Technical Report

UCAM-CL-TR-106
ISSN 1476-2986

Number 106

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Abstraction mechanisms for hardware verification

Thomas F. Melham

May 1987

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

© 1987 Thomas F. Melham

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Contents

Introduction 1

1 Abstraction 2

2 Four Types of Abstraction 3

3 Specification using Higher Order Logic 3
- 3.1 Introduction to Higher Order Logic 4
- 3.2 Specifying Behaviour in Logic 5
- 3.3 Specifying Structure in Logic 5

4 Formalising Structural Abstraction 6
- 4.1 An Example 7

5 Formalising Behavioural Abstraction 8
- 5.1 An Example 9

6 Formalising Data Abstraction 10
- 6.1 An Example 11

7 Formalising Temporal Abstraction 13
- 7.1 Constructing Temporal Abstraction Functions 15
  - 7.1.1 Defining Timeof 16
  - 7.1.2 Using Timeof to Define Abstractions 17
- 7.2 An Example 18

8 Conclusion 21

Acknowledgements 22

References 22

Introduction

Recent advances in microelectronics have given designers of digital hardware the potential to build electronic devices of unprecedented size and complexity. With increasing size and complexity, however, it becomes increasingly difficult to ensure that such systems will not malfunction because of design errors.

This problem has prompted some researchers to look for a firm theoretical basis for correct design of hardware systems. Mathematical methods have been developed to model the functional behaviour of electronic devices and to verify, by formal proof, that their designs meet rigorous specifications of intended behaviour. Proving the correctness of a design using such hardware verification techniques typically involves:

- Writing a set of formulas S which express the specification of the device whose design is to be proven correct.
- Describing the design or implementation of the device by a second set of formulas I.
- Proving that a 'satisfaction' or correctness relation holds between the sets of formulas I and S—i.e. that the implementation satisfies the specification.

This process is usually carried out at each level of the structural hierarchy of the design. The top level specification is shown to be satisfied by some connection of components; the specifications of these components are in turn shown to be satisfied by their implementations, and so on—until the level of primitive components is reached.

If this technique is to be used to verify large and complex designs, it is clear that the 'satisfaction relation' that is used cannot be strict equivalence. Otherwise, at each level of the design hierarchy, the specifications will contain all the information present in the design descriptions at the level below. For large and complex designs this means that the specifications at the upper levels of the hierarchy will themselves become so large and complex that they can no longer be seen to reflect the intended behaviour of the device. Furthermore, proofs of equivalence between such complex formulas will become unmanageable, even when automated theorem proving tools are used.

A satisfaction or correctness relation based on the idea of *abstraction*, rather than equivalence, is the key to making formal verification of large designs tractable. The aim of this paper is to discuss the role of abstraction in hardware verification and to show how various types of abstraction can be formalised and used to control the complexity of specifications and correctness proofs.

The organisation of the paper is as follows. Section 1 introduces the idea of abstraction and its role in controlling the complexity of hardware verification. In section 2 an informal description is given of four basic abstraction mechanisms that can be used in proofs of hardware correctness. Section 3 contains a brief introduction to the logic that will be used to prove devices correct, and reviews the standard techniques for formally specifying the behaviour and structure of hardware in logic. Sections 4 through 7 show how each of the four basic abstraction mechanisms can be formalised in logic; in each

section, an example is given to illustrate the use of the abstraction mechanism being discussed.

1 Abstraction

Abstraction involves the suppression of irrelevant detail or information, in order to concentrate on the things of interest. In hardware verification, the larger and more complex the system we are describing, the more irrelevant detail must be ignored to keep specifications small. Thus, in general, specifications will be abstractions of the devices actually implemented. This means that the satisfaction or correctness relation that is used must involve abstraction mechanisms that, by removing information, relate the more detailed implementation descriptions to their abstract specifications.

