover involves writing a statement for the component's specification, writing a statement for an implementation of the component, and then proving that the implementation logically implies the specification. Depending upon the theorem prover in question, the statements are given in some variant of higher-order logic.

To address the issue of clarity, consider the PVS implementation and specification of the single-pulser proposed by [9]. Their implementation is given below and is based upon the accompanying circuit diagram.[3]



$$\begin{aligned}
&\mathit{imp}(i,O):\mathit{bool} = (\exists x:(\mathit{delay}(i,x)\wedge \mathit{and}^{\circ}_{\bullet}(i,x,O)))\\
&\mathit{delay}(i,O):\mathit{bool} = (\forall t:(O(t+1)=i(t)))\\
&\mathit{and}^{\circ}_{\bullet}(a,b,c):\mathit{bool} = (\forall t:(y(t)=(-a(t))\times b(t)))
\end{aligned}$$

Notice that the PVS representation of the implementation is nothing but a translation of the diagram into the syntax of PVS — neither representation contains any more information than the other.[4] The behavioral specification is given in two parts, as follows. The first part, *spec1*, expresses that "whenever there is a pulse on the input signal i, say from time n to time m, there is a unique time k in the vicinity of the input pulse so that the output signal is asserted [9]."

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

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

[2] PVS has been developed by SRI International.

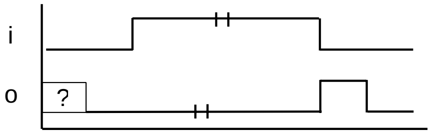
[3] In the circuit diagram, the rectangular object depicts a device that delays its input for one unit of time before passing it to the output, the small circle represents logical negation, and the gate on the right represents boolean *and*.

[4] We could consider the lengths of wires in the circuit diagram as information not available in the sentential representation, but that information is more fine-grained than our logic is tuned to handle at present.

As pointed out by [9], *spec1* is not sufficient because it does not expresses the desired behavior of the single-pulser between input pulses. Specification *spec2* is intended to correct this problem; it states that any output pulse must occur in the vicinity of an input pulse.

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

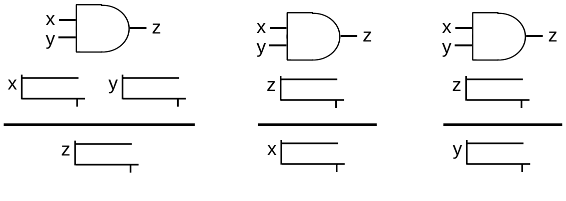
spec1 can be captured using the following timing diagram.[5] One advantage to use the timing diagram is that there is no need for the *Pulse* definition because it is inherent in the diagram. This suggests that diagrams may be more compact than sentential representations in some circumstances.



We claim that the timing diagram is a clearer representation of the intended behavior of a single-pulser than the two sentential specifications. The meaning of neither sentential specification is immediately clear, despite the fact that they are written in a straightforward style of higher-order logic. In fact, the average person might construct a diagrammatic depiction of the specifications in the process of understanding their full meanings. The advantages of clear specification are well known in the verification community. Careless interpretation of either specifications or implementations can lead to lost time in establishing proofs, or worse still, invalid proofs of correctness. Of course, the argument can also be made that there are issues of interpretation involved in using diagrams as well; we agree, but claim that the clarity of properly formalized diagrammatic representations minimizes this problem.

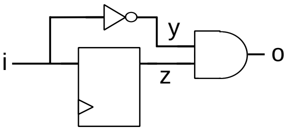
We now turn to comparing the sentential single-pulser verification to our proposed diagrammatic verification. There are two aspects to consider: the time to develop proofs and the conciseness of the resulting proof. The PVS proof referenced in [9] took an estimated half-hour of proof time for a relatively novice PVS user. The main time expenditure was in properly formulating the specification, which took considerably longer than the actual verification [12]. Although we have no evidence to support this, we believe that specifications may be easier to state and debug using diagrammatic representations that are more familiar to practicing designers.

Understanding our diagrammatic proof requires understanding our rules of inference. Our intent is to design the logic such that diagrammatic rules of inference mimic the informal reasoning steps used by designers in practice. Example inference rules relating *and* gates and timing diagrams appear below; other inference rules on *and* gates, such as one where a low input yields a low output, can be derived from these three primitive rules.

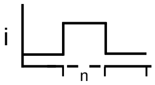


Now consider the following diagrammatic proof that corresponds to the proof of *spec1*. In the timing diagrams, the dashed axis notation denotes that the dashed tick repeats the number of times indicated on the dashed line.

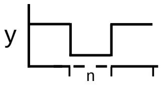
[5] The double line markings on the timing diagram indicate that the given signal levels hold for an undetermined period of time; the transitions on the signals must still occur simultaneously.

1. 

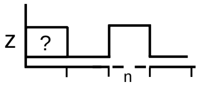
Assume

2. 

Assume

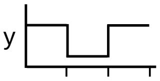
3. 

Inverter Rule, 2

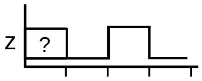
4. 

Unit Delay Rule, 2

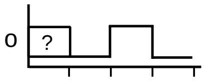
5. a. $n = 1$ Assume

b. 

Instantiate value of n, 3

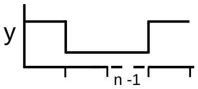
c. 

Instantiate value of n, 4

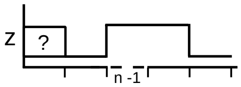
d. 

And Rule, 5b, 5c

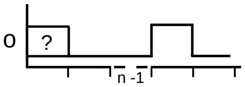
6. a. $n > 1$ Assume

b. 

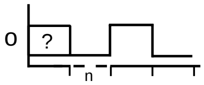
Repetition expansion, 3

c. 

Repetition expansion, 4

d. 

And Rule, 6b, 6c

7. 

Merge, 5d, 6d

Comparing this proof to the PVS proof trace from [9] (reproduced in appendix A), it seems reasonable to argue that the diagrammatic proof is easier to follow and quite possibly easier to produce than the one required to verify *spec1* in PVS. The steps taken in the diagrammatic proof are also at a lower granularity (for sake of example) than those we expect to be taken in practice, thus compacting the proof even further. Though we have no measured results to support this, empirical evidence using *Hyperproof* indicates that proofs are often substantially shorter than their purely sentential equivalents [5].

The above presentation argues the benefits of using diagrammatic representations in formal verification, but it does not adequately address our particular approach of developing a logic of hardware diagrams. Certainly it would seem reasonable to merely provide a diagrammatic interface to an existing sentential logic, thereby allowing us to rely on existing tools for verification. There are certain obvious immediate benefits to such an approach, such as the timeliness with which diagrams could be used to aid in formal verification.

We believe, however, that this approach is not the correct one to take for three reasons. By using diagrams merely as an interface tool, we leave them as second-class citizens to sentential logic in the realm of formal reasoning. We believe that diagrams are as valid a representation as sentential forms in reasoning and are interested in the creation of logics that put diagrams on equal par as a valid representation. In addition, using diagrams merely in an interface capacity sidesteps our belief that there are logical relationships between different diagrammatic hardware representations. Identifying these relationships may lead us to even stronger frameworks for verification, but to do so requires closer examination of the diagrams themselves as first-class citizens.

The third reason is more practical. Diagrams, by their very nature, often encode substantial amounts of detailed information that may or may not be relevant to a verification effort. A translation effort would need to be able to decide what information to capture in the translation and what to ignore. As most systems would tend to be conservative, the translation might result in a considerably larger specification than what is actually needed for the verification effort. Leaving the logic at the level of the diagrams prevents this; the user accesses only what information is relevant for the verification task at hand. This would seem to be an issue mostly in timing diagrams, where the relationships between pieces of the diagrams are more subtle and open to interpretation than in either circuit diagrams or algorithmic state machine charts. Errors in interpretation often result in incorrect specifications; we believe such errors would be easier to detect while working directly with the diagrams, rather than by working through a translated representation.

