Proof and Design

Machine-assisted Formal Proof as a Basis for Behavioural Design Tools

Michael P. Fourman

Laboratory for Foundations of Computer Science,
The University of Edinburgh,
King's Buildings,
Edinburgh EH9 3JZ
Michael.Fourman@ed.ac.uk

Summary. We describe a formalisation of the design process, based on the use of predicate logic to model system behaviour. In our formalisation, derived rules of predicate logic are used to represent the design state, design steps are represented as proof steps, and a goal-directed proof-assistant is used as the basis for behavioural design tools.

We introduce the relational model, and discuss its relationship to more operational models. In particular, we show that it is sound (relative to these operational models) for circuits that obey common design rules. We show how derived rules in a sequent calculus may be used to represent the relationship between a partial design, the original specification, and a specification of the tasks remaining to complete the implementation. We show how the relational hardware model can be used to express and relate hardware descriptions at different levels of abstraction. We describe a formal framework for top-down design based on this formulation of refinement. Finally, we give an example of the application of these methods, developed using Abstract Hardware Limited's LAMBDA design system.

1. Introduction

The design engineer must consider many aspects of a system. At the logical level he deals with the correct implementation of Boolean functions, at the electrical level, with voltages, delays and power, and at the physical level with placement and routeing. Tools are available that successfully manage these logical, electrical and physical concerns to produce working silicon systems with millions of transistors on a chip.

Logical design is increasingly separated from technological issues, and it is possible to design 'abstract hardware' at a technology-independent register-transfer level (RTL). This level of design may be expressed in a hardware description language (HDL). 'High-level synthesis' tools are available to automatically generate implementations, in a variety of technologies, from RTL descriptions.

To quote from one high-level-synthesis supplier,

> *"The input to HDL synthesis is an RTL functional description of the network to be implemented. RTL descriptions are distinct from*

> *behavioural descriptions. The distinction is that ... an RTL level description has a definite implied architecture."* [Carlson, 1991]

At the behavioural level, the designer is on his own; there are few tools to support the development of RTL descriptions from more-abstract behavioural specifications. But this level of design is crucial.

> *"The areas in which the designer can exert the most influence using style as a control mechanism are in describing the sequential timing/clocking schemes, utilising design hierarchy, and taking advantage of predefined blocks."* [*ibid.*]

In many application areas, the rapid development of silicon systems is now limited by this lack of tools that support behavioural design.

Placement and routeing tools are based on Euclidean geometry. Tools at the electrical level are based on analytical models of the behaviour of transistors and interconnect, manipulated using methods that often go back to Leibniz and Newton. We are concerned with the *high-level behaviour* of digital systems, which abstracts away the electrical and physical aspects. Current synthesis tools are primarily based on Boolean algebra and the theory of state machines. High-level behavioural tools require a corresponding mathematical foundation.

The behaviour of a digital system can be complex. To manage this complexity, we must consider many levels of behavioural *abstraction*. Many specialised calculi have been introduced for describing and reasoning about system behaviour at particular levels – for example, ISP [Bell and Newell, 1971], CCS [Milner, 1989], CSP [Hoare, 1985], Temporal Logic [Moskowski, 1985]. Higher-order, predicate logic is a general-purpose formalism that allows us to describe and relate both behaviour and structure at many different levels [Gordon, 1986].

We show how predicate logic may be used as the mathematical basis for tools that support interactive behavioural synthesis. We describe, and justify, the use of theorem-proving methods in the verification and design of digital circuits.

Outline

We start by introducing hardware devices and circuits. The behaviour of a circuit is given operationally by a simulation model. A circuit may be specified to implement a given function, and its correctness tested by simulation. However, exhaustive simulation is impossible, and reasoning about simulation models is intractable. Although our approach is aimed at high-level design, we also discuss a lower level – implementing gates as networks of transistors – to provide small, accessible examples of some of the subtleties of modelling and reasoning about behaviours.

We show that for reasoning about the correctness of well-formed circuits – those obeying a set of *design rules* – a relational model of behaviour is adequate. Specifications and behaviours of circuits are represented as predicates; an implementation is correct if the implemented behaviour is contained within the specified behaviour. We also show how this framework can be extended to incorporate busses with multiple tri-state drivers; these sub-circuits violate the design rules, but form an essential ingredient of many practical circuits.

We then address the formalisation of the design process. Design is a problem-solving task. The problem is to construct an artifact that, if correctly used, will satisfy a given design requirement, or specification. Some problems are trivial and have an immediate solution; others must be refined and decomposed into subgoals – simpler tasks which must be solved before the primary goal can be achieved.

In the process of design, we must manipulate behaviours, and generate, propagate, or discharge, constraints that specify in what environments the design may correctly be used. Since relations may be expressed as predicates, a proof-assistant for predicate logic can provide machine support for reasoning about behaviours. We show how a proof assistant for higher-order logic can be used as the basis of tools for the design and verification of hardware.

Thus we address the application of theorem-proving techniques to the *design* of digital systems, this treatment has been developed in a programme started in [Fourman *et al.*, 1987], and developed in [Fourman *et al.*, 1988] [Finn *et al.*, 1989] [Fourman and Mayger, 1989] [Mayger and Fourman, 1991] [Fourman and Hexsel, 1991]. A related (and for us inspirational) approach is described in [Hanna *et al.*, 1989].

2. Devices

Devices are the primitive building blocks from which we construct circuits. We begin by introducing, informally, a number of simple hardware devices. These will provide a stock of examples for later use. Devices have a number of ports – points at which wires may be connected. We may stimulate some ports with signals, and observe the response.

The simplest devices are *combinational;* they respond directly to each new stimulus, the response only depends on this and is independent of previous stimuli. We say combinational devices have no state.

A combinational *gate* implements a boolean function. For example, a **nand** gate, $\mathbf{nand}(a, b, out)$, has two input ports a, b, and one output port, *out*. If we place boolean values (a, b) on the input ports, then, after some delay, the value $\neg(a \wedge b)$ will appear on the output. We abstract away the delay involved in producing the correct output, and consider the gate as an implementation of the boolean function. A general combinational gate, for example an adder, computes a number of outputs as boolean functions of a number of inputs.

A bi-directional *switch,* $\mathsf{S}(g, a, b)$, is another combinational device. It has a control port, g, and two more ports, a, b, which are connected when the gate value is 1 and independent when it is 0. A *tri-state driver,* $\mathsf{D}(g, in, out)$, acts as a uni-directional switch; the gate controls a connection from input to output. MOS transistors act as less-ideal switches: an n-type transistor, $\mathsf{N}(g, a, b)$, can be switched on by a high signal (logic 1) on its gate, to convey a logic 0 (but will not effectively convey a logic 1); a p-type transistor, $\mathsf{P}(g, a, b)$, has the dual behaviour – it is switched on by a logic 0, to convey a logic 1. Our fundamental model of behaviour will allow us to distinguish the bi-directional switch, the tri-state driver, and the n-type transistor. At a more-tractable, more-abstract level, they are all viewed as switches.

A *sequential* device has state, and may change state at particular times called "clock ticks". The response of a sequential device may depend on the history of previous stimuli, which established the current state, as well as the current stimulus. A sequential device is often described as a finite-state machine, with functions giving the next state and outputs in terms of the current state and inputs.

For example, a basic *flip-flop,* $\mathrm{Dff}(s, r, q, \overline{q})$, has two inputs, and two outputs; a boolean state, q, and its complement, $\overline{q}$, are available as outputs. (The physical device also has a clock input, but in the abstract model this is implicit.) The next-state function of the flip-flop may be given as a table showing how the current state, q, and the boolean input values, (s, r), at a clock tick, determine the state, q', following the clock tick.

q	s	r	q'
x	0	0	x
x	0	1	0
x	1	0	1
x	1	1	$\overline{x}$

Of course, in practice there is some delay, after a clock tick, before the new state and output values are achieved. If the clock period is long enough, we may abstract away this delay; and consider only the discrete sequence of states of the device at successive clock ticks, in response to a sequence of inputs synchronised with the clock.

Devices may combine combinational, bi-directional, tri-state, and sequential behaviours. Synchronous circuits are networks of devices, with a common clock used for all the sequential components. The period between successive clock ticks is sufficiently long to allow the values in the network to stabilise. So we may ignore the delays inherent in the behaviour of the primitive devices, and compute the behaviour of a circuit by computing successive stable states.

3. Circuits

Circuits are constructed from components. A circuit may be used as a component of a more complex system. Devices provide a basic set of components. So a system may be presented as a hierarchical composition based on primitive devices.

The interface, or *sort* of a circuit is a finite set of ports. Circuits are made by assembling components, and connecting ports together with wires; some wires may then be hidden, others being left available for future connections.

We use a syntax, that has much in common with a fragment of first-order predicate logic, to describe circuits constructed in this way. A stock of variables is used to name wires. A device is represented by a *circuit expression* with free variables representing wires connected to its ports. Assembly of devices is represented using a binary operator $\|$, with connections along wires represented by shared variables. Hiding of an internal wire is represented by a quantifier H – internal wires become bound variables.

The circuit expression

$$\mathsf{H}q, \overline{q}, d.\ \mathsf{Dff}(d, clr, q, \overline{q}) \parallel \mathbf{inv}(q, d) \parallel \mathbf{nand}(\overline{q}, b, c)$$

represents *one* decomposition of a circuit into components. This example could be represented in several other ways; for example, as

$$\mathsf{H}a.\ \mathbf{nand}(a, b, c) \parallel \mathsf{H}i, o.\ \mathbf{inv}(i, o) \parallel \mathsf{Dff}(o, clr, i, a)$$

Two expressions are *equivalent* if they represent the same circuit. This equivalence may be generated by making the composition operator associative and commutative, moving hiding outside compositions, and renaming of bound variables.

Remark 3.1. The *behaviour* of this circuit will turn out to be the same as that of

$$\mathsf{H}a.\ \mathsf{Dff}(a, clr, q, a) \parallel \mathbf{nand}(a, b, c)$$

but this is not an equivalent circuit – equivalence is a purely structural matter.

This gives a syntactic presentation of circuits. We will also take a somewhat more algebraic view. Circuits may be constructed using three operations: assembly, renaming, and hiding. We describe these informally here – in an appendix we give a mathematical definition. Each circuit has a *sort*, U, which is a finite set of ports, the interface to the circuit. We write $\mathtt{ctt}_U$ for the collection of circuits of sort U.

Assembly We can assemble finitely many circuits of the same sort by connecting together the corresponding ports. Syntactically, we have a trivial circuit of each sort, which we write as T, representing an empty box with the appropriate interface, and a binary operation $\|$.

Renaming A renaming is a function $U \xrightarrow{f} V$ (not necessarily 1-1) from one set of ports to another. Given a circuit, $C \in \mathtt{ctt}_U$, imagine that we place it inside a box with interface V. We obtain a circuit $f^\star(C) \in \mathtt{ctt}_V$ by connecting each port $p \in U$ with the corresponding port, $f(p) \in V$.

Hiding We can also use a function $U \xrightarrow{f} V$ to construct a circuit of sort U from a circuit, $C \in \mathtt{ctt}_V$: we place C inside a box with interface U, and again connect each port $p \in U$ with the corresponding port, $f(p) \in V$ – this time from the outside in. We call this circuit $\mathsf{H}_f(C)$.

Examples

Parallel Composition The *parallel composition* of two circuits, $M \in \mathtt{ctt}_U$, and $N \in \mathtt{ctt}_V$ (given syntactically as $M(u)$ and $N(v)$, respectively), is a circuit of sort $U \times V$, given syntactically as $M(u) \parallel N(v)$, or algebraically as $\pi_1^\star M \parallel \pi_2^\star N$, using the obvious projections

$$U \xleftarrow{\pi_1} U \times V \xrightarrow{\pi_2} V$$

Serial Composition The **serial composition** of two circuits, $M \in \mathtt{ctt}_{U\times V}$, and $N \in \mathtt{ctt}_{V\times W}$ (given syntactically as $M(u,v)$ and $N(v,w)$, respectively), is a circuit of sort $U \times W$, given syntactically as $\mathsf{H}v.\ M(u,v) \parallel N(v,w)$, or algebraically as $\mathsf{H}_\pi(\pi_1^\star M \parallel \pi_2^\star N)$, using the projections

$$\begin{tikzcd} U \times V & U \times V \times W \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] \arrow[d, "\pi"] & V \times W \\ & U \times W & \end{tikzcd}$$

Wires Allowing hiding along arbitrary sort morphisms introduces wires to our circuits. Hiding the trivial circuit along a diagonal in ${S_{fin}}^{op}$ gives a wire, $W = \mathsf{H}_\Delta \mathsf{T}$. Syntactically, it is convenient to represent a wire as an infix equality relation: $W(a,b) \equiv a = b$.