The specification of a microprocessor, for example, would include a description of the effect of each instruction on the machine registers. The description of a microcoded implementation of the system would contain far more information, including the exact sequence of microinstructions which implements each macroinstruction. To show that this design is correct, we would have to prove that each microcode sequence produces the effect on the registers that is required by the specification of the corresponding macroinstruction. Here, the abstraction mechanism used to relate the levels of description is the composition of state changes; by composing a sequence of low level state changes to get a single transition, we 'hide' from the top level specification information about the intermediate states that occur during macroinstruction execution. At the abstract level, we only know *what* each instruction does; at the more detailed implementation level, we also know *how* the instruction does it.

Another example is the correctness proof of a multiplier circuit. A multiplier is constructed from hundreds of transistors, each of which exhibits quite complex behaviour. Furthermore, this device performs multiplication by some particular method, e.g. repeated addition. All this information will be explicit in the formulas that describe the multiplier's design but will be irrelevant to its abstract specification, which will simply state that it correctly performs multiplication.

The correctness proof of this device will show that the logical formulas expressing its specification represent a valid abstract view of the formulas describing its design. The design description will include, for example, assertions about voltages present at certain points in the device; these must be translated into assertions in the specification about the numbers being multiplied. Such a translation is an abstraction mechanism for relating *numbers* in the specification to the *voltages* in the design which represent them.

If we have proved that a formal specification represents a valid abstract view of a device, we can use it in place of the more detailed implementation description when proving the correctness of a larger system which contains the device as a component. This will allow us to derive a more clear and concise behavioural description of the larger system's implementation than would otherwise be possible. If we do this at each level of the proof of a hierarchically structured design, we can control the size and complexity of the entire proof. In this way, abstraction mechanisms combine with hierarchical structuring to make it possible to handle proofs of large systems.

2 Four Types of Abstraction

The type of abstraction most fundamental to formal verification of hardware is *structural* abstraction—the suppression of information about a device's internal structure. The idea of structural abstraction is that the specification of a device should not reflect its internal construction, but only its externally observable behaviour. The description of a device's implementation, however, must contain explicit information about its structure; the mechanism of structural abstraction therefore involves formalising the idea that such information concerns 'internal' structure.

A second type of abstraction, *behavioural* abstraction, concerns specifications that only partially define a device's behaviour, leaving unspecified its behaviour in certain states or for certain input values. Such partial specification is appropriate, for example, when it is known that a device will never have to operate in certain environments and it is therefore unnecessary to specify its expected behaviour in *all* environments. The mechanism of behavioural abstraction serves to relate partial specifications to implementation descriptions that fully define the device's behaviour, by showing that they agree on the device's behaviour for all states and inputs that are of interest, i.e. that are defined by the specification.

The concept of *data* abstraction is well known from programming language theory. There are several abstract data types which are useful for formal specification of hardware; a simple example is the type of boolean truth values, a data abstraction from analog signals routinely used by hardware designers to reason about circuits. Another example is the type of n-bit integers, represented at the less abstract level by vectors of booleans. A data abstraction step consists of constructing a mapping from the data types of an implementation description to the more abstract data types of the specification. This mapping is then used to show that the operations carried out on the low level data types correctly implement the desired operations on the high level types.

A final type of abstraction is *temporal* abstraction, in which the sequential or time-dependent behaviour of a device is viewed at different 'grains' of discrete time. An example of temporal abstraction is the unit delay or register, implemented using an edge triggered flip flop. At the abstract level of description the device is specified as a unit delay, one 'unit' of discrete time corresponding to the clock period; at the detailed level of description the grain of time is finer, several units of time corresponding to the delay through a gate. The proof of a microcoded computer design, mentioned above, also involves temporal abstraction; at the low level there is one microcode step per unit of discrete time, and at the higher level there is one machine instruction step per unit of discrete time. Relating the levels of temporal abstraction involves mapping points or periods of low level time to points or periods of high level time and showing that a many-step low level computation implements a one-step high level computation.

3 Specification using Higher Order Logic

The formalism that we use to specify and verify hardware is a variety of higher order logic, adapted for this purpose by Mike Gordon at the University of Cambridge. In

this section, we first give a very brief introduction to this logic; a more complete and formal presentation is given by Gordon in [2,3]. We then review how the behaviour and structure of digital hardware can be specified using higher order logic.