3 The Heterogeneous Logic

Having argued in favor of formal diagrammatic reasoning in hardware verification, we now proceed to present our formalization of hardware diagrams.

3.1 Physical Devices

Given that the diagrammatic forms supported in this logic all describe physical hardware, our model of physical components must reflect the relationships individually reflected in each of the diagrammatic representations. This model will need to be able to account for structural properties (depicted by circuit diagrams), state properties (depicted by ASM charts), timing properties (depicted by timing diagrams), and input-output behavior (depicted in all three forms of diagrams in different ways and to different degrees). Due to this, the definitions in this section are more extensive than those needed later to model any one of the diagrammatic representations.

The model is developed as follows: we will define an abstract device to be a certain kind of mathematical structure. This definition will focus on the structure of the device (how it is built up from components such as gates and wires) and the interface between the device and its environment.

Given this, we will define what we mean by a concrete device, a notion intended to capture the idea of a piece of hardware in a particular configuration at a particular time. This will allow us to define notions of state for concrete devices and input-output characteristics for abstract devices. Once we have all these notions in place, it will be a relatively straightforward matter to define what it means for a device to be described by, or to implement, a given diagram.

The definitions in this section take as primitive the notions of *port* and *wire*. A port is a physical point of connection between a gate or delay element and its surrounding circuitry. We will assume that our entire device is an encapsulated component so that we may speak of interface ports that connect the device in question to some external environment. A wire is used to connect one port to another, with the result that any two ports attached by a wire carry the same value at all times. In addition, we assume the existence of a universe U of values that are valid voltages on wires in a device; for now an exact U is unnecessary, but this universe will eventually be taken to be the set $\{0, 1\}$.

Definition 1

1. An *assignment* is a total function from a set of ports to the universe U. More specifically, an assignment on a set of ports P is termed a P-assignment. A particular P-assignment can be viewed as representing one possible combination of values on the ports in P.

2. A *basic component* is a tuple $c = \langle I, O, F, D\rangle$ where I and O are disjoint sets of ports called the input ports and output ports of c, respectively; F is a function which assigns each $p \in O$ a function F_p from I-assignments into U; and D is a function from O to the non-negative integers called the output delay function.

3. A *gate* is a basic component in which D is the constantly zero function for all ports $p \in O$.

4. A *unit delay element* is a basic component in which I and O each contain a single port, $F(p)$ is the identity function for $p \in O$, and $D(p) = 1$ for $p \in O$.

5. A *wiring* is a function c from wires to sets of ports of cardinality at least two. Ports which are elements of $c(w)$ for a single wire w are said to be *wired together* or *connected* by w. By the *co-domain* of a wiring c we mean $\bigcup_{w \in dom(c)} c(w)$.

Definition 2 An *abstract device* is a 5-tuple $D = \langle I, O, B, W, c\rangle$ satisfying the following conditions:

1. I and O are disjoint sets of ports providing the interface to D; they are called the interface input ports and the interface output ports, respectively.

2. B is a set of basic components. The sets of ports for the elements in B must all be pairwise disjoint and disjoint from I and O.

3. W is a set of wires and c is a wiring with domain W and co-domain a subset of the ports of D; this includes the interface ports of D as well as the ports of all gates and delay elements of D.

4. The sets $c(w)$ partition the ports of D. Each cell of this partition contains exactly one internal output port or interface input port. Each cell of the partition contains at least one internal input port or output interface port of D.

Definition 3 An abstract device is called *combinational* if it does not contain any delay elements. Abstract devices which are not combinational are called *sequential.*

Definition 4 Given an abstract device D, the elements of $I \cup O$ are called the *interface ports* of D. All other ports in D are called *internal ports. Internal input ports* are those internal ports serving as an input port to some basic component b in D; *internal output ports* are those internal ports serving as an output port to some basic component b in D.

Physical devices have a graph-like structure which we want to use for structural comparisons between devices. We are most interested in the paths between gates within devices; the following set of definitions formalizes paths within physical devices.

Definition 5 Given an abstract device D, there is a *connecting step* from port p_i to port p_j, denoted $p_i \leadsto p_j$, under the following conditions:

1. If p_i is an internal input port of basic component b, then $p_i \leadsto p_j$ iff p_j is an internal output port of b.

2. If p_i is an internal output port or an input interface port, then $p_i \leadsto p_j$ iff p_j is an internal input port or an output interface port and p_i and p_j are wired together by some wire w.

Notice that according to this definition, there can be no connecting steps from output interface ports to any other ports.

Definition 6 If $p_i \leadsto p_j$, the basic component or wire that connects p_i and p_j is called the *connecting element* of p_i and p_j.

Definition 7 Given an abstract device D:

1. There is a *connecting path* in D from port p_0 to port p_n (denoted $p_0 \leadsto^* p_n$) if there exists a finite transitive chain of connecting steps

$$p_0 \leadsto p_1 \leadsto p_2 \leadsto \ldots \leadsto p_n$$

2. There is a *connecting cycle* in D if there is a connecting path $p \leadsto^* p$ for some port p in D such that the path contains at least two distinct ports. In this case, D is said to contain *feedback.*

Although any abstract device corresponds to a piece of physical hardware, there are additional restrictions we would like to place on the devices we are willing to consider in this investigation. In particular, we would like all basic components within an abstract device to contribute something to the functionality of the device; we would also like to guarantee that all feedback loops pass through a delay element. These requirements are captured in the following definition.

Definition 8 An abstract device D is *well-connected* iff:

1. Treating the basic components as nodes and the wiring function as giving rise to edges yields a connected and directed graph.

2. For each internal port p in D there exists a connecting path from p to an element of O.
3. Every connecting cycle in D contains some step $p_i \rightsquigarrow p_j$ such that the connecting element of p_i and p_j is a delay element.

Lemma 1 *If a device is well-connected, then it cannot have two output ports p_i and p_j wired together.*

Definition 9

1. Given basic components $b_1{=}\langle I_1, O_1, F_1, D_1\rangle$ and $b_2 = \langle I_2, O_2, F_2, D_2\rangle$, a bijection $\phi : I_1 \cup O_1 \to I_2 \cup O_2$ is an *isomorphism between b_1 and b_2* iff
   (a) $\phi(I_1) = I_2$.
   (b) $\phi(O_1) = O_2$.
   (c) For each $p \in O_1$, $F_1(p)$ is equivalent to $F_2(\phi(p))$.
   (d) $D_1 = D_2$.
2. Given abstract devices $D_1 = \langle I_1, O_1, B_1, W_1, c_1\rangle$ and $D_2 = \langle I_2, O_2, B_2, W_2, c_2\rangle$, a bijection ϕ from the ports of D_1 to the ports of D_2 is an *isomorphism between D_1 and D_2* iff
   (a) $\phi(I_1) = I_2$.
   (b) $\phi(O_1) = O_2$.
   (c) For each $b_1 \in B_1$, there exists a $b_2 \in B_2$ such that the restriction of ϕ to the ports of b_1 is an isomorphism between b_1 and b_2.
   (d) For all ports p_a and p_b in D_1, p_a and p_b are wired together iff $\phi(p_a)$ and $\phi(p_b)$ are wired together.
3. Abstract devices D_1 and D_2 are *isomorphic* if there exists an isomorphism between them.

While the definitions above are sufficient to indicate the physical structure of a device, some additional terminology is needed to talk about the behavior of a device. One aspect of behavior is the functional definition implicit in the basic components used and how they are connected. A related aspect is the observable behavior of a device — that which is observed when values in U are supplied on the input interface ports and the device computes values for the output interface ports. To talk about these notions, we introduce the concept of a concrete device, capturing the idea of using abstract devices for computation.