4. Simulation Semantics

If we build a circuit, we can investigate its behaviour experimentally. Until recently, it was common to build a prototype "bread-board", whose behaviour could be tested, before implementing a design in production technology. Current practice replaces this by simulation: running a program intended to model the circuit's behaviour.

Circuit behaviour may be simulated at many levels. An electrical simulation may attempt to model the interactions of voltage and current over time, but this level of detail is not tractable for circuits of any complexity. Moreover, the uncertainties of the production process lead to variations in electrical parameters. The results of an electrical simulation are thus approximate; *a priori,* we must suppose that, over an extended simulation, they diverge from reality.

When we use a digital circuit, we are not concerned with every detail of the waveforms on its ports; we consider only a more abstract, qualitative, description. For example, to reason about the logical behaviour of a synchronous circuit we generally consider only whether the voltages on wires "at clock ticks" are "high" or "low". Practical circuits have behaviours that are qualitatively stable; although the detailed electrical behaviour will vary with process variations, the abstract behaviour is invariant. We describe a simulation model at this abstract level.

Our simulation model formalises the description of the behaviour of synchronous boolean circuits given earlier. The idea is that we can compute information about the results on the outputs of a circuit, given information about the values on the inputs. However, our account is more general; for example, it allows us to model bi-directional behaviours, and tri-state drivers. We describe how a circuit responds to a given stimulus. Later we will distinguish inputs and outputs semantically.

Combinational Logic

First consider purely combinational behaviour. We want to account for the possible boolean values on a wire. Our simulator uses a set $\mathcal{B}$ of four values. In addition to the Booleans, $\{0, 1\}$, there are two new "values": the value on a floating wire, one that is not driven, is represented by Z; the value on a short-circuited wire, one that is driven by conflicting outputs, is denoted by $\mathsf{!}$.

The elements of $\mathcal{B}$ represent the information we have about the value on a wire. We may have no information, represented by Z, good information, represented by the appropriate boolean, or self-contradictory information, represented by $\mathsf{!}$. We use $\sqsubseteq$ for the *information ordering,* which makes the booleans incomparable, places Z below them, and $\mathsf{!}$ above them.

For a circuit of sort U, a stimulus is represented as a valuation, $v \in \mathcal{B}^U$, giving the information available about the value at each port. To compute the response to a stimulus, our simulator uses a model of each primitive device to derive new information about the values on wires from the information provided. This step is iterated until a stable configuration – another element of $\mathcal{B}^U$ – is found.

So the behaviour of the circuit is represented as a map $m : \mathcal{B}^U \xrightarrow{m} \mathcal{B}^U$, giving the response to any stimulus. The idea is to represent the *increased* information available from observation of the *stable* response to a given stimulus. Furthermore, more information about the stimulus should give more information about the response. This leads us to the following definition.

Definition 4.1. *A* combinational behaviour *is a map* $m : \mathcal{B}^U \xrightarrow{m} \mathcal{B}^U$ *that is*

inflationary – $v \sqsubseteq m(v)$
idempotent – $m(v) = m(m(v))$
monotone – $v \sqsubseteq w \Rightarrow m(v) \sqsubseteq m(w)$

Example 4.1. We give some examples of combinational behaviours; in each case, we describe the response, ρ, to a stimulus, σ.

1. A wire, $W(a, b)$ is modelled by

$$\rho a = \rho b = \sigma a \sqcup \sigma b$$

2. A bi-directional switch $\mathsf{S}(g, a, b)$ is modelled by

$$\begin{array}{rcll} \rho g & = & \sigma g & \\ \left.\begin{array}{r}\rho a\\ \rho b\end{array}\right\} & = & \sigma a \sqcup \sigma b & \text{if } 1 \sqsubseteq \sigma g \\ \rho a & = & \sigma a & \text{otherwise} \\ \rho b & = & \sigma b & \end{array}$$

3. A tri-state driver $\mathsf{D}(g, a, b)$ is modelled by

$$\begin{array}{rcll} \rho g & = & \sigma g & \\ \rho a & = & \sigma a & \\ \rho b & = & \sigma a \sqcup \sigma b & \text{if } 1 \sqsubseteq \sigma g \\ \rho b & = & \sigma b & \text{otherwise} \end{array}$$

4. An n-type transistor $\mathsf{N}(g, a, b)$ is modelled by

$$\begin{array}{rcll} \rho g & = & \sigma g & \\ \left.\begin{array}{r}\rho a\\ \rho b\end{array}\right\} & = & \sigma a \sqcup \sigma b & \text{if } 1 \sqsubseteq \sigma g \text{ and } 0 \sqsubseteq \sigma a \sqcup \sigma b \\ \rho a & = & \sigma a & \text{otherwise} \\ \rho b & = & \sigma b & \end{array}$$

A p-transistor is defined dually.
This model only captures some aspects of the behaviour of a MOS transistor. Because behaviours must be monotone, an un-driven gate ($g = \mathsf{Z}$) must switch the transistor off, and a gate in conflict ($g = \mathsf{!}$) must switch it on. A real transistor might behave differently. We must bear this in mind when we apply the model.

5. For a nand gate, **nand**(a, b, c), we give two different models. Both treat a and b as inputs, that is $\rho a = \sigma a$ and $\rho b = \sigma b$. They also both treat c as a function of a and b, so they set

$$\rho c = \sigma c \sqcup f(\sigma a,\ \sigma b)$$

for an appropriate function $\mathcal{B} \times \mathcal{B} \xrightarrow{f} \mathcal{B}$.
The function f should restrict to the usual truth-table on the Boolean values, and be monotone. This determines some values, but leaves open an embarrassment of choices. Table 1 shows the values that are completely determined by these conditions:

f	Z	0	1	!
Z	Z		Z	
0		1	1	
1	Z	1	0	!
!			!	!

table 1

f	Z	0	1	!
Z	Z	Z	Z	Z
0	Z	1	1	1
1	Z	1	0	!
!	Z	1	!	!

table 2

f	Z	0	1	!
Z	Z	1	Z	!
0	1	1	1	!
1	Z	1	0	!
!	!	!	!	!

table 3

Table 3 gives the maximum amount of information compatible with the constraints; Table 2 gives the minimum. There are many other possibilities.

Our model allows many subtle distinctions to be made! Later we will see that for many circuits these distinctions are irrelevant.

We now *define* inputs and outputs semantically.

Definition 4.2. *Let m be a simulation behaviour of sort $I \cup O$.*

– *We say the ports, I* act as inputs *if there is no response on $p \in I$ to any stimulus of ports in I:*

$$\forall\ \sigma.\ (\forall q \in O.\ \sigma q = \mathsf{Z}) \Rightarrow \forall p \in I.\ (m\sigma)p = \sigma p$$

– *Furthermore, we say that a port $q \in O$ acts as a* driven output *if a boolean stimulus of the inputs produces a boolean response on q*

$$\forall\ \sigma. m\sigma = (m(\sigma \downarrow q)) \sqcup \sigma$$

$$\forall\ \sigma.\ (\forall r \in O.\ \sigma r = \mathsf{Z}) \wedge (\forall p \in I.\ (m\sigma)p \in 2) \Rightarrow (m\sigma)q \in 2$$

(where $\sigma \downarrow\ q$ is like the stimulus σ on every port except q, which is left floating: $(\sigma \downarrow q)q = \mathsf{Z}$).

Stable Configurations

We can generalise our definition of a behaviour to use an arbitrary complete lattice $\mathcal{V}$ in place of $\mathcal{B}$. We do this because we will later use the increased generality to model sequential behaviours, by choosing an appropriate lattice, $\mathcal{V}$. We do it *now* because we want to introduce some ideas and results that we will apply now to combinational behaviours, and again, later, to this more-general situation.

Definition 4.3. *Let $\mathcal{V}$ be a complete lattice. A* simulation behaviour *of sort U, with* signals *$\mathcal{V}$, is an inflationary, idempotent, monotone, continuous function $b : \mathcal{V}^U \longrightarrow \mathcal{V}^U$. That is, in addition to the properties already described, we have*

$$b\left(\bigsqcup \sigma_i\right) = \bigsqcup b(\sigma_i)$$

Our earlier definition of a combinational behaviour is simply a special case – in which continuity is trivial, as $\mathcal{B}$ is finite.

Definition 4.4. *The* stable configurations *of a behaviour, $m : \mathcal{V}^U \longrightarrow \mathcal{V}^U$, are its fixed points,*

$$\mathcal{F}_m = \{v | v = m(v)\}$$

Lemma 4.1. *A behaviour is determined by its stable configurations.*

Proof. Any subset, $\mathcal{F} \subseteq \mathcal{V}$, closed under $\bigwedge$ is the fixed-point set of a behaviour defined by

$$b(v) = \bigwedge\{w \geq v | w \in \mathcal{F}\}$$

Taking fixed points generalises the iteration used by a simulator to search for a stable configuration. The fixed points of *any* monotone, inflationary operator, p, on $\mathcal{V}^U$ are closed under $\bigwedge$, and so determine a behaviour, $\overline{p}$, given by

$$\overline{p}(v) = \bigwedge\{w \geq v | w \in \mathcal{F}_p\}$$

For a behaviour, $m = \overline{m}$.

We now describe operations for computing the behaviour of a compound circuit, from the behaviours of its components.

Definition 4.5. *We define operations on behaviours in terms of the corresponding operations on fixed-point sets.*

Assembly: We define an operation $\|$ on $\mathcal{S}_U$ by

$$\mathcal{F}_{m \,\|\, n} = \mathcal{F}_m \cap \mathcal{F}_n$$

Renaming: We describe the action of a renaming function, $U \xrightarrow{r} V$, on a behaviour $\mathcal{V}^U \xrightarrow{m} \mathcal{V}^U$, by describing its fixed points

$$\mathcal{F}_{r^*(m)} = \{v \in \mathcal{V}^V | r \circ v \in \mathcal{F}_m\}$$

Hiding: For a renaming, $U \xrightarrow{r} V$*, and* $\mathcal{V}^V \xrightarrow{m} \mathcal{V}^V$*, we define*

$$\mathcal{F}_{\mathsf{H}_r(m)} = \{r \circ v | v \in \mathcal{F}_m\}$$

We have described the structure of the simulation model declaratively, using operations on fixed-point sets. It is worthwhile expressing these operations operationally, as explicit constructions on behaviours.

Lemma 4.2. *The operations on behaviours have the following operational descriptions:*

Assembly: This corresponds to iterating both behaviours alternately, to find a common fixed point – which is well-defined as behaviours are inflationary and monotone.

Renaming: Suppose $f : U \longrightarrow V$ *is a renaming, and* $\mathcal{V}^U \xrightarrow{m} \mathcal{V}^U$*. We describe* $f^\star(m)$ *by giving the response to a given stimulus* $\sigma \in \mathcal{V}^V$*. We apply* m *to the stimulus* $f \circ \sigma \in \mathcal{V}^U$*, and then define*

$$f^\star(m)\sigma v = \bigsqcup\{m(f \circ \sigma)u | f(u) = v\}$$

Hiding: Suppose $f : U \longrightarrow V$ *in* S_{fin}*. Given* $\mathcal{V}^V \xrightarrow{m} \mathcal{V}^V$*, and* $\sigma \in \mathcal{V}^U$*, we define* $\sigma^\star : V \longrightarrow \mathcal{V}$ *by* $v^\star(v) = \bigsqcup\{\sigma(u) | f(u) = v\}$*. To determine the response to* σ *at* u*, we apply* m *to* $\sigma^\star$*, and use the resulting information about values on ports* $f(u) \in V$*.*

$$\mathsf{H}_f m\sigma u = m\sigma^\star(f(u))$$

or equivalently

$$\mathsf{H}_f m\sigma = f \circ (m\sigma^\star)$$

The stimulus σ *on* U *determines a stimulus* $\sigma^\star$ *on* V*. We observe,* via f*, the response to* $\sigma^\star$ *given by* m*.*

Specification and Correctness

We have seen that several behaviours could be acceptable as **nand** gates. Let us now consider the problem of specifying, for example, a **xor** gate. The functionality is clear – the boolean valuations corresponding to the truth table should be the boolean stable configurations of the device – but there are other issues. The output should respond to a boolean stimulus on the inputs.

Definition 4.6. *A* partially correct *implementation of a boolean function* $F(\mathbf{x})$ *is a circuit,* C*, whose boolean stable configurations correspond to the appropriate truth table – the graph of the function.*

A totally correct *implementation has, in addition, the appropriate input/output behaviour: if the inputs are stimulated with boolean values, and the outputs are left floating – stimulated with* Z *– then a boolean response appears on the outputs and the inputs are unchanged.*

Verifying a CMOS Gate We consider the low-level problem of verifying a CMOS gate in order to make two points. First, that a simple model of transistor behaviour as boolean switches is inadequate. Second, that for certain "design styles" we can show that a switch model is adequate – in the sense that we can prove a meta-theorem showing that total correctness (in the simulation model) follows from appropriate reasoning about the purely boolean switch model.