3.1 Introduction to Higher Order Logic

Higher order logic includes terms corresponding to the standard notation of predicate calculus. Thus, '$P\,x$' expresses the proposition that x has property P and '$R(x,y)$' means that the relation R holds between x and y. Higher order logic has the usual logical operators $\neg$, $\wedge$, $\vee$, $\supset$ and $\equiv$ denoting negation, conjunction, disjunction, implication and equivalence respectively. The universal and existential quantifiers $\forall$ and $\exists$ are used to express the concepts of *every* and *some*; '$\forall x.P\,x$' means that P holds for every value of x and '$\exists x.P\,x$' means that P holds for some (i.e. at least one) value of x. Terms of the form '$(c \Rightarrow t_1 \mid t_2)$' denote the conditional 'if c then t_1 else t_2'. The term '$f \circ g$' denotes the composition of the functions f and g defined by $(f \circ g)(x) = f(g(x))$. The constants T and F denote the truth values *true* and *false* respectively.

The logic we use is *higher-order*. That is, variables are allowed to range over functions and predicates. The well ordering property of the natural numbers, for example, can be expressed using a variable P ranging over predicates:

$$\forall P.\,(\exists n.\,P\;n) \supset (\exists n.\,P\;n \wedge \forall n'.\,n'{<}n \supset \neg P\;n)$$

This states that if any predicate P is true of at least one number then there is a smallest number for which P is true.

In higher order logic, functions and predicates can take other functions and predicates as arguments. Functions can also yield functions as results. Consider, for example, the function Rise defined as follows:

$$(\mathsf{Rise}\;ck)(t) \equiv \neg ck(t) \wedge ck(t{+}1)$$

Rise is intended to express the notion that a clock ck rises at time t. The clock, modelled by a function ck from natural numbers to booleans, is given as the argument to Rise. The result of applying Rise to ck is also a function from numbers to booleans. Thus Rise is an example of a function that both takes a function as an argument and yields a function as a result.

Higher order logic is a typed logic; every term of the logic has a *type*, though these are usually omitted when a term is written down. Informally, types can be thought of as denoting sets of values and the value of a term as an element of the set denoted by the term's type. The basic types of the logic include the type of natural numbers, *num* and the type of boolean truth values *bool*. Types can be built from other types using *type operators*. For example, the type of functions from *num* to *bool* is denoted by '$num{\rightarrow}bool$', using the infix type operator '$\rightarrow$'.

Writing '$tm{:}ty$' indicates explicitly that a term tm has type ty. For example, the term 'Rise ck' can be written with explicit type information as

$$(\mathsf{Rise}\;ck){:}num{\rightarrow}bool$$

or even as

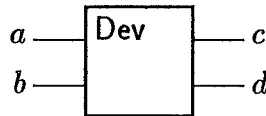
$$((\mathsf{Rise}{:}(num{\rightarrow}bool){\rightarrow}(num{\rightarrow}bool))\ (ck{:}num{\rightarrow}bool)){:}num{\rightarrow}bool$$

though it is seldom necessary to give this much type information.

An important primitive constant in the logic we are using is the ε-operator, the informal semantics of which are as follows. If $\mathsf{P}[x]$ is a predicate involving a variable x of type ty then $\varepsilon x.\, \mathsf{P}[x]$ denotes some value of type ty such that P is true of that value. If there is no such value (i.e. $\mathsf{P}[v]$ is false for each value v of type ty) then $\varepsilon x.\, \mathsf{P}[x]$ denotes some fixed but arbitrarily chosen value of type ty. Thus, for example, '$\varepsilon n.\, 4<n \wedge n<6$' denotes the value 5, '$\varepsilon n.\, (\exists m.\, n = 2\times m)$' denotes an unspecified even natural number, and '$\varepsilon n.\, n<n$' denotes an arbitrary natural number. For further discussion of the ε-operator, see [2] or [10].