Definition 10 A *concrete device* is an ordered pair $\langle D, i\rangle$ where D is an abstract device and i is an assignment to the ports of D such that 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.

Definition 11 A concrete device $\langle D, i\rangle$ is said to be *well-connected* iff D is well-connected.

Definition 12 Concrete devices $\langle D_1, i_1\rangle$ and $\langle D_2, i_2\rangle$ are *isomorphic* iff D_1 and D_2 are isomorphic and for all ports p in D_1, $i_1(p) = i_2(\phi(p))$, where ϕ is the isomorphism between D_1 and D_2.

Lemma 2 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 assignment i' satisfying the following conditions:

1. $\langle D, i' \rangle$ is a concrete device.

2. If p is an interface input port of D, then $i'(p) = a(p)$.

3. If p is an output port to a delay element with input port p_{in}, then $i'(p) = i(p_{\text{in}})$.

4. 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.

5. If p is any other port, let w be the wire connected to p and let q be the unique internal output port or input interface port in $c(w)$. Then $i'(p) = i'(q)$.

Proof First we establish that there exists some assignment i' that satisfies the listed conditions. We can construct a function i' from ports to values using conditions 2 through 5. Inspection tells us that every port in D will be assigned a unique value in i' by one of these conditions. Therefore, i' meets the definition of an assignment.

We must now establish that the assignment i' described above is unique. Assume that there are assignments i_1 and i_2 that both satisfy the listed conditions, but such that $i_1 \neq i_2$. Let p be a port in D such that $i_1(p) \neq i_2(p)$ and there is no port p' in D such that $i_1(p') \neq i_2(p')$ and p' lies on a path from an input interface port to p. Clearly p is not an interface input port since by definition, $i_1(p) = a(p) = i_2(p)$. If p is the output port of a gate or delay element, then since gates and delay elements are associated with deterministic functions it must follow that $i_1(q) \neq i_2(q)$ for some port q that serves as an input port for that same gate or delay element. If there is a path from an input interface port to p, this would contradict the choice of p since q would then have to lie on a path from an input interface port to p. If no such path exists, then the value of p must be determined by a feedback loop. Both i_1 and i_2 were constructed from i, and the feedback loop must contain a delay element, so the deterministic nature of gates tells us that any value relying strictly on that of a feedback loop must be uniquely determined, so $i_1(p) = i_2(p)$.

The only remaining option is that p is some other port and $i_1(q) \neq i_2(q)$ where q is the unique output port or input interface port connected to p. This also contradicts our choice of p since q would then have to lie on a path from an input interface port to p. It therefore follows that $i_1 = i_2$, so the constructed assignment must be unique.

The only condition left unproven is the first one. However, by definition a concrete device is simply a pair of a device and an assignment for that device. We have established that i' is an assignment for D, so $\langle D, i' \rangle$ meets the definition of a concrete device. □

Definition 13 Let $C = \langle D, i \rangle$ be a concrete device and let a be an assignment to the interface input ports of D. The unique assignment i' given by the above lemma is said to be the *derived assignment* for C, given a, and is written $C[a]$.

Definition 14 Let $C = \langle D, i \rangle$ be a concrete device and let a be an assignment to the interface input ports of D. Concrete device $\langle D, C[a] \rangle$ is said to *follow from* C, given a.

The above definitions are sufficient for observing the behavior of a device as one set of input values is applied to the input ports. Taking this idea one step further, it will often be desirable to observe how a device behaves over an entire sequence of input assignments, as formalized below.

Definition 15 An *assignment sequence* for an abstract device D is a sequence of assignments $i_1, i_2, \ldots$ to the interface input ports I of D.

Definition 16 Let $C = \langle D, i \rangle$ be a concrete device. A *run* of C is a sequence $r = \langle C_0, C_1, \ldots \rangle$ of concrete devices such that there is an assignment sequence $\langle i_1, i_2, \ldots \rangle$ of length one shorter than that of r, $C_0 = C$, and for each $j \geq 1$, C_j is the concrete device that follows from C_{j-1} given i_j.

Definition 17 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 $r = \langle C_0, C_1, \ldots \rangle$ of a concrete device is a sequence $\langle O_0, O_1, \ldots \rangle$ of length equal to that of r, where each O_i is the output of C_i.

Definition 18 Given a concrete device $C = \langle D, i \rangle$, the *state* of C is the restriction of i to the output ports of the delay elements in D. The set of *possible states* of C is the set of all assignments on the output ports of the delay elements in D.

The definitions up until this point have been concerned with structural, as opposed to behavioral, relationships between devices. However, it is often useful in reasoning about hardware to talk about behavioral relationships as well; such relationships are captured in the following definitions.

Definition 19

1. Concrete devices $C_1 = \langle D_1, i_1 \rangle$ and $C_2 = \langle D_2, i_2 \rangle$ are *behaviorally indistinguishable* 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.
2. Abstract device D_1 can *simulate* concrete device $\langle D_2, i_2 \rangle$ if there exists an assignment sequence i_1 such that $\langle D_1, i_1 \rangle$ is behaviorally indistinguishable from $\langle D_2, i_2 \rangle$.
3. We say that abstract device D_1 *unconditionally simulates* D_2 if for all assignment sequences i_2 for D_2, D_1 can simulate $\langle D_2, i_2 \rangle$. If there exist i_2 and i_2' such that D_1 simulates $\langle D_2, i_2 \rangle$ but D_1 does not simulate $\langle D_2, i_2' \rangle$, then we say that D_1 *contextually simulates* D_2.
4. Abstract devices D_1 and D_2 are *behaviorally equivalent* iff D_1 is isomorphic to some D_1' which unconditionally simulates D_2 and D_2 is isomorphic to some D_2' which unconditionally simulates D_1.

Lemma 3 *If abstract devices D_1 and D_2 are isomorphic, then they are behaviorally equivalent.*

3.2 Circuit Diagrams

While there are many pieces of information one could include in a model of circuit diagrams, our definition focuses on only three things: the ends of the lines on the left side of such a diagram, which conventionally represent input wires, the ends of the lines on the right side of diagram, traditionally representing output wires, and the points of connection between lines and the icons representing gates and delays. The valid gate icons are shown in Fig. 2.

Convention dictates that the term "circuit diagram" be used for those diagrams which bear some correlation to functioning physical devices, regardless of whether that relationship is partial or total. This does not rule out the existence of diagrams that have no physical analog, due to

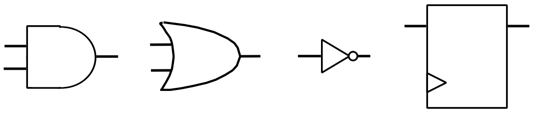


Figure 2: Icons to be used for binary and, binary or, negation, and unit delay, respectively from left to right.

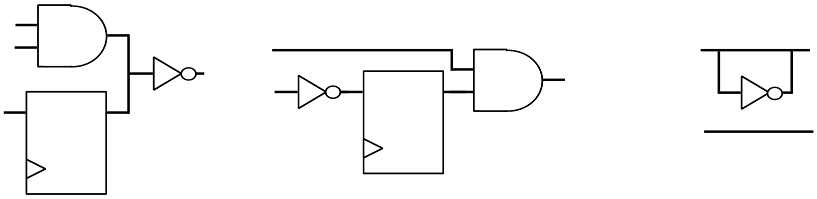


Figure 3: Examples of circuit sketches, not all of which represent functional physical devices.

any ambiguous or inconsistent physical relationships they might represent. In order to keep these two notions separate, we will use the term "circuit sketch" to refer to any drawing of gates and wires, reserving the term "circuit diagram" for those sketches that represent implementable physical devices. Both of these terms are given formal definitions in the remainder of this section.