We start with some simple examples. A recurrent CMOS idiom is the pair of complementary transistors in series whose gates are driven by the same signal:

$$\mathrm{SER}(in, a, b, out) \stackrel{\mathrm{def}}{\equiv} \mathsf{P}(in, a, out) \parallel \mathsf{N}(in, out, b)$$

If in is driven high, the n-transistor is switched on: any configuration with $out = b$ is stable, a 0 on b will appear on out. Dually, if in is driven low, the p-transistor is switched on.

Definition 4.7. *An* inverter *is constructed from a pair of complementary transistors in series:*

$$\mathrm{INV}(in, out) \stackrel{\mathrm{def}}{\equiv} \mathrm{SER}(in, 1, 0, out)$$

Lemma 4.3. *The inverter is a totally correct implementation of a* **not** *gate; in acts as an input; out acts as an output.*

Definition 4.8. *A* transmission gate *is a pair of complementary transistors, in parallel:*

$$\mathrm{TG}(s, \overline{s}, a, b) \stackrel{\mathrm{def}}{\equiv} \mathsf{N}(s, a, b) \parallel \mathsf{P}(\overline{s}, a, b)$$

Lemma 4.4. *A transmission gate can be combined with an inverter to form a switch*

$$\mathsf{S}(g, a, b) \equiv \mathsf{H}\overline{g}.INV(g, \overline{g}) \parallel TG(g, \overline{g}, a, b)$$

Definition 4.9. *We define a circuit* XORIMP *as*

$$\mathrm{XORIMP}(a, b, c) \stackrel{\mathrm{def}}{\equiv} \mathsf{H}\overline{a}.\ \mathrm{INV}(a, \overline{a}) \parallel \mathrm{SER}(b, a, \overline{a}, out) \parallel \mathrm{TG}(\overline{a}, a, b, out)$$

Exercise 4.1. Show that XORIMP is a totally correct implementation of the exclusive or function

The MOS transistor is not a perfect switch. Any model that treats it as one will allow us to validate some incorrect circuits. For example, if our transistors were switches we could validate the circuit obtained by omitting the transmission gate from XORIMP – but this circuit is only partially correct.

The simulation model is just a model. We have already remarked that the assumption of monotonicity is unrealistic. The model is also an oversimplification; it cannot account for the behaviour of many practical CMOS circuits. Nevertheless, it is a reasonable approximation for many purposes.

This simulation model has also been studied by [Zhou and Hoare, 1992], and used by [Bryant, 1984] – who also discusses more-detailed simulation models.

We introduce a simple switch model and relate it to the simulation model. We prove that for circuits that obey certain design rules the switch model is as adequate as our simulation model.

Complementary Static CMOS The simplest CMOS design style is exemplified by the CMOS inverter (Definition 4.7). When the input is 1 the upper transistor is off and the lower one on; the output is a logic 0 propagated to the output. When the input is 0, the situation is reversed, and a logic 1 is propagated to the output.

More complex gates can be constructed in a similar style, for example, the **nand** gate.

$$NAND(a,b,c) \stackrel{\text{def}}{\equiv} \mathsf{H}x.\ \mathsf{N}(a,0,x) \parallel \mathsf{N}(b,x,c) \parallel \mathsf{P}(a,1,c) \parallel \mathsf{P}(b,1,c)$$

The output is 0 if both n-transistors are on $(a \wedge b)$ and 1 in the complementary case $(\neg a \vee \neg b))$.

Generally, a complementary CMOS gate consists of two parts connected at the output: a pull-up network of p-transistors whose gates are driven by the inputs, connected to logic 1 and the output; and a pull-down network of n-transistors whose gates are also driven by the inputs, connected to logic 0 and the output.

The pull-up and pull-down networks follow a particular discipline. We distinguish two types of port: the *input* ports, which are connected to the gates of the transistors; and the others, *nodes* which may be connected by switching a transistor on. In forming a network, we use only one kind of transistor, p (for a pull-up network) or n (for a pull-down network); we only connect ports of the same type, input – input and node – node; and may not hide input ports. One of the non-input ports is connected to 0 (for a pull-down network), or 1 (for a pull-up network). These networks may be represented by a labelled graphs. The edges of the graph correspond to transistors (of the appropriate type). The label on an edge is the port connected to the gate, and the nodes connected by the edge are the ports that are connected when the transistor is on.

We now discuss the behaviour of a network (then we will go on to connect two networks to form a gate). The input ports act as inputs. A given boolean stimulation of these inputs connects some nodes of the graph and leaves others unconnected – it determines a subgraph that contains all the nodes of the original, but only those edges that are switched on. The response to a boolean stimulus on the gates is determined by the connected components of the corresponding subgraph.

Lemma 4.5. *If the gates of a pull-down network are driven by boolean signals then a node is driven to* 0 *iff there is no boolean stable configuration in which it takes the value* 1*, and dually for pull-up networks.*

Proof. The boolean stable configurations of the network must assign the same value to all nodes in a connected component. Let z be the node connected to 0. If the other nodes are left floating (*ie.* stimulated with Z), the response will propagate 0 to all nodes in the connected component of z. Furthermore, the configuration in which nodes connected to z take the value 0, and all nodes not connected to z take the value 1 is stable.

To a network C of sort U we associate a boolean function, $[\![C]\!] \in 2^{2^U}$ which characterises a set $\mathcal{S}_C = \mathcal{F}_C \cap 2^U$ – the stable boolean configurations, $\mathbf{x} \in 2^U$, of C. We represent this boolean function, in turn, by a boolean expression. For example, the boolean stable configurations of an n-type transistor $\mathsf{N}(g, a, b)$ are the boolean valuations that make the expression $g \rightarrow a = b$ true.

Definition 4.10.

$$\begin{aligned} [\![\mathsf{N}(g,a,b)]\!] &= g \rightarrow a = b \\ [\![\mathsf{P}(g,a,b)]\!] &= \neg g \rightarrow a = b \\ [\![A \parallel B]\!] &= [\![A]\!] \wedge [\![B]\!] \\ [\![\mathsf{H}x.\ A]\!] &= \exists x.\ [\![A]\!] \end{aligned}$$

Lemma 4.6. *The boolean stable configurations* $\mathbf{x} \in 2^U$ *of a pull-up or pull-down network* C *are given by*

$$\mathcal{S}_C = \{\mathbf{x} |\ [\![C]\!](\mathbf{x})\}$$

Proof. By induction on the structure of the network. Only hiding is not obvious. The point is that if there is some simulation value on a hidden port that permits some boolean stable configuration on the visible ports, there is some boolean value on the hidden port compatible with the visible configuration. Consider, for example, a pull-down network. By Lemma 4.5, either the value on the hidden port is 0, or there is a configuration with the same visible values which gives this port the value 1.

Example 4.2. 1. The stable configurations of the network

$$\mathsf{H}v.\ \mathsf{N}(a, v, 0) \parallel \mathsf{N}(b, v, out) \parallel \mathsf{N}(c, v, out))$$

are given by

$$\exists v \begin{pmatrix} a \rightarrow & v = 0 \\ \wedge\ b \rightarrow & v = out \\ \wedge\ c \rightarrow & v = out \end{pmatrix}$$

which is equivalent to

$$a \wedge (b \vee c) \ \rightarrow\ out = 0$$

2. The network

$$\mathsf{H}v, w.\ \mathsf{N}(a, v, 0) \parallel \mathsf{N}(b, w, 0) \parallel \mathsf{N}(c, v, w) \parallel \mathsf{N}(d, v, out) \parallel \mathsf{N}(e, w, out)$$

has the following boolean behaviour

$$\exists v,\ w \left(\begin{array}{rl} a \rightarrow & v = 0 \\ \wedge\ b \rightarrow & w = 0 \\ \wedge\ c \rightarrow & v = w \\ \wedge\ d \rightarrow & v = out \\ \wedge\ e \rightarrow & w = out \end{array} \right)$$

which is equivalent to

$$\begin{array}{c} (c \wedge (a \vee b) \wedge (d \vee e)) \\ \vee \\ (a \wedge d) \vee (b \wedge e) \end{array} \rightarrow\ out = 0$$

Notice that, in this example, the transistor whose gate is driven by c is used bidirectionally – in one direction when $a \wedge c \wedge e$, and in the other when $b \wedge c \wedge d$.

The Behaviour of a Gate. For the general complementary gate, we have a pair of boolean relations, $P(\mathbf{x}, out)$ and $N(\mathbf{x}, out)$, representing the behaviours of the pull-up and pull-down networks, respectively. $P(\mathbf{x}, out)$ is true if the p-network is compatible with the configuration $(\mathbf{x}, out)$. Notice that $P(\mathbf{x}, 1)$ is always true, while $P(\mathbf{x}, 0)$ is true if the output is *not* driven high. Dually, $N(\mathbf{x}, 0)$ is always true, while $N(\mathbf{x}, 1)$ is true if the output is *not* driven low.

When we connect the outputs of a pull-up and pull-down network to form a gate, we should check that we don't connect the output (or any internal nodes) to 0 and 1 at the same time – otherwise we go outside the scope of the boolean model. We also want to check that the output is driven – that the circuit will produce a boolean output.

Lemma 4.7. *There is* no conflict *on the output of the gate (ie boolean inputs do not produce a* ! *output) iff*

$$\forall \mathbf{x}. \exists out.\ P(\mathbf{x}, out) \wedge N(\mathbf{x}, out)$$

The output is driven*, for boolean inputs, iff*

$$\forall \mathbf{x}. \neg P(\mathbf{x}, 0) \vee \neg N(\mathbf{x}, 1)$$

The conjunction of the pull-up and pull-down relations captures the boolean stable configurations of the gate.

Definition 4.11. *The* boolean behaviour *of a gate whose pull-up and pull-down networks are characterised by $P(\mathbf{x}, out)$ and $N(\mathbf{x}, out)$ is given by*

$$S(\mathbf{x}, out) \stackrel{\text{def}}{\equiv} P(\mathbf{x}, out) \wedge N(\mathbf{x}, out)$$

. If there is no conflict and the outputs are driven, then this boolean behaviour determines the response of the gate to boolean inputs: S is a functional relation, and the gate is a totally correct implementation of this function.

Theorem 4.1. *A complementary gate with boolean behaviour S, is a totally correct implementation of a boolean function F iff*

$$S(\mathbf{x}, out) \;\equiv\; out = F(\mathbf{x})$$

Proof. If S is functional then the gate has no conflict and is driven.

Remark 4.1. Frequently we don't want to specify a gate as implementing a given boolean function; we specify the output that should be produced for some input patterns, but don't care what output is produced for other inputs.

Such a specification may be represented as a boolean relation, $R(\mathbf{x}, out)$, such that $\forall \mathbf{x}. \exists out. R(\mathbf{x}, out)$. It is then natural, but insufficient, to attempt to verify an implementation by showing that

$$S(\mathbf{x}, out) \quad \Rightarrow \quad R(\mathbf{x}, out) \tag{4.1}$$

If we adopt this style of specification and verification we must also check separately that there are no conflicts, and that the output is driven. Neither condition follows automatically from the implication, 4.1. (Note that we have defined a gate to have driven outputs, other circuits, such as a tri-state driver, are useful in practice. The switch model is not adequate for reasoning about these.)

5. Relational Semantics

The simulation model accounts for some subtleties of behaviour, but is unnecessarily detailed for some purposes – and not nearly detailed enough for some others! It is natural to consider the set of boolean stable configurations as a simpler model of behaviour. This *relational semantics* (along with its extension to sequential behaviours, which we will come to later) was introduced by Hanna [Hanna and Daeche, 1983], and popularised by Gordon [Gordon, 1986] and Hunt [Hunt, 1989].

Definition 5.1. *In the* relational model*, a circuit C of sort U is modelled by a subset $[\![C]\!] \subseteq \mathcal{V}^U$ – for some suitable choice of $\mathcal{V}$ to represent signals on wires. For a boolean combinational circuit $\mathcal{V} = 2$.*

Assembly. For circuits of sort U we define

$$\begin{aligned} [\![A \parallel B]\!] &= [\![A]\!] \cap [\![B]\!] \\ [\![\top]\!] &= \mathcal{V}^U \end{aligned}$$

Renaming. Given a renaming function, $U \xrightarrow{r} V$*, and a behaviour* $[\![A]\!] \subseteq \mathcal{V}^U$*, we define*

$$r^\star([\![A]\!]) \quad = \quad \{\mathbf{v} \in \mathcal{V}^V \mid r \circ \mathbf{v} \in [\![A]\!]\}$$

Hiding. For a renaming, $U \xrightarrow{r} V$*, and* $[\![B]\!] \subseteq \mathcal{V}^V$*, we define*

$$\mathsf{H}_r([\![B]\!]) = \{r \circ \mathbf{v} \mid \mathbf{v} \in [\![B]\!]\}$$

This definition simply extends and re-presents the relational treatment we gave earlier for transistor networks. Syntactically, we use $\wedge$ to interpret composition, and $\exists$ to represent hiding.