3.2 Specifying Behaviour in Logic

The technique for specifying hardware behaviour with higher order logic is well established; see, for example, Gordon's paper [4] or the work done by Hanna and Daeche [7]. Hardware devices can be specified by predicates that describe their behaviour in terms of the values on their external ports. Consider, for instance, a device with external ports a, b, c and d:



Such a device is specified by a four-place predicate Dev, defined such that '$\mathsf{Dev}(a, b, c, d)$' is true exactly when the combination of the values of a, b, c and d is one that could occur on the corresponding ports of the device.

Devices that exhibit time-dependent behaviour can be specified by allowing the values on the external ports to vary over time. An inverter with δ time units of delay, for example, can be specified by the predicate Inv defined as follows.

$$\mathsf{Inv}(i, o) \equiv \forall t.\, o(t{+}\delta) = \neg i(t)$$

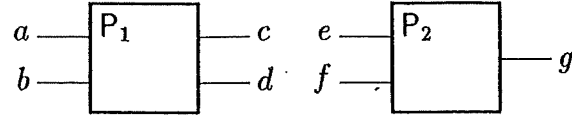
In this specification, instants of discrete time are modelled by the natural numbers; i and o are functions from num to $bool$ giving the sequence values on the input and output ports at successive moments of time. Such functions are sometimes called 'signals'. The predicate Inv specifies an inverter with δ delay by asserting that for all times t, the value of the output signal at time $t{+}\delta$ is the negation of the value of the input signal at time t.

3.3 Specifying Structure in Logic

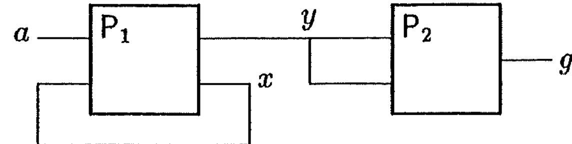
Given predicates that specify the behaviours of the components of a design, we wish to derive a logical expression that describes both how these components are interconnected and what the net behaviour of the connected components will be. In logic, this is

done simply by forming a conjunction of the predicates that specify the components, identifying variables that represent connected ports (see Gordon's paper [4]).

Suppose, for example, $\mathsf{P}_1(a, b, c, d)$ and $\mathsf{P}_2(e, f, g)$ are the predicates that specify the two components P_1 and P_2 of a design:



and suppose that d is connected internally to b, and c to both e and f:



The resulting structure can be specified in logic by the predicate Imp, defined as follows:

$$\mathsf{Imp}(a, x, y, g) \equiv \mathsf{P}_1(a, x, y, x) \wedge \mathsf{P}_2(y, y, g)$$

The predicates P_1 and P_2 define the allowable values on the ports of the two components; the predicate Imp simply asserts that the values allowed on the ports of the connected components are just those allowed by *both* P_1 and P_2.

4 Formalising Structural Abstraction

We now consider how the mechanism of structural abstraction can be formalised in higher order logic. The abstract specification of a device's intended behaviour will, in general, be given in terms that do not make reference to any particular structure that realizes the behaviour. The formal description of some particular implementation, however, will contain information about its structure, in the form of constraints on signals not present in the specification. The correctness relation that we formulate must express the idea that these constraints are 'internal' to the implementation description.

Consider, for example, the design described by the predicate Imp, defined above. This predicate defines a logical relation that involves the external ports a and g as well as the internal signals x and y. An abstract specification of this device, however, will be given only in terms of the external ports a and g, and will not include information about the values of x and y. Suppose that such a specification is given by $\mathsf{Spec}(a, g)$. The correctness relation for this device must state that Spec and Imp agree on what values are allowed to occur on the external ports a and g, given that x and y are internal signals.

We begin by considering what it means for x and y to be 'internal' signals, hidden from the device's environment. If x and y are internal signals, then their values are determined solely by the constraints imposed internally by the components of the device. Thus, if '$\mathsf{Imp}(a, x, y, g)$' is true, we can interpret this as meaning that x and y are the values generated internally when the values a and g are present on the external ports.

Now, two values a and g are allowed to occur on the external ports just when it is *possible* for internal signals to be generated such that $\mathsf{Imp}(a,x,y,g)$ comes out true. Therefore two values a and g are allowable on the external ports just when there are *some* values x and y such that $\mathsf{Imp}(a,x,y,g)$ holds, i.e. when it is the case that $\exists\, x\, y.\mathsf{Imp}(a,x,y,g)$.