The definitions below rely on a notion of how gate icons and wire lines are modeled. We model a wire line as an ordered pair of the points it connects. We model a binary gate icon 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.

Definition 20 A *circuit sketch* s is a tuple $\langle P, I, O, W, N, D, A, R \rangle$ satisfying the following conditions:

1. 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, respectively.

2. W, N, and D are disjoint subsets of $P \times P$. The pairs in these three sets are called the *wire lines*, *negation icons*, and *delay icons*, respectively. The first and second elements of these are called the left and right elements, respectively.

3. A and R are disjoint subsets of $P \times P \times P$. The elements of these sets are called the *and gate icons* and *or gate icons*, respectively. The first, second, and third elements of these are called the top left, bottom left, and right elements of the icon, respectively.

In giving this definition we have abstracted away from many features of real world circuit diagrams, like the fact that the icons have particular shapes, or even the fact that the diagrams must be two-dimensional. These features have their own importance in a full account of the use of diagrams in hardware design, but they are not relevant to the logical issues which we are discussing, and are hence excluded from our current model.

There are also features of circuits that are relevant to our task but which are not captured in this definition. For example, our definition would allow a wire to connect a point to itself, or a point to serve as the right element of more than one gate icon. Clearly these kinds of sketches have no place in real design.

We have a choice about what to do about such models. One is to try to impose additional conditions on the syntax of our diagrams, as might be done in a traditional well-formedness definition. Another is to define what it means for a sketch to describe a device and then try to characterize which sketches describe devices. The second approach seems more interesting, so we follow it here. The remainder of this section defines other syntactic properties of circuit sketches that will be useful in this effort; the sketch characterization definitions are provided in Sect. 3.2.

Definition 21 Given a circuit sketch s, there is a *simple connection* between connection points p_a and p_b, denoted $p_a \leadsto p_b$ if p_a is a left element and p_b the right element of some member of $W \cup N \cup D \cup A \cup R$.[6]

Definition 22 Given a circuit sketch s, a *connecting path* from connection point p_a to connection point p_b is a finite chain of simple connections

$$p_a = p_0 \leadsto p_1 \leadsto p_2 \leadsto \ldots \leadsto p_n = p_b$$

Definition 23 1. Given circuit sketches s_1 and s_2, a bijection ϕ from the connection points of s_1 to the connection points of s_2 is an *isomorphism between* s_1 *and* s_2 iff

   (a) $phi(I_1) = I_2$.

   (b) $phi(O_1) = O_2$.

   (c) $\langle p_a, p_b \rangle \in W_1 \leftrightarrow \langle \phi(p_a), \phi(p_b) \rangle \in W_2$.

   (d) $\langle p_a, p_b \rangle \in N_1 \leftrightarrow \langle \phi(p_a), \phi(p_b) \rangle \in N_2$.

   (e) $\langle p_a, p_b \rangle \in D_1 \leftrightarrow \langle \phi(p_a), \phi(p_b) \rangle \in D_2$.

   (f) $\langle p_a, p_b, p_c \rangle \in A_1 \leftrightarrow \langle \phi(p_a), \phi(p_b), \phi(p_c) \rangle \in A_2$.

   (g) $\langle p_a, p_b, p_c \rangle \in R_1 \leftrightarrow \langle \phi(p_a), \phi(p_b), \phi(p_c) \rangle \in R_2$.

2. Circuit sketches s_1 and s_2 are *isomorphic* iff there exists an isomorphism between them.

Definition 24 Circuit sketch s_1 is a *subsketch* of circuit sketch s_2, denoted $s_1 \subseteq s_2$ iff $P_1 \subseteq P_2$, $I_1 \subseteq I_2$, $O_1 \subseteq O_2$, $W_1 \subseteq W_2$, $N_1 \subseteq N_2$, $D_1 \subseteq D_2$, $A_1 \subseteq A_2$, and $R_1 \subseteq R_2$. Whenever $s_1 \subseteq s_2$, we say that s_2 is an *extension* of s_1.

3.2.1 Semantics

We can now use our model of physical hardware devices to give semantics for circuit sketches. We will define what it means for a circuit sketch to describe a device and what it means for a device to implement a circuit sketch. The distinction between these two notions will be that diagrams can describe a device either partially or fully, while devices do not partially implement diagrams. The separation of these notions is consistent with the notion that design is building up of sketches whose limit is the diagram one wants to implement. In addition, we will use our definitions of these notions to give well-formedness conditions on circuit sketches.

[6] We are thus using $\leadsto$ ambiguously with both devices and circuit sketches. This overloading should not cause any confusion since context will always make it clear which notion is being used.

Definition 25 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) $p \in s_I \rightarrow \phi(p) \in D_I$.
   (b) $p \in s_O \rightarrow \phi(p) \in D_O$.
   (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) $g = \langle p_a, p_b, p_c \rangle$ 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. s *describes* D, written $D \models s$, if there is a depiction map ϕ from s to D.

3. 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 hold. This is written $D \models_s s$ (the subscript s stands for "structural").

4. 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 $D \models_b s$.

Notice that $D \models_s s$ entails $D \models s$ and $D \models_b s$ but these latter two notions are incomparable.

Lemma 4 *If $D \models s$ and s_0 is a subsketch of s, then $D \models s_0$.*

Notice that this lemma will not hold with $\models$ replaced by $\models_s$ or $\models_b$.

Theorem 1 *Let s be a circuit sketch. s describes some well-connected device if and only if s has the following properties:*

1. *The elements of $N \cup D \cup A \cup R$ are all pairwise disjoint with respect to the sets of connection points they contain; these connection points are also disjoint from I and O.*

2. *There is no connection point p in s that is the right element of more than one tuple in W.*

3. *There is no connection point p_a in s such that $\langle p_a, p_b \rangle$ and $\langle p_c, p_a \rangle$ are both wire lines in s.*

4. *There is no $\langle p_a, p_b \rangle$ in W such that p_a and p_b are both right elements for tuples in $N \cup D \cup A \cup R$.*

5. *Any cyclic path from connection point p_a to connection point p_b contains an element $\langle p_c, p_d \rangle$ of D.*

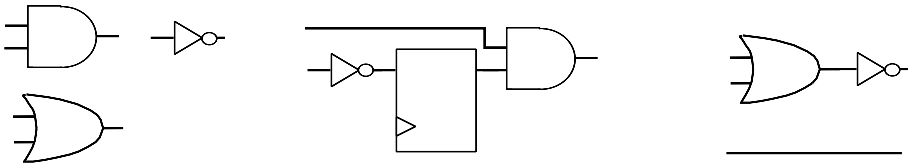


Figure 4: Well-formed circuit sketches; the first and third are incomplete.

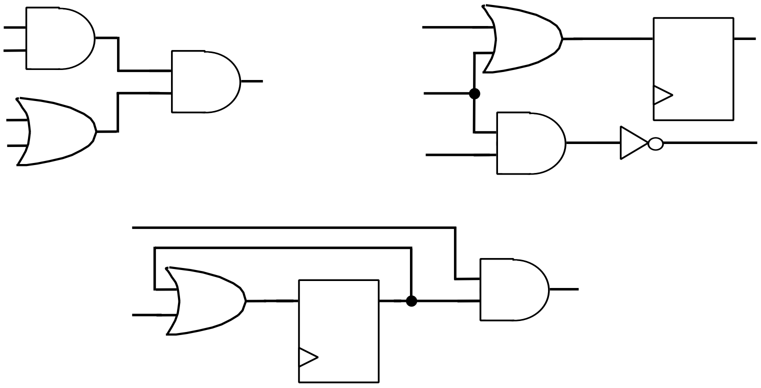


Figure 5: Examples of complete well-formed circuit sketches.