As we have seen, the application of the relational model requires some delicacy: in general its treatment of hiding is unsound (with respect to the simulation model), and it will not even provide partial correctness arguments. However, for common design styles we can give general arguments to show that it is simply related to the simulation model, and that total correctness follows from partial correctness.

Verifying Combinational Circuits

We have shown how to use the relational model to verify complementary CMOS implementations of combinational gates. We now extend our use of the model to cover circuits composed from combinational gates.

In general, $[\![C]\!] \subset \mathcal{F}_C \cap 2^U$, but these sets may not be equal – because hiding in the relational model does not correspond to hiding in the simulation model. However, as before, we can impose restrictions on the circuits we consider to ensure that these operations do correspond. We have to ensure that if there is some simulation value on a hidden port that permits a boolean stable configuration on the visible ports, there is some boolean value on the hidden port compatible with this visible configuration.

Definition 5.2. *A* combinational gate*, G, is a device with two types of port: inputs, $I(G)$, which act as inputs; and outputs, $O(G)$, which are driven with a boolean response to boolean stimuli on the inputs.*

The boolean stable states of a combinational gate characterise its response to boolean input stimuli.

Lemma 5.1. – *Composing two combinational gates, A, B, such that*

$$O(B) \cap I(A) = \emptyset$$

by connecting some inputs of B to outputs of A, produces a combinational gate $A \parallel B$, with

$$
\begin{aligned}
O(A \parallel B) &= O(A) \cup O(B) \\
I(A \parallel B) &= I(A) \cup I(B) \setminus O(A)
\end{aligned}
$$

- *Connecting two inputs (ie. renaming along a surjection that is 1-1 on outputs) of a combinational gate gives a combinational gate.*
- *Hiding an output of a combinational gate gives a combinational gate.*

Furthermore, for circuits composed using these operations, from primitive combinational gates, the relational model gives the set of boolean stable configurations.

We say that circuits built using the operations of this Lemma obey the following *design rules:*

- Don't connect outputs together.
- Don't hide floating inputs.
- Don't create feedback.

Specification and Correctness In specifying and designing circuits, we often only want to specify the output that should be produced for some input patterns, but don't care what output is produced for other inputs. Such a *lax* specification is represented as a boolean relation, $\mathrm{SPEC}(\mathbf{x}, \mathbf{y})$, such that $\forall \mathbf{x}. \exists \mathbf{y}. \mathrm{SPEC}(\mathbf{x}, \mathbf{y})$ – where $\mathbf{x}$ represents the inputs, and $\mathbf{y}$ the outputs.

Sometimes, as with tri-state drivers, we don't even want to insist on any boolean response for some inputs. We introduce an input constraint $C(\mathbf{x})$ to specify those inputs for which a boolean response is required, and consider lax specifications $\mathrm{SPEC}(\mathbf{x}, \mathbf{y})$, such that

$$\forall \mathbf{x}. C(\mathbf{x}) \Rightarrow \exists \mathbf{y}. \mathrm{SPEC}(\mathbf{x}, \mathbf{y})$$

Definition 5.3. *A circuit satisfies the specification,* $\mathrm{SPEC}(\mathbf{x}, \mathbf{y})$ *under input constraints* $C(\mathbf{x})$ *iff stimulating the inputs with boolean values,* $\mathbf{x}$*, satisfying the constraints,* $C(\mathbf{x})$*, produces a boolean response,* $\mathbf{y}$*, on the outputs, such that* $\mathrm{SPEC}(\mathbf{x}, \mathbf{y})$.

This generalises Definition 4.6: a circuit is a totally correct implementation of a boolean function F iff it satisfies the specification $out = F(\mathbf{x})$, with no input constraints.

Theorem 5.1. *A circuit built from combinational gates, following the design rules, satisfies the specification,* $\mathrm{SPEC}(\mathbf{x}, \mathbf{y})$ *under constraints* $C(\mathbf{x})$ *iff*

$$C(\mathbf{x}), \mathrm{IMP}(\mathbf{x}, \mathbf{y}) \vdash \mathrm{SPEC}(\mathbf{x}, \mathbf{y})$$

where $\mathrm{IMP}(\mathbf{x}, \mathbf{y})$ *is the relational model of the circuit's behaviour.*

We can compute a lax approximation, $\mathrm{LAX}(\mathbf{x},\mathbf{y})$, to a circuit's behaviour by using lax specifications for the behaviours of its primitive components in place of their actual behaviours. If the component specifications have no input constraints this is straightforward: we simply interpret assembly and hiding by $\wedge$ and $\exists$. Then,

$$\mathrm{IMP}(\mathbf{x},\mathbf{y}) \vdash \mathrm{LAX}(\mathbf{x},\mathbf{y})$$

– because the operations $\wedge$ and $\exists$ are monotone.

Corollary 5.1. *A circuit built from combinational gates, following the design rules, satisfies the specification,* $\mathrm{SPEC}(\mathbf{x},\mathbf{y})$ *under constraints* $C(\mathbf{x})$ *provided*

$$C(\mathbf{x}), \mathrm{LAX}(\mathbf{x},\mathbf{y}) \vdash \mathrm{SPEC}(\mathbf{x},\mathbf{y})$$

If the component specifications carry input constraints, we must propagate these to derive a specification and constraints for the compound circuit. When we assemble components we accumulate constraints conjunctively, as proof obligations. This may result in constraints on internal nodes. These must be refined to input constraints,by applying the following lemma.

Lemma 5.2. *We can compute a lax specification and constraints for a circuit constructed according to the rules of Lemma 5.1 as follows:*

– *If circuit A satisfies the specification,* $\mathrm{SPEC}_A(\mathbf{x},\mathbf{y})$ *under input constraints* $C_A(\mathbf{x})$*, and circuit B satisfies the specification,* $\mathrm{SPEC}_B(\mathbf{x},\mathbf{y})$ *under input constraints* $C_B(\mathbf{x})$*, and*

$$C(\mathbf{v}) \wedge \mathrm{SPEC}_A(\mathbf{x},\mathbf{y}) \wedge \mathrm{SPEC}_B(\mathbf{x},\mathbf{y}) \Rightarrow C_A(\mathbf{x}) \wedge C_B(\mathbf{x})$$

(where $A \parallel B$ has inputs $\mathbf{v}$),
then $A \parallel B$ satisfies the specification $\mathrm{SPEC}_A(\mathbf{x},\mathbf{y}) \wedge \mathrm{SPEC}_B(\mathbf{x},\mathbf{y})$ *under constraints* $C(\mathbf{v})$*.*

– *If circuit A satisfies the specification,* $\mathrm{SPEC}_A(\mathbf{x},\mathbf{y})$ *under input constraints* $C_A(\mathbf{x})$*, and $y \in \mathbf{y}$ then $\mathsf{H}y.\ A$ satisfies the specification* $\exists y.\ \mathrm{SPEC}_A(\mathbf{x},\mathbf{y})$ *under input constraints* $C_A(\mathbf{x})$*.*

Ports appearing free in outstanding proof obligations may not be hidden – this corresponds precisely to the free variable restrictions on the standard rule for left introduction of the existential quantifier.

Remark 5.1. All this certainly *seems* simple enough – but it is simple largely because our design rules exclude a number of difficulties. Here are two examples of problems that occur if we connect outputs:

1. If we consider the following circuit given by composing an **and** and an **or** gate

$$c = \mathbf{and}(at, bt) \parallel c = \mathbf{or}(at, bt)$$

for the input values $a = 1$, $b = 0$, we obtain $c = 1$ & $c = 0$, which is nonsense and reveals the inapplicability of the relational model – rather than the impossibility of building the circuit. What this circuit actually *does* will depend on the technology used to implement the gates.

2. A more subtle misuse of the model is to connect the outputs of two devices modelled nondeterministically – by lax specifications, and to use the model naïvely to deduce that they must influence each other to produce a mutually satisfactory choice of output. Again, this would be unsound.

Busses and Tri-state Drivers. We have also excluded some difficulties by restricting our attention to combinational gates – which always produce a boolean output when stimulated with boolean inputs. If we introduce components with conditional specifications we must introduce corresponding proof obligations, to ensure that the input conditions are met. Lemma 5.2 tells us how to reason about these conditions within the relational model – see Example 5.2 below.

Definition 5.4. *A bus with two drivers given by*

$$\mathrm{BUS}(c1, a1, c2, a2, out) \stackrel{\mathrm{def}}{\equiv} \mathsf{D}(c1, a1, out) \parallel \mathsf{D}(c2, a2, out)$$

A tri-state driver, $\mathsf{D}(g, a, b)$, is not a combinational device; nor is this bus circuit, formed by connecting the outputs of two tri-state drivers.

Example 5.1. The bus with two drivers, $\mathrm{BUS}(a1, c1, a2, c2, out)$, satisfies the specification, $S(c1, a1, c2, a2, out) \stackrel{\mathrm{def}}{\equiv}$

$$\begin{aligned} c1 &\Rightarrow out = a1 \\ c2 &\Rightarrow out = a2 \end{aligned}$$

under the constraint

$$C(c1, a1, c2, a2) \stackrel{\mathrm{def}}{\equiv} c1 \neq c2$$

(or the weaker constraint $c1 \wedge c2 \Rightarrow a1 = a2$).

Note that this circuit does *not* satisfy the specification

$$C(c1, a1, c2, a2) \Rightarrow S(c1, a1, c2, a2, out)$$

because the output is not driven to a boolean value for all boolean inputs.

Example 5.2. We may implement a multiplexor by

$$\mathrm{MUX}(c, a, b, out) \stackrel{\mathrm{def}}{\equiv} \mathsf{H}\overline{c}.\ \mathrm{INV}(c, \overline{c}) \parallel \mathrm{BUS}(c, a, \overline{c}, b, out)$$

This circuit is a combinational gate; the inverter ensures that the constraint on the bus inputs is satisfied.

Bus-structured designs do not obey our first design rule. To treat them we model the bus together with tri-state drivers as a single component, that *when it is driven by non-conflicting drivers* places the appropriate value on the bus. The constraint that exactly one driver is active is introduced as a proof obligation. At the concrete level, we only need the weaker constraint that whenever two drivers are active they must carry the same value. However, this weaker constraint does not allow data refinement, where one abstract value may have a number of different concrete representations[1], so we adopt the stronger version.

6. Sequential Logic

Sequential devices may be implemented as combinational devices with feedback through a delay. We use this representation to derive a simulation model of sequential behaviour from our treatment of combinational behaviour.

The combinational component of a sequential device of sort U, with state bits S is modelled as a combinational behaviour of sort $S + U + S$. The two copies of S represent, respectively, the current state, which acts as an input, and the next state, which acts as an output. Since we need not represent the response on the current-state, or the stimulus on the next state, we can simplify this description.

Definition 6.1. *A* sequential behaviour *of sort U, with a set S of state bits, is an idempotent, monotone function*

$$m : \mathcal{B}^S \times \mathcal{B}^U \xrightarrow{m} \mathcal{B}^U \times \mathcal{B}^S$$

which is inflationary on U: if $(\rho, s') = m(s, \sigma)$ then, for $u \in U$ we have $\sigma(u) \sqsubseteq \rho(u)$.

We simulate a sequential circuit by computing the stable configuration reached after each successive clock tick. The next-state at one clock tick is fed back to form the current state at the next clock tick. We consider a *sequence,* $\sigma \in \left(\mathcal{B}^U\right)^{\mathsf{N}}$ of values as a stimulus, and compute the response – another sequence of values.

Definition 6.2. *Let m be a sequential behaviour of sort U, with state bits S. Given a stimulus sequence $\sigma \in \left(\mathcal{B}^U\right)^{\mathsf{N}}$, and an initial state $s_0 \in \mathcal{B}^S$, we define a sequence, s_m, of states $(s_m)_i$, and a response sequence, $(m^\star\sigma) \in \left(\mathcal{B}^U\right)^{\mathsf{N}}$, by*

$$\begin{aligned} (s_m)_0 &= s_0 \\ ((m^\star\sigma)(i), (s_m)_{i+1}) &= m((s_m)_i, \sigma(i)) \end{aligned}$$

[1] I am grateful to Simon Finn for this, typically astute, observation.