A correctness relation involving structural abstraction can therefore be formulated as follows. The design described by Imp is correct with respect to the abstract specification Spec if we can prove that:

$$\forall\, a\, g.\ \mathsf{Spec}(a,g) \equiv \exists\, x\, y.\mathsf{Imp}(a,x,y,g)$$

That is, Spec and Imp agree on what values are allowed on the external ports a and g, given that the signals x and y are internal to the design described by Imp.

Thus, correctness relations involving structural abstraction can be expressed formally in higher order logic by 'hiding' internal signals in the design description using the existential quantifier and proving that the resulting term is logically equivalent to the specification. This technique of using '$\exists$' to hide internal structure is fairly common; see, for example, [1,4] or [9].

4.1 An Example

This example illustrates the use of structural abstraction in a very simple correctness proof. A delay device with δ time units of delay can be specified by the predicate Delay, defined as follows:

$$\mathsf{Delay}(i,o) \equiv \forall t.\, o(t+\delta) = i(t)$$

A series connection of two such devices can be specified by the predicate Imp defined by:

$$\mathsf{Imp}(i,x,o) \equiv \mathsf{Delay}(i,x) \wedge \mathsf{Delay}(x,o)$$

If the internal signal x is hidden, this connection of devices should satisfy the specification given by:

$$\mathsf{Spec}(i,o) \equiv \forall t.\, o(t+2\delta) = i(t)$$

Formally, the correctness relation that we want to prove is:

$$\forall\, i\, o.\ \mathsf{Spec}(i,o) \equiv \exists\, x.\ \mathsf{Imp}(i,x,o)$$

The proof of this correctness statement proceeds as follows:

1. By the definition of Delay:

$$\exists\, x.\mathsf{Imp}(i,x,o) \equiv \exists\, x.(\forall t.\, x(t+\delta){=}i(t) \wedge \forall t.\, o(t+\delta){=}x(t))$$

2. By the symmetry of equality:

$$\exists\, x.\mathsf{Imp}(i,x,o) \equiv \exists\, x.(\forall t.\, x(t+\delta){=}i(t) \wedge \forall t.\, x(t){=}o(t+\delta))$$

3. Selective rewriting with the equation '$\forall t.\, x(t){=}o(t{+}\delta)$' yields:

$$\exists\, x.\mathsf{Imp}(i,x,o) \equiv \exists\, x.(\forall t.\, o((t{+}\delta){+}\delta){=}i(t) \wedge \forall t.\, x(t){=}o(t{+}\delta))$$

4. Since x does not occur in the left hand conjunct, the scope of the existential quantifier can be limited to the right hand conjunct:

$$\exists\, x.\mathsf{Imp}(i,x,o) \equiv \forall t.\, o((t{+}\delta){+}\delta){=}i(t) \wedge \exists\, x.(\forall t.\, x(t){=}o(t{+}\delta))$$

5. The right hand conjunct '$\exists\, x.\forall t.\, x(t){=}o(t{+}\delta)$' is a tautology; the equivalence therefore reduces to:

$$\exists\, x.\mathsf{Imp}(i,x,o) \equiv \forall t.\, o((t{+}\delta){+}\delta) = i(t)$$

6. Simplifying the expression '$(t{+}\delta){+}\delta$', yields:

$$\exists\, x.\mathsf{Imp}(i,x,o) \equiv \forall t.\, o(t{+}2\delta) = i(t)$$

7. From which the desired correctness statement follows immediately:

$$\exists\, x.\mathsf{Imp}(i,x,o) \equiv \mathsf{Spec}(i,o)$$

This proof, although trivial in itself, illustrates the general approach to proving correctness statements that use structural abstraction. This commonly involves rewriting with the equations defining internal signals (step 3 above) and eliminating these equations by restricting the scope of the existential quantifiers (steps 4 and 5). For other examples of the use of this technique, see [1] or [4].