Theorem 2 *Let s be a circuit sketch. s is implemented by some well-connected device if and only if it has the properties of theorem 1 plus the following properties:*

1. *Treating the connection points as nodes and the elements of $W \cup N \cup D \cup A \cup R$ as denoting edges renders a connected and directed graph; in other words, the entire sketch must represent a single connected hardware component.*
2. *For all connection points p, there must be a directed path from p to a connection point in O.*
3. *For all connection points p not in I, p is the right element for some tuple in $W \cup N \cup D \cup A \cup R$.*
4. *For all connection points p not in O, p is the left, top left, or bottom left element for some tuple in $W \cup N \cup D \cup A \cup R$.*

Notice that the conditions in each of the above theorems are entirely syntactic; that is, they make reference only to the internal structure of sketches.

Definition 26 A *well-formed* circuit sketch is a sketch satisfying the conditions of 1 in the above theorem. A *complete* well-formed circuit sketch is a sketch that satisfies both conditions 1 and 2 of the theorem. We will reserve the term *circuit diagram* for complete well-formed circuit sketches.

Lemma 5 *For any device D there exists a circuit diagram C that it unique up to isomorphism on circuit diagrams such that $D \models_s C$.*

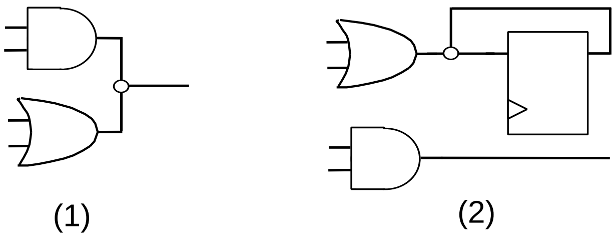


Figure 6: Examples of circuit sketches that are not well-formed. Example 1 has a wire connecting the right elements of two gates. Example 2 has two problems: the circuit is not connected and there is no path from the right element of the delay to a connection point in O.

3.3 ASM Charts

ASM charts are a variant of state machine that combines 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 assignments of T or F to signal variables. Examples of ASM graphs appear in Fig. 7; more extensive examples and details can be found in [13].

When using an ASM chart for computation, each state is viewed as lasting a single tick of a system clock, with the underlying hardware for a state being combinational in nature. Assuming we have a current state of the ASM and a function indicating the value of each signal used in a conditional branch in the ASM, we can easily determine the next state of the ASM by following the control paths in the chart. Any conditional output encountered along the satisfied control path will take effect for the current state.

The order in which conditional tests are made in a state is irrelevant to our exploration. What is relevant are the overall conditions under which we move from one state to another during computation (the transitions), and under which we produce certain outputs in the system. Our model captures only these relevant details. 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.

Definition 27 An *ASM graph* g is a tuple $\langle T, B, O, N, R, P\rangle$ satisfying the following conditions:

1. T is a set of objects called the states of g. B is a set of conditional branch objects and O is a set of conditional output objects. These sets are all pairwise disjoint. We will refer to the union of these sets as the *objects* of g.

2. N is a set of signal names.

3. R is a subset of $T \times T \times \mathcal{P}(N) \times \mathcal{P}(N)$ called the *next state transitions* of g.[7] 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.

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

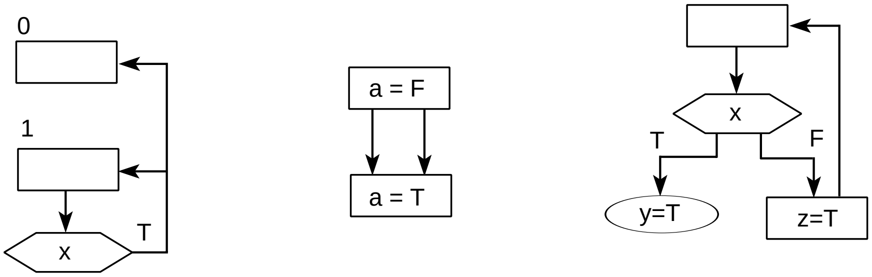


Figure 7: Examples of ASM graphs; additional examples are provided in the semantics section.

4. P is a subset of $T \times N \times U \times \mathcal{P}(N) \times \mathcal{P}(N)$ called the *output conditions* of g. The first two elements are called the *assertion state* and *asserted variable*, respectively. The third element is called the *assignment value* and is a member of the universe U of possible signal values, in this case $\{T, F\}$. The last two elements are called the *true conditions* and the *false conditions*, respectively.

Definition 28 Given an ASM graph g, the *external signals* of g are those elements of N that appear in the true or false conditions of some next state transition or output condition of g but are not the asserted variable for any output condition in g. Any signal in N that is not external is classified as *internal.*

Definition 29 1. Given ASM graphs g_1 and g_2, a bijection ϕ_o from the objects of g_1 to the objects of g_2 and a bijection ϕ_n from N_1 to N_2, $\langle \phi_o, \phi_n \rangle$ is an *isomorphism between* g_1 *and* g_2 iff

(a) $\phi_o(T_1) = T_2$.

(b) $\phi_o(B_1) = B_2$.

(c) $\phi_o(O_1) = O_2$.

(d) $\langle t_1, t_2, c_t, c_f \rangle \in R_1 \leftrightarrow \langle \phi_o(t_1), \phi_o(t_2), \phi_n(c_t), \phi_n(c_f) \rangle \in R_2$.

(e) $\langle t, n, u, c_t, c_f \rangle \in P_1 \leftrightarrow \langle \phi_o(t), \phi_n(n), u, \phi_n(c_t), \phi_n(c_f) \rangle \in P_2$.

2. ASM graphs g_1 and g_2 are *isomorphic* if there exists an isomorphism $\langle \phi_o, \phi_n \rangle$ between them.

Definition 30 ASM graph g_1 is a *subgraph* of ASM graph g_2 if and only if $T_1 \subseteq T_2$, $B_1 \subseteq B_2$, $O_1 \subseteq O_2$, $R_1 \subseteq R_2$, and $P_1 \subseteq P_2$. In this case, we call g_2 an *extension* of g_1.

3.3.1 Semantics

ASM graphs depict the algorithms computed by devices, capturing the control flow of devices but ignoring the structural details as to how the associated algorithms are computed in the devices. Given the level of abstraction associated with ASM graphs, the modelling relationships between ASM graphs and devices will be considerably more complicated than those required for circuit diagrams. Similarly to the section on circuit diagram semantics, we will define what it means for an ASM graph to describe a device and what it means for a device to implement the algorithm reflected in an ASM graph.

Definition 31 Let g be an ASM graph. A *signal-value assignment* for g is a function from the names N of g to the set $\{T, F\}$. An *external signal-value assignment* is the restriction of a signal-value assignment to the external signals of g.

Definition 32 Let g be an ASM graph, t a state in g, and i a signal-value assignment.

1. A next-state transition $\langle t_s, t_t, c_t, c_f \rangle \in R$ is *satisfied by* t *and* i if $t_s = t$, $i(n) = T$ 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 t and i, this transition is called the *next state of* g *under* t *and* i.
2. Output condition $\langle t_s, n, u, c_t, c_f \rangle \in P$ is *satisfied by* t *and* i if $t_s = t$, $i(v) = T$ for all names v in c_t, and $i(v) = F$ for all names v in c_f.
3. 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 t and i, $i'(n) = u$, $i'(s) = i(s)$ if s is an external signal and $i'(s) = F$ for all other signals s. i' is called the *signal update of* g *under* t *and* i.[8]

Definition 33 An ASM graph g is *deterministic* if no two next state transitions are satisfied by the same state t and signal-value assignment i.

Definition 34 An ASM graph g is *transitionally complete* if for all states t and all signal-value assignments i there exists a next-state transition that is satisfied by t and i.