Thus, computing the response to a given sequence of stimuli, from a given initial state, s_0, gives a map $\left(\mathcal{B}^U\right)^{\mathsf{N}} \xrightarrow{m^\star} \left(\mathcal{B}^U\right)^{\mathsf{N}}$*, or equivalently a map*

$$\left(\mathcal{B}^{\mathsf{N}}\right)^U \xrightarrow{\overline{m}_{s_0}} \left(\mathcal{B}^{\mathsf{N}}\right)^U$$

We can consider $\mathcal{B}^{\mathsf{N}}$, with the point-wise ordering, as a lattice of values in its own right. A sequential behaviour can be represented as a simulation behaviour, using this lattice of values.

Definition 6.3. *A simulation behaviour,* $b : \left(\mathcal{B}^{\mathsf{N}}\right)^U \longrightarrow \left(\mathcal{B}^{\mathsf{N}}\right)^U$*, with signals in* $\mathcal{B}^{\mathsf{N}}$ *(see Definition 4.3), is* causal *if*

$$(b\sigma) \downarrow n = (b(\sigma \downarrow n) \downarrow n)$$

Where $(\sigma \downarrow n)pm = \mathsf{Z}$ *for* $m \geq n$*, and* $((\sigma \downarrow n)p)m = (\sigma p)m$ *for* $m < n$*.*

Definition 6.4. *A simulation behaviour is* well-behaved *if it is causal, with inputs, and outputs which are driven with boolean signals in response to boolean signals on the inputs. We say a circuit is* well-behaved *iff its behaviour is.*

Lemma 6.1. *Let m be a sequential behaviour of sort U, with state bits S, then*

$$\left(\mathcal{B}^{\mathsf{N}}\right)^U \xrightarrow{\overline{m}_{s_0}} \left(\mathcal{B}^{\mathsf{N}}\right)^U$$

is a causal simulation behaviour with signals in $\mathcal{B}^{\mathsf{N}}$

If, in addition, the sequential behaviour gives boolean outputs and next state in response to boolean inputs and current state, and the initial state is boolean, then $\overline{m}_{s_0}$ *is well-behaved.*

For combinational behaviours, we saw that adherence to simple, natural, and un-demanding design rules allows us to use the relational model to reason about the correctness of circuits.

For sequential behaviours the situation is similar, but more complex. The lattice $\mathcal{B}^{\mathsf{N}}$ is far richer than the finite lattice $\mathcal{B}$, and the scope for subtle behavioural distinctions is correspondingly rich. For example, we might relax the requirement of total correctness, to ask only that outputs be driven after some initialisation period. We restrict ourselves to the most straightforward generalisation of the relational model. We consider signals represented as functions from natural numbers to some set, V, of values. For example, boolean signals $\mathbf{v} \in 2^{\mathsf{N}} \subseteq \mathcal{B}^{\mathsf{N}}$.

As before, we want to relate a relational model, representing the behaviour of a component of sort U as a predicate on valuations, $\mathbf{v} \in \left(\mathsf{V}^{\mathsf{N}}\right)^U$, to the simulation model based on the lattice $\mathcal{V}^{\mathsf{N}}$, where $\mathcal{V}$ is the lattice obtained by adjoining bottom and top elements ! and Z to the set V. We consider circuits with distinguished inputs, I, and outputs, O. The definition of satisfaction generalises straightforwardly to this new setting:

Definition 6.5. *A circuit, with inputs I and outputs O,* satisfies the specification, SPEC($\mathbf{x}, \mathbf{y}$) under input constraints $C(\mathbf{x})$ *iff stimulating the inputs with values,* $\mathbf{x} \in \left(\mathsf{V}^{\mathsf{N}}\right)^{I}$, *satisfying the constraints,* $C(\mathbf{x})$, *produces a boolean response,* $\mathbf{y} \in \left(\mathsf{V}^{\mathsf{N}}\right)^{O}$, *on the outputs, such that* SPEC($\mathbf{x}, \mathbf{y}$).

Definition 6.6. *There is a* non-zero delay *between an input, p, and an output, q, of a well-behaved sequential circuit with behaviour m if a change on p has no immediate response on q:*

$$\forall \sigma, \sigma', n.\ (\forall r, t.\ r \neq p \vee t < n \Rightarrow \sigma r t = \sigma' r t) \Rightarrow \sigma q n = \sigma' q n$$

Lemma 6.2 (Feedback with Delay). *Well-behaved, sequential circuits are closed under the operations of Lemma 5.1. Furthermore, if there is non-zero delay between an input p and an output q of a well-behaved circuit, then connecting p and q gives a well-behaved circuit.*

Proof. By induction on time.

So, we relax the design rule forbidding feedback to, 'do not create zero-delay feedback loops,' (otherwise we could, for example, connect the two ports of an inverter to get $x\ t = not\ (x\ t)$ in the model). Our other design rules stand.

Theorem 6.1. *We can compute a lax specification and constraints for a sequential circuit built from well-behaved circuits using the operations of Lemma 6.2, just as we did for combinational circuits in Lemma 5.2.*

7. Using the Relational Model

Design rules allow the designer to rely on abstract reasoning, using the relational model. To take advantage of behaviour at a lower level of abstraction, a designer will sometimes break the rules, on purpose. For example, consideration of driving ratios may show that connecting outputs has the effect of a 'wired and'; or detailed analysis of gate delays may show that we can make a flip-flop with cross-coupled nand gates. If we obey the rules then the relational model will not lead us astray; those want to break the rules have to pay a price – they must either produce more-detailed models, or use other methods such as circuit-level simulation[2].

[2] Of course, simulation also depends on models. However, it is much easier to introduce *ad hoc* adjustments to an operational simulation model than to a compositional formal model. Some may consider this an advantage.

We briefly review the relational model. A circuit's behaviour is modelled by a relation expressing how the signals on its ports *must* be related if it is operating correctly and there are well-formed signals on the ports. State is implicit in this representation; for example, a buffer may be modelled by the predicate

$$\Delta(in, out) \stackrel{\text{def}}{\equiv} \forall t. out(t+1) = in\ t$$

We often model devices by lax specifications; the intended interpretation of these is restrictive – the device only allows combinations of signals satisfying the predicate, but not prescriptive – the predicate may allow combinations of signals that are not exhibited in operation. For example, the predicate $\Delta(in, out)$ could be used to *model* a device whose output at time zero is indeterminate, or one which will always output some particular value at time 0. We will also allow input constraints; we only require the device to behave relationally when these constraints are satisfied.

This formalism is easily (maybe too easily) applied to clocked synchronous digital designs;

- time is abstracted as a discrete sequence of instants (represented by natural numbers),
- the clock is implicit, and
- time 0 corresponds to an implicit global reset.

There is an implicit presumption that each wire carries a sequence of well-defined values; if the hardware allows other voltage or current values, that do not correspond to abstract logic values, the model may lead us astray. Our design rules ensure that this presumption is justified.

Composition

The model is *compositional*; the behaviour of a circuit may be composed from the behaviours of its sub-circuits. The composition of components is represented by the *conjunction* of the corresponding predicates, using shared variables to express connections. Internal wires may be hidden using existential quantification. For example, a two-place buffer may be composed from two buffers; the behaviour is composed as follows:

$$\Delta_2(a, c) \stackrel{\text{def}}{\equiv} \exists b. \Delta(a, b) \wedge \Delta(b, c)$$

We may then *prove* that

$$\Delta_2(a, c) \equiv \forall t. c(t+2) = a\ t$$

Abstraction

We can take more-abstract interpretations of the same formalism. For example, we can model the boolean signals carried on a bundle of wires, abstractly, as a single sequence of abstract values given by interpreting the bit-pattern on the bus at each time. This is a simple example of data-abstraction. We can also take a more abstract view of time, for example by considering the values on wires only at every n^{th} clock tick. Temporal- and data-abstraction were first formalised by [Melham, 1988]. More examples are discussed in [Fourman and Hexsel, 1991].

We must represent the abstraction of behaviours as well as their composition. For example: it is essential to be able to view an adder circuit both at the boolean level, where it implements some function on boolean vectors, and at the abstract level where it implements an arithmetic operation on (a subrange of) the integers; or to view a microprocessor at a level that abstracts away microcode and pipelining, as well as at the detailed register-transfer level.

In general, descriptions at different levels are connected by representation relations $\mathbf{rep}(conc, abs)$ linking concrete signals to corresponding abstract signals. Typically, these relations arise from abstraction functions

$$\mathbf{rep}(conc, abs) \equiv abs = \mathbf{abs}(conc)$$

If the abstraction function **abs** is not total, then some concrete signals will not correspond to any abstract value, and we must either at the abstract level allow the possibility that some signals are not well-defined, or at the concrete level generate proof obligations to show that the concrete values occurring do always have abstract counterparts. The situation is analogous to the subtleties we have already considered in relating the simulation model – which allows values that do not correspond to either of the 'abstract' boolean values – to the relational model.

Once we are in the digital domain, we can *choose* to use only total abstraction functions. However, in passing from circuit-level behaviour to a digital abstraction, we *must* use a partial abstraction function if our abstract view is to be stable under small circuit-level perturbations. To successfully isolate digital design from circuit-level considerations, we must be sure that the physical signals in our circuits can be described at the abstract level. For simplicity, we presume that if the inputs to a device are well-defined at all times, then so are the outputs. We must check that this property is preserved when we compose devices to build compound circuits. This is the rôle of our design rules.

Correctness

A basic goal for a designer producing a register-transfer level (RTL) description is logical correctness. The logical correctness of an RTL description with respect to a behavioural specification can be represented directly as a theorem-proving goal [Hanna and Daeche, 1983], and computer-assisted formal reasoning tools can be used to verify non-trivial circuits [Cohn, 1988] [Cohn, 1989] [Hunt, 1987], [Busch, 1991]

A behaviour expresses the combinations of signals allowable. A specification should express the acceptable behaviours. Specifications are thus higher-order objects – sets of behaviours. Methodologically, it is more useful to express specifications at the same level as behaviours. We specify constraints which must be imposed, and allow any implementation which imposes at least these constraints. To verify an implementation against a specification, we have to prove a theorem of the following form:

$$\texttt{circuit}, \texttt{environment} \vdash \texttt{specification} \tag{7.1}$$

This picks out a set of acceptable behaviours, but not all possible sets of behaviours can be expressed in this way. The interpretation of this theorem is that if a circuit with behaviour `circuit`, is placed in an external environment satisfying conditions `environment`, then it will impose the constraints given by the specification, `specification`.

Because of the transitivity of $\vdash$, and the monotonicity of $\wedge$ and $\exists$, this formulation allows us to verify designs hierarchically, basing a design on specified behaviours. On the basis of the specifications of our subcomponents, we deduce that the design will impose certain constraints. If the components we use impose more constraints, then our implementation will still impose the required constraints (and, typically, will impose other, incidental, constraints).

Using this approach, we might be tempted to use a formally inconsistent behaviour as a universal design; it entails *every* specification. Our design rules prevent this. Composing causal devices such that

$$\forall \mathbf{x}.\ \exists \mathbf{y}.\ \mathrm{DEV}(\mathbf{x}, \mathbf{y})$$

according to the design rules, will not generate formally inconsistent behaviours. Note that, in general, proving theorems of the form

$$\texttt{environment} \vdash \texttt{circuit} \equiv \texttt{specification}$$

provides no more information than 7.1 about the hardware; the intended interpretation of the model is not that the circuit will exhibit all behaviours allowed by the predicate `circuit`, only that it will not exhibit any others.

8. Proof and Design

The relational model forms the basis for the application of theorem-proving methods to the verification of hardware. However, *post-hoc* verification is unnecessarily hard, and of limited value. Correctness should be considered as part of the design process – not as an after-thought. We show how tools for incremental behavioural synthesis, developing an RTL description from an abstract behavioural specification, may be based on Milner's idea of tactical problem solving. The logical correctness of the design is proved as side-effect of our approach.

Goal-directed problem-solving

The goal-directed approach to problem solving involves successively reducing a goal to a collection of subgoals (whose achievement would allow us to achieve the goal). This process stops when only trivial goals remain. Explicit machine assistance for goal-directed problem solving was implemented elegantly in LCF, using Milner's formalisation of tactics [Milner, 1983] [Gordon *et al.*, 1979]. This idea has been used in other tactical proof assistants, including HOL [Gordon, 1985]. We apply it to the formalisation of digital design.

Tactics Problem-solving generates a tree whose nodes are goals: the leaves are the outstanding subgoals, the root is the initial goal. Refining an outstanding subgoal spawns subgoals, and extends the tree. A *tactic* is a method for refining a goal, it produces the subgoals and also a validation. If a tactic applied to a goal A produces a list B_i of subgoals, the *validation* gives a way of transforming achievements of the B_i into an achievement of A. We construct the tree using tactics, which also annotate it with validations. When all the offspring of a goal have been achieved, the validation attached to it is applied to achieve that goal.