ASM graphs developed in practice sometimes contain diagrammatically unreachable states. These states are included in the graph only because the state of a device is unpredictable at startup time and the ASM graph needs to indicate a transition in the event that the corresponding device starts up in such a state.[9] While ASM graphs are more complete with these states included, their omission often simplifies the picture, allowing for a clearer view of the intended algorithm, without sacrificing functionality.[10] The following definition captures the removal of such startup states.

Lemma 6 *Given ASM graph g' there is a unique ASM graph g such that*

1. $B = B'$, $O = O'$, $N = N'$, *and* $P = P'$.

2. $T \subseteq T'$.

3. for all $t \in T' - T$,

   (a) There are no output conditions in P with source state t.

   (b) There is only one next state transition in R with source state t.

Definition 35 Given ASM graph g', the ASM graph g satisfying the conditions of lemma 6 is called the *functional reduction* of g'.

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

[9] Note that this notion of unreachable does not capture states that are unreachable at computation time; some states may never be reached due to the particular input values of the tested signals. We are only concerned with diagrammatically unreachable states for the present discussion.

[10] Some models of ASM graphs and state machines include a specific start state in order to address this issue.

We have established sufficient framework to discuss when a given ASM graph describes a given physical device and when a given device implements the algorithm depicted in an ASM graph. First, we need to establish relationships between the signals and states of ASM graphs and the ports and states of devices.

Definition 36 Let g be an ASM graph and D be an abstract device.

1. A *state map* from g to D is a function from the states T of g to the possible states of D.

2. Let ϕ be a function from the signal names of g to the ports of D. ϕ 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 input interface port and every internal output port wired to an output interface port of D in its co-domain.

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. We also need to know when there is a correspondence as just described. These two notions are captured in the following definitions.

Definition 37 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.

Definition 38 A state map ϕ_t 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 t in g there exists an assignment a for D which is compatible with ϕ_n and i and reflects state $\phi_t(t)$ such that:

1. If t' is the next state of g under t and i, then $\phi_t(t')$ is the next state of D under $\phi_t(t)$ and a.

2. If i' is the signal update of g under t and i, then assignment a' derived from $\langle D, a\rangle$ is compatible with i'.

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

As in the section on circuit diagrams, we now define four relationships between ASM graphs and devices that capture the various granularities of relationships between them. Using these definitions, we can identify properties of ASM graphs required for implementation in devices. There are various algorithms for implementing an ASM graph in physical hardware; the interested reader is referred to [13] for examples of such algorithms.

Definition 39

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 *full structural implementation* of g iff there exists a surjective state map and a complete signal map that capture g and D. This is written $D \models_s g$.

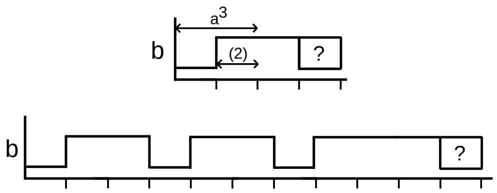


Figure 8: The diagram on the top demonstrates the notation for interval durations in the logic. The diagram on the bottom shows the expansion of the upper diagram with respect to the duration markings.

3. D is a *functional structural implementation* of g iff there exists an ASM graph g' such that D is a full structural implementation of g' and g is the functional reduction of g'. We will denote this relationship as $D \models_{fs} g$.

4. 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 $D \models_b g$.

Theorem 3 *An ASM graph g describes some well-connected device iff g is deterministic.*

Theorem 4 *An ASM graph is fully structurally implemented by a well-connected device iff it is deterministic, transitionally complete, and contains 2^k states for $k \geq 1$.*

Knowing which ASM graphs can be realized in physical devices leads us to make the following definition:

Definition 40 A deterministic ASM graph will be called an *ASM chart.*

3.4 Timing Diagrams

Timing diagrams are perhaps the easiest to model because there is little information contained in the diagram that is irrelevant to our exploration. We will view a timing diagram as a collection of individual waveforms specified over the same set of time ticks. Our notation for timing diagrams supports the specification of intervals which are useful not only for naming sections of a diagram, but also for specifying repetitions of segments of a diagrams. For the latter purpose, a duration marker may be placed within parentheses on the arrow marking an interval, as shown in Fig. 8. The valid duration markers are positive integers, variable names, and the sign +, denoting an unknown (possibly infinite) number of repetitions.

Our model of an individual waveform needs to take into account the signal levels depicted at each time tick; in our notation, there are three valid levels, high, low, and unknown. There are also two line styles, solid and dotted, used to distinguish between the levels that a signal can and cannot have at a time tick, respectively. Finally, we use color on waveforms to classify them as representing input, output or internal signals.

There are only two pieces of information reflected in timing diagrams which we take as irrelevant in our models. The first is the ordering of signals within the diagram. The second is the amount of space between ticks in a timing diagram. Given that we are working with purely synchronous systems, this latter piece of information is indeed irrelevant for our investigation. The syntax rules for constructing timing diagrams are simplistic enough that we will not need separate definitions for well-formed and non-well-formed diagrams as we did in the cases of circuit diagrams and ASM charts; we will simply use the term timing diagram throughout the section.

Definition 41 A *timing pattern* is a tuple $\langle K, v, x, c\rangle$ where

1. K is a set of ordered time ticks, represented by integers.
2. v is a function from K to the values set {high, low, unknown}.
3. x is a function from K to the exclusion set {excluding, non-excluding}.
4. c is an element of the set of possible colors.

Definition 42 Timing patterns p_1 and p_2 are *equivalent* for a set of time ticks K if for all time ticks $k \in K$, $v_1(k) = v_2(k)$ and $x_1(k) = x_2(k)$.

Definition 43 A *timing diagram* is a tuple $\langle N, P, K, I\rangle$ satisfying the following conditions:

1. N is a set of names.
2. P is a function from N to timing patterns.
3. K is an ordered set of time ticks represented by integers such that K contains the ticks of each timing pattern in the range of P.
4. I is a subset of $K \times K \times D$ where D is a set of duration markers for intervals. The first two elements are called the start tick and end tick of the interval, respectively; the last element is called the duration of the interval.

Definition 44 Given two timing diagrams T_1 and T_2, a bijection ϕ from N_1 to N_2 is a *pattern isomorphism* if for all names $n \in N_1$, $P_1(n)$ is equivalent to $P_2(\phi(n))$. T_1 and T_2 are *isomorphic* if $K_1 = K_2$, $I_1 = I_2$, and there exists a pattern isomorphism between N_1 and N_2.

Definition 45 Timing diagram T_1 is a *subdiagram* of timing diagram T_2 if and only if $N_1 \subseteq N_2$, $I_1 \subseteq I_2$, $K_1 \subseteq K_2$, $c_1 = c_2$ and for all names $n \in N_1$, $P_1(n)$ is equivalent to $P_2(n)$ for all time ticks in K_1. If T_1 is a subdiagram of T_2, then T_2 is called an *extension* of T_1.

As demonstrated in Fig. 8, it is possible to remove intervals from the diagram by reproducing the portion of the diagram within the interval as many times as is specified by the duration marking on the interval. Along the same lines, we will often want to take an interval of variable duration and instantiate it with a particular duration for purposes of proofs. A useful application of both techniques appears in the example proofs later in the paper. The following definitions capture the syntactic conditions under which one diagram is such an expansion or instantiation of another.

Definition 46 Timing diagram T_1 is an *instantiation* of waveform collection T_2 if $N_1 = N_2$ and for each $\langle k_s, k_e, d\rangle$ in $I_2 - I_1$, one of the following cases holds:

1. $d = 1$.

2. d is a duration variable and $\langle k_s, k_e, d' \rangle \in I_1$ where d' is a positive integer.

3. d is the marker $+$ and $\langle k_s, k_e, d' \rangle \in I_1$ where d' is a positive integer or a duration variable.

Definition 47 A timing diagram is *fully instantiated* if it contains no non-numeric interval duration labels.

Definition 48 Timing diagram T_1 is a *trivial expansion* of waveform collection T_2 if the following conditions are satisfied:

1. $N_1 = N_2$.

2. $I_1 = (I_2 - \{\langle k_s, k_e, d \rangle\}) \cup \{\langle k_e, k_e + (k_e - k_s), d - 1 \rangle\}$ or $I_1 = (I_2 - \{\langle k_s, k_e, d \rangle\}) \cup \{\langle k_s, k_e, d - 1 \rangle\}$ where d is not the marker $+$ and $d > 1$ and for all timing patterns $P_1(n)$ and $P_2(n)$ they are equivalent:

   (a) Up to k_e.

   (b) From k_e to $k_e + (k_e - k_s)$ in $P_1(n)$ with k_s to k_e in T_2.

   (c) from $k_e + (k_e - k_s)$ to the end in $P_1(n)$ with k_e to end in T_2.

Definition 49 Timing diagram T_n is an *expansion* of timing diagram T_1 if there exists a sequence of timing diagrams $T_1 \ldots T_n$ such that for all $1 > i \geq n$ T_i is either an instantiation or a trivial expansion of T_{i-1}.

3.4.1 Semantics

The distinction between description and implementation is less pronounced when working with timing diagrams because their very nature tends towards partial information. Timing diagrams rarely depict all possible combinations of values on signals; only those value combinations of interest to the problem at hand are represented. Similarly, timing diagrams often include only those signals for which there are explicit timing relationships to consider, leaving out the remaining signals in a device. Our distinction between description and implementation will be based on the latter consideration — the number of signals depicted relative to a given device.

Definition 50 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 unknown, then it corresponds to any value of v_d that lies in the universe of values for devices.

Definition 51 Given a timing diagram T and a device D, a *waveform map* is an injective function from the signal names in T to the ports of D such that:

1. Input signals in T map to input interface ports of D.

2. Output signals in T map output interface ports of D.

3. Internal signals in T map to internal output ports of D.

Determining whether or not a timing diagram describes the behavior of a device requires that we be able to check the values on wires in the device against the values on signals in the timing diagram. This definition captures this check for an individual tick in a timing diagram.

Definition 52 Given an assignment a to the ports of a device D, a time tick k in a timing diagram T, and a waveform map ϕ from T to D, *a matches k* if for all names n in the domain of ϕ:

1. If $P(n)_x$ is non-excluding at tick k, then $a(\phi(n))$ must correspond to $P(n)_v$ at k.
2. If $P(n)_x$ is excluding at tick k, then $a(\phi(n))$ must not correspond to $P(n)_v$ at k.

We now extend our terminology to cover comparing an entire timing diagram to the functionality of a physical device. Our approach is to generate an assignment sequence for the device using the values depicted for the input signals in the timing diagram, then match the outputs generated by the device (under the timing diagram inputs) to the outputs reflected in the timing diagram. This approach is captured in the next three definitions.

Definition 53 Given a fully instantiated timing diagram T with k time ticks and a waveform map ϕ from T to device D, the *generated assignment sequence* for D under T and ϕ is an assignment sequence $a_1, \ldots, a_k$ such that:

1. if $P(n)_v$ is low at tick i, $a_i(\phi(n)) = 0$.
2. if $P(n)_v$ is high at tick i, $a_i(\phi(n)) = 1$.
3. if $P(n)_v$ is unknown or undefined at tick i, $a_i(\phi(n))$ can be any value in the universe U.

Definition 54 Let D be a device, T be a fully instantiated timing diagram, and ϕ be a waveform map from T to D. Let $a_1, \ldots, a_k$ be the generated assignment sequence for D under T and ϕ. T and D are said to be *compatible* under ϕ if there exists concrete device $\langle D, a' \rangle$ such that for each R_i in the run of D under $a_1, \ldots, a_k$ for D under T and ϕ, the generated b-assignment of R_i matches tick i in T.

Definition 55 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 from T to D and a concrete device D' for D such that all fully instantiated timing diagrams for T are compatible with D'.
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 $D \models_b T$.

Theorem 5 *Any fully instantiated timing diagram describes some well-connected device.*

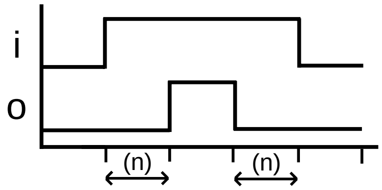


Figure 9: Timing diagram that cannot be implemented by a physical device.

Proof Any fully instantiated timing diagram can be expanded into a timing diagram with no duration markers larger than 1. For each output waveform in the collection, a temporal expression can be written that contains a disjunct for each time tick t where the waveform is true; the disjunct for time t is a conjunct of the values of each non-output signal name at each time $t_0 < t$.

Taking the conjunction of the expressions for each output waveform yields an expression that corresponds to the output patterns in the timing diagram. We can implement this conjunction in a device and we claim that the original timing diagram describes this device by construction. □

Theorem 6 *There exist timing diagrams that cannot be described by any well-connected device.*

Proof As pointed out in [8], the timing diagram for a single-pulser that generates the output pulse in the middle of the input pulse is such a timing diagram; such a diagram is given in Fig. 9. □

4 Rules of Inference

Using the semantics developed in the previous four sections, we can now state the conditions for creating rules of inference, such as the sample ones given in Sect. 2. Consider two diagrams G_1 and G_2 which may or may not be diagrams of the same type. If it is the case that whenever $D \models G_1$, then $D \models G_2$ for every device D, then a rule can be formulated to infer G_2 from G_1. Similarly, given a set S of diagrams (not necessarily all of the same type), if whenever a device D models every diagram in S then D also models some diagram G_1, then G_1 can be inferred from S.

While this requirement does not dictate which of the modeling relationships (structural or behavioral) should be used in defining rules of inference, in this logic behavioral modeling is used to create rules between diagrams of the same type and structural modeling is used to create rules between diagrams of different types. As a further example of inference rules, Fig. 10 contains all the primitive inference rules needed between timing diagrams and circuit diagrams; while we do not provide the proof here, these inference rules are sound within the logic. Other inference rules that we might expect to see between circuit diagrams and timing diagrams can be derived from this basic set, as can rules for other components that are constructed out of the four basic gates. In addition, the logic will contain inference rules involving the other representations, as well as rules bridging diagrammatic and sentential representations; examples appear in Fig. 11.

5 Related Research

Using visual representations in hardware design and verification is not a new idea. Various design tools and description languages have employed diagrammatic representations [6] [7] [15], and

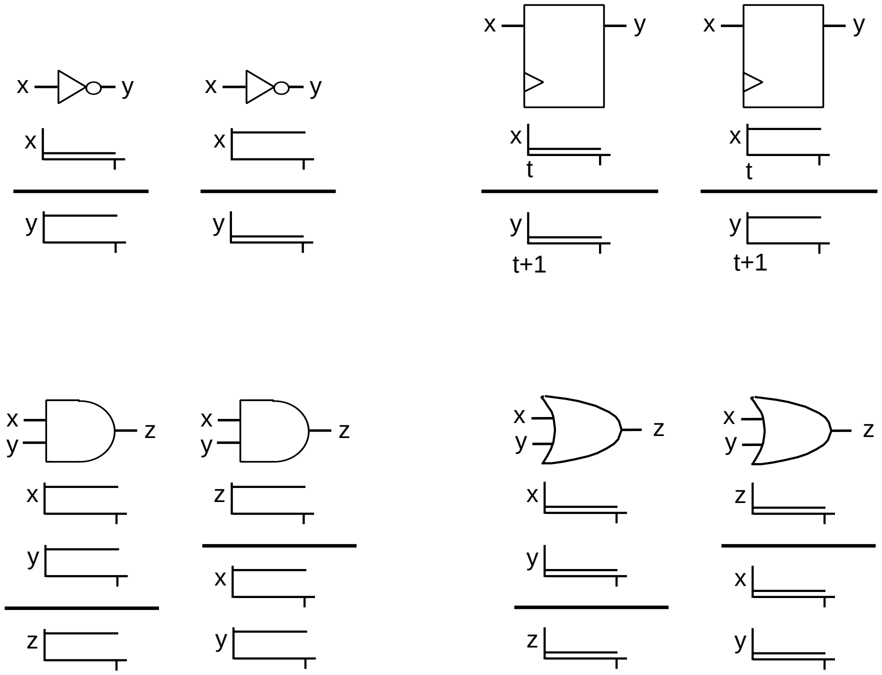


Figure 10: Primitive inference rules between circuit and timing diagrams.