If our goal is to prove a theorem, an achievement is a proof [3] – demonstrating the theorem which is the last line of the proof. Primitive valid tactics correspond to the primitive inference rules of the logic. An inference rule may be viewed as a function from proofs to proofs; this function is the validation of the tactic. For example, the rule $\frac{A \quad B}{A \wedge B}$ provides a function which given a proof of A and a proof of B yields a proof of $A \wedge B$. The corresponding tactic, CONJ_TAC takes a goal which is a conjunction (it fails on all other goals) and produces its conjuncts as subgoals, together with the validation provided by the inference rule. Other tactics correspond to the other inference rules.

This is all very well, but it would be excruciating to use only these tactics. Often we want to produce new tactics. This raises a problem; how can we ensure that our tactics are correct? The user should only be allowed to apply validations that produce theorems – not other objects, such as formulae, some of which look so like theorems that in a moment of misguided inspiration he may mistake one for the other. This is accomplished by the type discipline of the implementation language, SML [Milner *et al.*, 1990]. Theorems form an

[3] Conversely, with current technology for computer-assisted formal reasoning, a non-trivial proof is often an achievement!

abstract type, elements of which may only be constructed using the functions corresponding to primitive inference rules. New tactics may be written (a simple example would be to repeatedly apply CONJ_TAC to all outstanding goals until all top-level conjunctions have disappeared) but their validations can only produce theorems by applying the primitive validation functions. The type discipline ensures that, whatever tricks we may use in coding tactics, we can never produce spurious theorems. If our validations succeed they will produce theorems – although it is possible that these will not be the theorems we initially sought.

In Milner's formulation, the link between the outstanding subgoals and the initial goal is *implicit* in the validation functions that decorate the tree. Until we have applied the validations we cannot be certain they will not fail – nor can we see, by inspection of the tree, which theorems they will produce should they succeed. For us this is a significant defect as we will see when we introduce our representation of the design process.

Schematic proof Although LCF and HOL neatly formalise goal-directed problem solving, we prefer to represent the partial proof tree (which will correspond to an intermediate design state) syntactically as an explicit data object, rather than operationally *via* validation functions. We follow an approach introduced by Paulson who, in his prover ISABELLE [Paulson, 1987], made the link between the goal and the outstanding subgoals fundamental. The goal tree is replaced by a valid derived rule of the logic. The premises of this rule correspond to subgoals, its conclusion to the initial goal.

The integrity of the system is again based on the type abstraction of SML: A rule is a member of a safe type of the system, meaning that all rules produced are *sound.* A rule consists of the following parts :

$$\frac{\begin{array}{c}\texttt{Premise}_n\\ \vdots\\ \texttt{Premise}_1\end{array}}{\texttt{Conclusion}}$$

Each of the *premises* and the *conclusion* is a *sequent* of the form :

$$\texttt{hypotheses} \vdash \texttt{assertion}$$

For a sequent to be *valid,* the *assertion* must be true in any interpretation that makes all the *hypotheses* true. For a rule to be *sound,* the conclusion must be valid whenever all of the premises are.

Starting from the primitive inference rules, new rules may be constructed by *resolution* of two existing rules A and B; first instantiate A and B by unifying the conclusion of rule A with a premise of rule B, then collapse the resulting proof tree to form a new derived rule.

Example A problem arises in goal-directed theorem proving when we want to achieve an existential goal by exhibiting a witness. When we strip the existential quantifier, we may not be ready to choose the witness. This leaves us with a goal to prove, Qx *for some choice of* x. An appropriate choice may only be apparent after reducing this goal to several subgoals involving x. The HOL/LCF approach allows us to defer the decision by using validations that prove an instantiation of the goal, but provides no support to ensure the compatibility of the instantiations made on different branches of the proof tree.

In Paulson's representation, rules are schematic; any substitution instance of a valid rule is valid. This allows us to develop the proof of the subgoal Qx schematically, with a variable in place of the witness, and to defer the choice of witness to an appropriate point in the proof. When we do instantiate x, unification automatically propagates the instantiation to the other premises – which correspond to different branches of the proof tree.

This delayed binding is crucial in our representation of the design task. In many cases, we cannot at the outset state the goal to be proved; we want to prove a theorem of the form

$$\texttt{circuit}, \texttt{environment} \vdash \texttt{specification} \tag{8.1}$$

but the implementation (and probably some of the environmental constraints) will only be fixed once the design is complete.

A simple theorem prover for first-order logic, based on these ideas, is described in Chapter 10 of Paulson's book [Paulson, 1991]. A similar approach has been used in Abstract Hardware Limited's LAMBDA theorem prover, which implements a polymorphic Higher-Order Logic, *Lambda Logic (L2)*.

Unification Unification is the process of finding an environment in which two expressions are equivalent. For example, $x + fy$ and $ga + b$ are equivalent in an environment with bindings $x := g\ a$ and $b := f\ y$. In this environment, both expressions prune to $ga + fy$. In this example $x\ y\ a\ b\ f$ and g are first-order (meta-) variables - they can be bound to expressions; LAMBDA also has second-order (meta-) variables such as $\mathcal{P}$ in the expression $\mathcal{P}(x, y)$. $\mathcal{P}$ can be bound to a *syntactic function* describing a method of using x and y to create an expression; such functions are created on-the-fly by second-order matching[4]. Such variables allow us to represent *rule schemas* such as the commutativity of $+$:

$$\frac{hypotheses \vdash \mathcal{P}(b + a)}{hypotheses \vdash \mathcal{P}(a + b)}$$

9. Top-down design

Higher-order logic provides a sufficiently expressive language for modelling both initial and intermediate design goals; we have to keep track of the relationship between an original specification, a partial design, and the obligations which remain to be discharged in order to complete the design. Given a

[4] ISABELLE achieves a similar result using higher-order unification.

specification, the design task is to produce an implementation that *achieves* the specification. Formally, we must prove a theorem

$$\texttt{implementation} \vdash \texttt{specification}.$$

However, the implementation cannot be described until the design is complete; so it cannot be specified in our initial goal. Our goal is thus refined as the design progresses.

We treat the implementation as a variable which is progressively specialised (by instantiation) until the design is complete. Our subgoals have the form:

$$\texttt{partial\_design}, \texttt{pending\_decisions} \vdash \texttt{remaining\_task}$$

Here the implementation has been specialised: `pending_decisions` is a variable which can accommodate further specialisation; `partial_design` represents decisions taken so far (and may itself contain variables whose instantiation will represent further refinement of those decisions). The goal is to be understood informally as, 'find instantiations of the remaining variables such that the design task is achieved'. We can keep track of sufficient conditions (subgoals) for the correctness of the eventual design (goal). By this we mean that the goal follows logically from the subgoals. So the design state is represented by a *rule* (derived by composing primitive rules in predicate logic) of the form,

$$\frac{\begin{array}{c}\texttt{partial\_design}, \texttt{pending\_decisions} \vdash \texttt{remaining\_task}_1\\ \vdots \\ \texttt{partial\_design}, \texttt{pending\_decisions} \vdash \texttt{remaining\_task}_n\end{array}}{\texttt{partial\_design}, \texttt{pending\_decisions} \vdash \texttt{specification}}$$

This should be read as saying, 'if some decisions not yet made, `pending_decisions`, taken together with the partial design, are sufficient to accomplish the remaining tasks, then the conjunction of these decisions with the partial design will implement the entire specification'. This rule, representing the design state, is closely related to the more-general tree used earlier. The premises of the rule are the (unachieved) leaves of the tree; the conclusion of the rule represents (schematically, since uninstantiated variables may remain) what would be achieved at the root if the leaves were achieved. A rule is thus used to represent what has been achieved by a partial design; the relationship between the original specification, the partial design, and any remaining obligations which must be achieved to complete the design. If this rule is valid then so is any later specialisation. Since variables, such as *pending_decisions*, express sharing, specialisations made to discharge one sub-goal are automatically propagated to the others. We represent design steps as transformations which may be applied to produce valid rules from valid rules. By allowing variables in the initial specification we can also accommodate refinement of the specification within the paradigm. The system ensures that only logically valid rules are generated.

Refinements transform rules by changing the 'partial_design' part of the bottom line, as well as totally reforming the top line. That is, the rule above changes to a rule of the same form, but with a simpler set of remaining tasks and a refined design:

$$\frac{\begin{array}{c}\texttt{refined\_design}, \texttt{pending\_decisions} \vdash \texttt{new\_task}_1 \\ \vdots \\ \texttt{refined\_design}, \texttt{pending\_decisions} \vdash \texttt{new\_task}_\mathrm{m}\end{array}}{\texttt{refined\_design}, \texttt{pending\_decisions} \vdash \texttt{specified\_behaviour}}$$

We start the design process from the trivial rule

$$\frac{\texttt{UnknownImp} \vdash \texttt{Spec}}{\texttt{UnknownImp} \vdash \texttt{Spec}}$$

This states the tautology that if `UnknownImp` implements `Spec` then `UnknownImp` implements `Spec`. `UnknownImp` is a variable representing what is left to be synthesised. As the design proceeds we define some of `UnknownImp` and in doing so we produce simpler specifications. The original specification can also contain variables which may instantiated as the design progresses. In simple cases, at the end of the synthesis there will be no premises remaining in which case the design is complete :

$$\frac{}{\texttt{CompleteImp} \vdash \texttt{Spec}}$$

Often, however, there are some constraints remaining such as maximum input range and microcode specifications; such constraints represent operational parameters of the design. The correctness statement is then

$$\frac{}{\texttt{Constraints}, \texttt{CompleteImp} \vdash \texttt{Spec}}$$

To summarise: *post hoc* formal verification starts from a completed design, whereas our approach integrates proof and design, allowing full exploitation of the benefits of abstraction and hierarchy. This integration is made possible by formally representing three things; the ultimate goal, what has already been done (the "partial design") and the "outstanding tasks". We ensure that IF, by making some additions to the present design, we achieve the outstanding tasks THEN we will have achieved the originally specified behaviour. The system's representation of the state of a design is always a valid rule. A new rule is produced with each refinement made by the designer.

New rules are generated by using second-order resolution: the conclusion of one rule is unified with a premise of another to produce a derived rule. Instantiation of variables allows this process to be used to modify the putative theorem as well as to discharge goals. The effects of design decisions are represented by specialisation of variables; this representation automatically propagates the effects of decisions made to accomplish one subtask to other affected subtasks. This allows us to support interactive design (as well as post-design verification), it also provides for refinement of the specification, and the introduction of constraints, as the design proceeds.

Primitive Design Steps

Primitive design steps are represented directly as rules which may be resolved with the rule representing the current design state to produce a new design state.

The simplest design steps represent ways of using particular components. These rules are *derived* from the specifications of the components, and would normally be available in a standard component library. For example, from the specification of the buffer we derive the rule

delayIntro.

$$\frac{\Delta(x, out) \vdash P(in(t))}{\Delta(x, out) \vdash P(out(t+1))}$$

The rule *delayIntro* allows the designer to use $\Delta(x, out)$ to replace any occurrence of $out(t+1)$ by $x(t)$. (In these rules, $\mathcal{P}x$ may be unified with any expression which contains zero or more copies of an expression matching x.)

From the specification

$$ADD(x, y, z) \stackrel{\text{def}}{\equiv} \forall t. z(t+2) = xt + yt$$

of a pipelined adder with latency 2 we derive a more interesting rule:

addIntro.

$$\frac{\begin{array}{c}\mathrm{ADD}(x, y, z) \vdash yt = b \\ \mathrm{ADD}(x, y, z) \vdash xt = a \\ \mathrm{ADD}(x, y, z) \vdash \mathcal{P}(z(t+2))\end{array}}{\mathrm{ADD}(x, y, z) \vdash \mathcal{P}(a+b)}$$

Suppose we wish to implement a sum $out = a + b + c$. We use an $\mathrm{ADD}(x, y, z)$ to simplify the occurrence of $a + b$ to be a single signal $z\ (t + 2)$. This reduces the task to implementing $out = z\ (t\ +\ 2) + c$, and introduces the extra conditions that $y\ t\ =\ b$ and $x\ t\ =\ a$, thus adding timing constraints into the system. We can represent the time-multiplexed reuse of the same adder by applying the same rule to reduce to $out = z(t' + 2)$, with the extra conditions $x\ t' = z\ (t\ +\ 2)$ and $y\ t'\ =\ d$.

In this way, applying the rule generates conditions stipulating the values the inputs of the ADD must take at times t, t' etc. The times t, t' are instantiated to refer to some common reference, say $t' = t + 2$ – this process is called *scheduling*. We can then use delays to eliminate any time offsets, and multiplexors, that supply each input with the appropriate value at each specified time, to satisfy the conditions. The basic multiplexor rules are:

MUXtrue.

$$\frac{\begin{array}{c}\mathrm{MUX}(in_1, in_2, c, out) \vdash c\ t = 1 \\ \mathrm{MUX}(in_1, in_2, c, out) \vdash P(out\ t)\end{array}}{\mathrm{MUX}(in_1, in_2, c, out) \vdash P(in_1\ t)}$$

MUXfalse.

$$\frac{\begin{array}{c}\mathrm{MUX}(in_1, in_2, c, out) \vdash c\ t = 0 \\ \mathrm{MUX}(in_1, in_2, c, out) \vdash P(out\ t)\end{array}}{\mathrm{MUX}(in_1, in_2, c, out) \vdash P(in_2\ t)}$$

Applying the *MUXtrue* rule replaces the first input of a MUX by its output at some time t, and adds the extra condition that the control signal should be true at that time. The second rule does the same thing to the second input, whenever the control signal is false. The use of these rules to multiplex the correct signals to the inputs of a component will automatically replace the conditions on the inputs by conditions which say when the control signals must be true and false.

These rules can be used to implement differing design strategies, corresponding to the availability of components (one or more ADDs, etc). The timing constraints keep track of how the components have been used.

Design Tactics

More general design steps are represented as tactics – transformations which produce valid rules when applied to valid rules. Atomic interaction with the designer can be replaced by the use of tactics, which may be programmed algorithmically or heuristically. This representation of design steps by tactics is quite general; different proof-styles – rewriting, backwards proof by production of sub-goals, forwards proof, and so on – represent different design styles. For example, backward proof can be used to model design synthesis, while local optimisation of a correct design is represented by forward proof. Forward proof steps can express the fact that a transformation preserves behaviour, or express the fact that a transformation preserves behaviour only conditionally – when certain constraints are satisfied. Our design strategy may require that these constraints be discharged immediately, or that they generate new subtasks, or that they be propagated, as constraints on the environment, to modify the specification. These variations are accommodated by the LAMBDA theorem prover, along with many other design styles.

10. A Behavioural Design Tool

In this section we present an example of interactive design, based on Abstract Hardware Limited's LAMBDA tool-set. The design goal of this system is to provide support for the interactive exploration of different architectures for a given behaviour, and provide a machine generated, synthesizable, RTL architecture. This is output in the standard VHDL hardware description language.

LAMBDA includes a general-purpose theorem prover for a polymorphic higher-order predicate logic. It supports the style of proof by refinement of valid rules, that we have discussed earlier, and provides other tools such as a rewriting package. Circuit design by manipulation of textually displayed rules, although in principle possible, soon becomes unmanageable. Instead, we provide an interface using a ready-made, widely-accepted notation for representing circuits – as *schematics.*

DIALOG is the schematic interface to the LAMBDA system. It maintains a link between the internal representation of a circuit (as a net-list, forming part of a rule) and an external, schematic, representation. This link is two-way; DIALOG is a schematic *editor.* DIALOG interprets a user action in editing the schematic as an invocation of a tactic to perform the corresponding action on the current rule. For example, wiring two ports together would result in the unification of the corresponding variables. Alternatively, tactics may be invoked explicitly (via menus or the LAMBDA proof browser); DIALOG presents changes to the current rule as changes to the schematic. Tactics may be implemented to achieve specific design tasks. For example, DIALOG incorporates a tactic to automatically create multiplexed data-paths.

As we have seen, the relational hardware model has limitations. For example, as 'false implies anything', connecting 0 to 1 trivially satisfies any specification; unfortunately real hardware is not so cooperative. This problem is avoided by the imposition of design rules; DIALOG enforces the following rules:

1. Don't connect two outputs together.
2. Don't create zero-delay loops.
3. Don't hide floating inputs.

In addition to its purely logical content, each component description (as used by DIALOG) contains extra information to allow the above rules to be enforced. Dialog also supports the use of busses driven by tri-state drivers [Finn *et al.*, 1992].

Example – an Infinite Impulse Response Filter

To finish with an example (following [Fourman and Mayger, 1989]) that gives some idea of the applicability of the methods we have discussed, we consider an infinite input response filter with control circuitry to reset the filter. The specification states that in normal operation (when reset is not an issue) the $n+3^{rd}$ output is given by a constant multiple of the n^{th} input added to constant multiples of the n^{th}, $n+1^{st}$ and $n+2^{nd}$ outputs. Thus we have

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathrm{SPEC}(a,b,c,d,in,reset,out) \stackrel{\mathrm{def}}{\equiv}} \\
\forall n. & (reset\, n = 1 & \Rightarrow & out\, t = \mathrm{Zero}) \\
\wedge & (reset(n+3) = 0 & \Rightarrow & \\
 & & & a \times out(n+2) \\
 & out(n+3) & = & + \; b \times out(n+1) \\
 & & & + \; c \times out\, n \\
 & & & + \; d \times in\, n)
\end{array}
$$

Here a,b,c and d are constants, and the signal in is represented by a function generating the sequence of signal values. Note that we do not specify how the sequences of signal values are to be mapped onto $times$. We use $+$, and $\times$, to indicate arithmetic on the abstract signal values, and `+` `*` for arithmetic on natural numbers, which are used to index the sequences.

The object of the synthesis exercise is to achieve an implementation which satisfies the specification. Many implementations are possible – some better than others. In general there is a tradeoff of performance against hardware. We will sketch the development of one implementation.

The object of the synthesis is to derive an implementation which achieves this specification .o do this we will use a number of arithmetic units, and decide how the abstract sequence of input values will appear on input ports and be presented to the arithmetic units.

Before descending to this level of detail, we design the control circuitry in terms of the abstract timing of the specification. We begin by stripping off the quantifier. This gives us the goal of designing hardware imposing the relation specified in the body. The restriction that n should not occur free in the assumptions (which comes from the quantifier rule) expresses the requirement that the hardware should not vary with time!

Top-level Control. The top-level control circuitry is trivial – we use a multiplexer to place 0 on the output when the reset signal is high. We can implement our system as

$$\mathsf{H}\, out_1.\ \mathrm{MUX}(reset, \mathrm{Zero}, out_1, out) \parallel \mathrm{SS}(in, out, out_1)$$

– provided $\mathrm{SS}(in, out, out1)$ satisfies the specification

$$\forall t.\ out_1(t+3) = a \times out(t+2) + b \times out(t+1) + c \times out\, t + d \times in\, t$$

Datapath. We choose to implement the filter described above by using pipelined adders and multipliers with 2 and 4 unit delays respectively. For example, the definition of ADD is just

$$ADD(x,y,z) \stackrel{\mathrm{def}}{\equiv} \forall t.\ z\,(t+2) \equiv x\, t + y\, t$$

It is impossible to implement this specification directly using these components. We cannot produce $out(n+2)$ and $out(n+3)$ in successive clock ticks – we need time to perform a multiplication and an addition. We save the situation by deciding that the notion of time used in the specification is abstract. We will implement the computation using a concrete time that has a number of ticks for each abstract tick.

We choose to relate the internal time scale of these components to the abstract sequencing of the specification by an abstraction relation which specifies that the external signals are given by sampling on every k^{th} tick of the internal clock. We introduce two components – abstract hardware – to represent this abstraction.

$$MACRO(k, x, y) \stackrel{\text{def}}{\equiv} (y \equiv \mathtt{fn}\, t \Rightarrow x\,(k * t))$$

$$MICRO(k, x, y) \stackrel{\text{def}}{\equiv} (x \equiv \mathtt{fn}\, t \Rightarrow y\,(k * t))$$

The only difference between these components is that, for MICRO, x is an input, and y an output; whereas MACRO 'operates' in the other direction. Formal logic can treat abstractions such as MICRO and MACRO on the same level as a component.

We introduce abstraction components,

$$\mathrm{MICRO}(N, out, micout) \parallel \mathrm{MICRO}(N, in, micin) \parallel \mathrm{MACRO}(N, mac, out_1)$$

to relate the concrete implementation time to the abstract specification time. This reduces our task to designing circuitry such that

$$\begin{array}{rcl} mac(N * (t' + 3)) & = & a \times micout(N * (t + 2)) \\ & + & b \times micout(N * (t + 1)) \\ & + & c \times micout(N * t) \\ & + & d \times micin(N * t) \end{array}$$

Since we may take the sampling ratio , N, as large as we need, we now have plenty of time. A value for N is determined during the design process. For a fixed clock rate, we must minimise k so as to increase throughput.

We use the ADD and MUL components to implement the operations. We allow a component to be used pipelined for more than one operation, so an addition is achieved by assigning the two values to be added to the inputs of the ADD at a time $t1$, and then replacing their sum by the output of the ADD at time $t1 + 2$.

Before allocating the operations to components, and scheduling times when they will be performed, we use the associativity of addition to re-arrange the additions to associate to the right. This allows us to carry out some arithmetic while we wait for other values to be produced, and so increases the opportunities for pipelining. Rearranging equations may appear trivial, since the designer instinctively *knows* that addition and multiplication are associative. In the case of finite precision arithmetic, however, addition is not necessarily associative and such a rearrangement is a common source of error. Our methodology ensures correctness by allowing the re-association while keeping track of the conditions under which it is valid (such as no overflows occurring).

The procedure we use for allocation in this example is to add one component of each type, and use these to perform all the additions and multiplications – we could use more hardware to improve performance. We must then *schedule* the operations – assign times to the various operations. The times are assigned using a *just in time* scheduling algorithm, which attempts to maximise the speed of the design by only performing operations just before their results are needed. During the course of this scheduling, values for m and n are calculated. They will depend on the way in which the ADD and MUL components are used. In our example, the scheduling algorithm produces a solution that uses 8 implementation-level clock ticks to each time step at the specification level.

Once scheduling has been performed, we use a design tactic to automatically introduce multiplexers and delays so as to connect the arithmetic components together and fulfil the various conditions on their input values. This leads to timing conditions on the control signals of the multiplexers, which are left in the final design state. There is no need to 'prove' these; they just specify the multiplexers' control requirements. In general, many blocks being synthesised at one level may generate such requirements. The requirements generated in this way form part of the specification at the next level, and may be considered together or re-partitioned as appropriate. Partitioning at one level of abstraction does not force partitioning of the resulting control requirements at the next level. As a result, requirements common to several blocks need only be implemented once.

Different implementations can be explored and the trade-offs between speed and extra components can be investigated. Our approach to design is integrated with formal proof. We can safely explore design alternatives which would otherwise require manual rewriting of the specification. Once we have finalised a design for this sub-block, we return to the top-level, push the abstractions MACRO and MICRO to the inputs and output, and propagate the timing parameters through the control circuitry. In this example this step is trivial since the control logic is entirely combinational; if it contained delays, these would be modified to accommodate the abstraction.

The final design (Figure 10.1) consists of a number of components connected in accordance with the design rules, and a specification (Figure 10.2) of the control signals to be applied when using the circuit. Last, but not least, we have a correctness theorem:

$$\mathrm{IMP}, \mathrm{CTRL} \vdash \mathrm{SPEC}(a, b, c, d, in, reset, out)$$

The MICRO abstract components now only occur on the inputs; they specify that we must present successive input values at every eighth (concrete) clock tick. Similarly, the MACRO abstract component specifies how the concrete output signal should be sampled to give the abstract output sequence.

$$
\begin{array}{lclcl}
\mathrm{IMP} & \stackrel{\mathrm{def}}{\equiv} & \mathrm{MICRO}(7, in, micin) & \| & \mathrm{MACRO}(7, w, out) \\
& \| & \mathrm{MICRO}(7, reset, c1) & & \\
& \| & \mathrm{ADD}(w18, w19, w20) & \| & \mathrm{MUL}(w15, w16, w17) \\
& \| & \mathrm{MUX}(c1, \mathrm{Zero}, w20, w) & \| & \mathrm{MUX}(c4, w17, w20, iw22) \\
& \| & \mathrm{MUX}(c5, iw22, iw17, w19) & \| & \mathrm{MUX}(c3, iw11, w17, w18) \\
& \| & \mathrm{MUX}(c4, iw6, iw10, iw18) & \| & \mathrm{MUX}(c3, iw19, iw18, w16) \\
& \| & \mathrm{MUX}(c4, w, iw7, iw19) & & \\
& \| & \mathrm{DELAYN}(6, w20, iw17) & \| & \mathrm{DELAYN}(3, iw7, iw10) \\
& \| & \mathrm{DELAYN}(5, micin, iw6) & & \\
& \| & \mathrm{DELAY}(w17, iw11) & \| & \mathrm{DELAY}(w, iw7)
\end{array}
$$

(DELAYN is a parameterised , n-unit delay.)