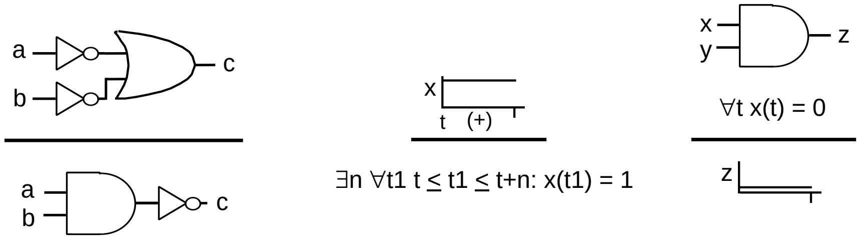


Figure 11: Examples of additional inference rules in the logic. The first rule could be derived from the rules given in Fig. 10. The second is an example of a primitive rule between timing diagrams and our sentential logic. The third rule would be derived from the primitive rules involving circuit diagrams, timing diagrams, and sentential logic.

systems for reasoning about some aspects of systems using diagrammatic representations have appeared over the past year [4] [3] [10] [14]. Many systems provide formalizations of timing diagrams [3] [10] [14] and some even provide formal definitions of the interaction between timing diagrams and sentential representations [14]; none of these support multiple diagrammatic representations. Another distinguishing feature of this research is that the logic has been developed *directly on the diagrammatic forms* rather than on an underlying sentential logic. [4] present a system in which a user can reason about system states using a graphical interval logic, but they translate their visual representations into a sentential logic for purposes of formal manipulation. Designing the logic directly at the level of the diagrams is more powerful because it lets the diagrams dictate the logical structure as opposed to having the constraints of a conventional logic dictate the visual representations that can be supported.

A previous attempt at defining a heterogeneous logic for hardware, along with arguments supporting the use of diagrams in hardware formal methods is presented in [8]. The logic in [8] is less fine-grained than the one presented here; their logic is based only on behavioral relationships while this work allows for reasoning about structural relationships between components.

6 Conclusions

Diagrams offer several potential advantages to hardware reasoning: they offer clear, compact, and user-transferable representations, and they lack the high learning overhead associated with the formal logics underlying many state-of-the-art sentential reasoning tools. We have provided a logical formalization of three types of hardware diagrams and presented an example of how such a logic can be used for hardware verification.

We are in the process of constructing a verification tool based upon a heterogeneous logic that contains the formalisms presented here as well as support for higher-order logic as a valid representation. Initially, this tool will take the form of a proof checker. We believe that such heterogeneous tools will provide the flexibility to make verifications not only more concise and readable, but also easier to produce than those performed in entirely sentential systems.

References

[1] Jon Barwise. Heterogeneous reasoning. In G. Mineau, B. Mouline, and J. Sowa, editors, *Conceptual Graphs and Knowledge Representation.* Springer-Verlag, 1993.

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

[3] Viktor Cingel. A graph-based method for timing diagrams representation and verification. In George J. Milne and Laurence Pierre, editors, *Correct Hardware Design and Verification Methods*, pages 1–14. CHARME, Springer-Verlag, 1993.

[4] 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.

[5] Ruth Eberle, April 1994. Personal communication.

[6] Simon Finn, Michael P. Fourman, Michael Francis, and Robert Harris. Formal system design — interactive synthesis based on computer-assisted formal reasoning. In Luc Claesen, editor, *Formal VLSI Specification and Synthesis: VLSI Design-Methods-I.* North-Holland, 1990.

[7] Graham Hutton. The Ruby interpreter. Technical Report 72, Chalmers University of Technology, May 1993.

[8] Steven D. Johnson, Gerard Allwein, and Jon Barwise. Toward the rigorous use of diagrams in reasoning about hardware. IULG Preprint Series, April 1993.

[9] Steven D. Johnson, Paul Miner, and Shyam Pullela. Studies of the single-pulser in various reasoning systems. In *Theorem-Provers and Circuit Design Proceedings*, September 1994.

[10] K. Khordoc, M. Dufresne, E. Cerny, P.A. Babkine, and A. Silburt. Integrating behavior and timing in executable specifications. In *CHDL*, pages 385–402, April 1993.

[11] Thomas Frederick Melham. Formalizing abstraction mechanisms for hardware verification in higher order logic. Technical Report TR 201, University of Cambridge Computer Laboratory, August 1990.

[12] Paul S. Miner, July 1994. Personal communication.

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

[14] Rainer Schlör and Werner Damm. Specification and verification of system-level hardware designs using timing diagrams. In *Proc. European Conf. on Design and Automation*, Paris, February 1993.

[15] Mandayam Srivas and Mark Bickford. SPECTOOL: A computer-aided verification tool for hardware designs, vol I. Technical Report RL-TR-91-339, Rome Laboratory, Griffiss Air Force Base, NY, December 1991.

A PVS Proof Trace

This PVS proof trace of the single-pulser verification was provided by Steve Johnson and originally appeared in [9].

Verbose proof for `single_pulse1`.

`single_pulse1:`

---

[1] $(\forall\ (o, i : \text{signal}) : \text{imp}(i, o) \supset \text{spec1}(i, o))$

Expanding the definitions of spec1, imp, delay, and$\overset{\circ}{\bullet}$, $-$, Pulse
Applying (SKOSIMP*),
Instantiating the top quantifier in + with the terms: m!1
Applying propositional simplification and decision procedures,
which yields 2 subgoals:

`single_pulse1.1:`

$\{-1\}$ $(\forall\ (t : \text{time}) : (x'(t + 1) = i'(t)))$
$\{-2\}$ $(\forall\ (t : \text{time}) : (o'(t) = (1 - i'(t)) \times x'(t)))$
$\{-3\}$ $n' < m'$
$\{-4\}$ $i'(n' - 1) = 0$
$\{-5\}$ $i'(m') = 0$
$\{-6\}$ $(\forall\ (t : \text{time}) : (n' \leq t \wedge t < m' \supset i'(t) = 1))$

$\{1\}$ $o'(m') = 1$

`single_pulse1.2:`

$\{-1\}$ $n' \leq j'$
$\{-2\}$ $j' \leq m'$
$\{-3\}$ $o'(j') = 1$
$[-4]$ $(\forall\ (t : \text{time}) : (x'(t + 1) = i'(t)))$
$[-5]$ $(\forall\ (t : \text{time}) : (o'(t) = (1 - i'(t)) \times x'(t)))$
$[-6]$ $n' < m'$
$[-7]$ $i'(n' - 1) = 0$
$[-8]$ $i'(m') = 0$
$[-9]$ $(\forall\ (t : \text{time}) : (n' \leq t \wedge t < m' \supset i'(t) = 1))$

$\{1\}$ $j' = m'$