Fig. 10.1. Final Design

$$
\begin{array}{llrcr}
\mathrm{CTRL} & \stackrel{\mathrm{def}}{\equiv} & c3(7*t'+7)=1 & \wedge & c3(7*t'+4)=0 \\
& \wedge & c3(7*t'+5)=0 & \wedge & c3(7*t'+19)=0 \\
& \wedge & c3(7*t'+11)=0 & \wedge & c3(7*t'+15)=1 \\
& \wedge & c3(7*t'+9)=1 & & \\
& \wedge & c5(7*t'+19)=0 & \wedge & c5(7*t'+11)=1 \\
& \wedge & c5(7*t'+9)=1 & & \\
& \wedge & c4(7*t'+15)=0 & \wedge & c4(7*t'+7)=1 \\
& \wedge & c4(7*t'+4)=0 & \wedge & c4(7*t'+5)=1 \\
& \wedge & c4(7*t'+19)=1 & \wedge & c4(7*t'+11)=0 \\
& \wedge & c4(7*t'+9)=1 & & \\
& \wedge & w15(7*t'+15)=a & \wedge & w15(7*t'+7)=b \\
& \wedge & w15(7*t'+4)=c & \wedge & w15(7*t'+5)=d
\end{array}
$$

Fig. 10.2. Control Specification

Using the LAMBDA system, it takes only a few minutes of interaction to produce the design we have presented here – or any of the many others, resulting from alternative allocation strategies, that a designer might want to consider. Indeed, our specialised scheduling tactics turn out to be re-usable for a wide variety of designs. They can be packaged with a graphical interface designed to represent the scheduling and allocation decisions taken by a designer, and applied effectively by users ignorant of formal methods to produce RTL architectures from behavioural descriptions.

Acknowledgement. Thanks to all my colleagues at Abstract Hardware Limited, who have implemented the LAMBDA–DIALOG system, particularly Simon Finn, Eleanor Mayger, and Mick Francis, who collaborated in the development of the approach to Formal System-Design outlined here. Thanks also to colleagues and students at Edinburgh: James McKinna for interpreting, developing, and (most important) recording my 1988 lectures at Edinburgh on some of these ideas; Peter Sewell for illuminating discussions about abstraction; John Power for concrete categorical contributions; and Stuart Anderson for abstract advice. Finally, thanks to the directors, organisers, and students of the Marktoberdorf Summer School – especially to Tony Hoare for his patient and gentle encouragement.

A. Appendix: The Category of Circuits

Earlier, we described circuits syntactically. More concretely, we can describe circuits mathematically, as a freely generated structure. We use the language of category theory. First we describe the structure of the category of circuits, then we *define* the category of circuits to be a freely generated category with this structure.

Let $B = S_{fin}$ the category of finite sets (of port names) and functions. We write $\mathtt{ctt}_U$ for the circuits of sort U. A function $U \xrightarrow{f} V$ in S_{fin} is thought of as *renaming* ports in U, using names from V. It induces a renaming operation $\mathtt{ctt}_U \xrightarrow{f^\star} \mathtt{ctt}_V$, syntactically a substitution, on circuits. The action of $f^\star$ is straightforward: ports given the same names are connected; ports not in the image of f are added.

B^{op} is a category of formal products; we call it the category of *sorts*. Circuits form a *fibration*, $\mathtt{ctt} \xrightarrow{c} B^{op}$. The fibres, $\mathtt{ctt}_U$, have finite products, $\|$, T, preserved by renaming. The square and triangle below commute:

$$\begin{tikzcd}
& 1 \arrow[dl, "\mathsf{T}"'] \arrow[dr, "\mathsf{T}"] & \\
\mathtt{ctt}_U \arrow[rr, "f^\star"] & & \mathtt{ctt}_V \\
\mathtt{ctt}_U \times \mathtt{ctt}_U \arrow[u, "\|"] \arrow[rr, "f^\star \times f^\star"] & & \mathtt{ctt}_V \times \mathtt{ctt}_V \arrow[u, "\|"']
\end{tikzcd}$$

The fibration has "existential" quantification along projections in B^{op}, also preserved by renaming. That is to say, for $W \times U \xrightarrow{\pi} U$ we have a left adjoint, H_π, to $\pi^\star$.

$$\begin{tikzcd}
\mathtt{ctt}_{W\times U} \arrow[r, shift right, "\mathsf{H}_\pi"'] \arrow[r, phantom, "\top"] & \mathtt{ctt}_U \arrow[l, shift right, "\pi^\star"']
\end{tikzcd}$$

where the H_π satisfy the "Beck-Chevalley" condition. The following square commutes:

$$\begin{tikzcd}
\mathtt{ctt}_{W\times U} \arrow[r, "\mathsf{H}_\pi"] \arrow[d, "W \times f^\star"'] & \mathtt{ctt}_U \arrow[d, "f^\star"] \\
\mathtt{ctt}_{W\times V} \arrow[r, "\mathsf{H}_\pi"'] & \mathtt{ctt}_V
\end{tikzcd}$$

More generally, $f^\star$ has a left adjoint, H_f, and if the square on the left is a pull-back, the square on the right commutes:

$$\begin{tikzcd} U \times_W V \arrow[r, "\hat{f}"] \arrow[d, "\hat{g}"'] & V \arrow[d, "g"] \\ U \arrow[r, "f"'] & W \end{tikzcd} \qquad \begin{tikzcd} cct_{U \times_W V} \arrow[r, "\mathsf{H}_{\hat{f}}"] \arrow[d, "\hat{g}^\star"'] & \mathtt{ctt}_V \arrow[d, "g^\star"] \\ cct_U \arrow[r, "\mathsf{H}_f"'] & \mathtt{ctt}_W \end{tikzcd}$$

Definition A.1. *A collection of primitive devices is a set, P, (viewed as a discrete category), with sorts given by a functor $P \xrightarrow{p} B^{op}$. We define the* category of circuits *generated by P, indexed over sorts, to be a fibration, $\mathtt{ctt} \xrightarrow{c} B^{op}$, freely generated by p, with fibre-wise finite products preserved by renaming, and hiding satisfying the Beck-Chevalley condition along projections.*

$$\begin{tikzcd} P \arrow[rr] \arrow[dr, "p"'] & & \mathtt{ctt} \arrow[dl, "c"] \\ & B^{op} & \end{tikzcd}$$

Circuits are the objects of the category $\mathtt{ctt}$; this construction also provides them with intrinsic structure. A morphism $A \xrightarrow{f} B$ in the category of circuits may be viewed as an implementation of A in terms of B. The structure allows us to associate to every circuit expression ϕ a circuit, $[\phi]$, whose sort is the set of free variables of ϕ. We will frequently elide the distinction between ϕ and $[\phi]$, and speak of "the circuit, ϕ".

Types Circuits with typed wires may be represented in a similar way. We introduce types by changing the base category of sorts. The category ${\mathcal{S}_{fin}}^{op}$ is the free category with associative products, on one generator. In general, we replace it by the free category with associative products generated by the set of types. This category may be represented by replacing finite sets and function by finite typed-sets (each element, x, has a type, $\tau(x)$) and type-preserving functions, (f such that $\tau(f(x)) = \tau(x)$). Type conversions, such as the conversion between a bus and a bundle of wires, may be added by adding formal morphisms to this category.

These categories of "circuits" are of course of more general application; they capture notions of hierarchical structure that pervade many areas of computer science.

Semantics

To provide semantics for circuits, we construct fibrations $\mathcal{M} \xrightarrow{m} \mathcal{B}^{op}$, with the structure of composition and hiding. The universal property of the circuit fibration then gives a diagram

$$\begin{tikzcd} \mathtt{ctt} \arrow[rr, "{[-]_{\mathcal{M}}}"] \arrow[dr, "c"'] & & \mathcal{M} \arrow[dl, "m"] \\ & \mathcal{B}^{op} & \end{tikzcd}$$

which assigns a model $[\phi]_{\mathcal{M}}$ to each circuit, ϕ.

Bibliography

[Bell and Newell, 1971] G.C. Bell and A. Newell. *Computer Structures: Reading and Examples*. McGraw-Hill, 1971.

[Bryant, 1984] R.E. Bryant. A switch-level model and simulation for MOS digital systems. *IEEE Transactions on Computers*, C-33:160–177, 1984.

[Busch, 1991] Holger Busch. Proof-based transformation of formal hardware models, 1991.

[Carlson, 1991] Steve Carlson. *Introduction to HDL-Based Design Using VHDL*. Synopsys Inc, 700 East Middlefield Road, Mountain View, CA, 1991.

[Cohn, 1988] Avra J. Cohn. A proof of correctness of the Viper microprocessor: The first level. In G. Birtwhistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71. Kluwer Academic Publishers, 1988.

[Cohn, 1989] Avra Cohn. Correctness properties of the Viper block model: The second level. In *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91. Springer-Verlag, New York, 1989. Also Report No. 134, Computer Laboratory, University of Cambridge, May 1988.

[Finn *et al.*, 1989] S. Finn, M. Fourman, M. Francis, and R. Harris. Formal system design—interactive synthesis based on computer-assisted formal reasoning. In Dr. Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Volume 1*, pages 97–110. Elsevier Science Publishers, B.V. North-Holland, Amsterdam, 1989.

[Finn *et al.*, 1992] S.P. Finn, M.D. Francis, R.B. Hughes, and G. Musgrave. Formal tools for tri-state design in busses. In L.J.M. Claesen and M.J.C. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications (A-20)*, pages 459–474. Elsevier Science publishers B.V. (North Holland), 1992.

[Fourman and Hexsel, 1991] M.P. Fourman and R Hexsel. Formal synthesis. In Graham Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 245–264. Springer-Verlag, 1991.

[Fourman and Mayger, 1989] M.P. Fourman and Eleanor M. Mayger. Formally based systems design—Interactive hardware scheduling. In *Proceedings of VLSI '89*, pages 101–112, Munich, Germany, 1989.

[Fourman *et al.*, 1987] M.P. Fourman, R.C. Holte, W.J. Palmer, and R.M. Zimmer. Top-down design and bottom-up proof. In *Proceedings of the Electronic Design Automation Conference*, pages 617–628, Wembley, 1987.

[Fourman *et al.*, 1988] M.P. Fourman, W.J. Palmer, and R.M. Zimmer. Proof and synthesis. In *Proceedings ICCD '88*, Rye Brook, NY, 1988.

[Gordon *et al.*, 1979] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. A mechanised logic of computation. In *Edinburgh LCF: Lecture Notes in Computer Science—Volume 78*. Springer-Verlag, 1979.

[Gordon, 1985] Mike Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, 1985.

[Gordon, 1986] Mike Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177. North-Holland, 1986.

[Hanna and Daeche, 1983] F.K. Hanna and N. Daeche. Specification and verification using higher-order logic. In Uehara and Barbacci, editors, *Proc. 7th Inst. Symp. on Computer Hardware Description Languages and their Applications*. Elsevier North Holland, 1983. (CHDL 85).

[Hanna *et al.*, 1989] F.K. Hanna, M. Longley, and N. Daeche. Formal synthesis of digital systems. In Dr. Luc Claesen, editor, *IMEC-IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, Volume 2*, pages 532–548. Elsevier Science Publishers, B.V. North-Holland, Amsterdam, 1989.

[Hoare, 1985] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International Series in Computer Science, 1985.

[Hunt, 1987] W.A. Hunt. The mechanical verification of a microprocessor design. In *HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 89–132. North-Holland, 1987.

[Hunt, 1989] Warren A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5:429–, 1989. Special Issue on System Verification.

[Mayger and Fourman, 1991] E.M. Mayger and M.P. Fourman. Integration of formal methods with system design. In A. Halaas and P.B. Denyer, editors, *IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration*, Edinburgh, Scotland, August 1991.

[Melham, 1988] Thomas F. Melham. Abstraction mechanisms for hardware verification. In G. Birtwhistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification, and Synthesis*, pages 267–291. Kluwer Academic Publishers, 1988.

[Milner *et al.*, 1990] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[Milner, 1983] R. Milner. How ML evolved. *Polymorphism (the ML/LCF/Hope Newsletter*, 1(1), January 1983.

[Milner, 1989] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[Moskowski, 1985] Ben Moskowski. A temporal logic for multilevelreasoning about hardware. *IEEE Computer*, 2(18):10–19, February 1985.

[Paulson, 1987] L. Paulson. Natural deduction proof as higher-order resolution. *Logic Programming*, 3:237–258, 1987.

[Paulson, 1991] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Zhou and Hoare, 1992] Chaochen Zhou and C.A.R. Hoare. A model for synchronous swithching circuits and its theory of correctness. *Formal Methods in System Design*, 1:7–28, 1992.