5 Formalising Behavioural Abstraction

Behavioural abstraction involves proving the correctness of designs with respect to *partial* specifications of intended behaviour, i.e. with respect to specifications that do not completely define the full range of behaviour that a device can exhibit, but only define its behaviour in environments or states that are of particular interest.

In logic, specifications are expressed by constraints on the values allowed on the external signals of a device; the 'range of behaviour' defined by a specification is given by the set of values that satisfy these constraints. A *partial* specification constrains a device's signals to have certain values in situations that are significant or relevant, but leaves unconstrained the signal values in all other situations. This means that, in the situations of 'undefined' behaviour, the predicate defining a partial specification will be satisfied by signal values that would not be allowed by a more complete specification of behaviour. Thus, the partial specification of a device imposes weaker constraints on its signal values than a complete specification would.

The formalisation of behavioural abstraction in logic is straightforward. Suppose that $\mathsf{Spec}(a,b)$ is the partial specification of a device and $\mathsf{Imp}(a,b)$ is the design description. The implementation Imp is correct with respect to the partial specification Spec if the following correctness relation holds:

$$\forall a\, b.\ \mathsf{Imp}(a,b) \supset \mathsf{Spec}(a,b)$$

This correctness statement asserts that whenever signals a and b satisfy Imp, they will also satisfy Spec. The implication may also be true, however, of signals for which Spec is true but Imp is false, i.e. there may be values that are allowed by the weaker constraints of Spec but not allowed by the constraints given by Imp. Thus, logical implication can be used to express the idea that the specification Spec is a behavioural abstraction, imposing weaker constraints on the signals a and b than those of the design description Imp. This use of logical implication for behavioural abstraction is natural and well known; see for example [1,7].

5.1 An Example

We now consider an example of the use of behavioural abstraction: the correctness proof of an RS latch, implemented using two cross coupled nor-gates. If a nor-gate with unit delay is specified by the three-place predicate Nor defined by:

$$\mathsf{Nor}(i_1, i_2, o) \equiv \forall t.\ o(t{+}1) = \neg(i_1\, t \vee i_2\, t)$$

then the implementation description of an RS latch is given by:

$$\mathsf{RS_imp}(s,r,q,\overline{q}) \equiv \mathsf{Nor}(r,\overline{q},q) \wedge \mathsf{Nor}(s,q,\overline{q})$$

The abstract specification of the latch will use the auxiliary predicates **store** and **stable**, defined as follows. The term '$\mathsf{store}\ s\ r\ t\ v$' will have the meaning 'the signals s and r take on values at time t such that the value v will be stored in the latch'. The formal definition of **store** is:

$$\mathsf{store}\ s\ r\ t\ v \equiv s(t){=}v\ \wedge\ s(t{+}1){=}v\ \wedge\ r(t){=}\neg v\ \wedge\ r(t{+}1){=}\neg v$$

This definition states, for example, that to store the value T in the latch the set and reset signals must have the values $s{=}\mathsf{T}$ and $r{=}\mathsf{F}$ for two units of time.

The term '$\mathsf{stable}\ sig\ v\ t\ n$' will have the meaning 'the signal sig is stable with value v during the period from time t to time $t{+}n$'. Formally, **stable** is defined as follows:

$$\mathsf{stable}\ sig\ v\ t\ n \equiv \forall t'.\ t{\le}t' \wedge t'{\le}t{+}n \supset sig(t'){=}v$$

Using **store** and **stable**, the partial specification of the RS latch is given by:

$$\begin{array}{l} \mathsf{RS_spec}(s,r,q,\overline{q}) \equiv \\ \quad \forall t\, v.\ \mathsf{store}\ s\ r\ t\ v \supset \\ \qquad\quad \forall n.\ \mathsf{stable}\ s\ \mathsf{F}\ (t{+}2)\ n \wedge \mathsf{stable}\ r\ \mathsf{F}\ (t{+}2)\ n \supset \\ \qquad\qquad\quad \mathsf{stable}\ q\ v\ (t{+}2)\ n \wedge \mathsf{stable}\ \overline{q}\ \neg v\ (t{+}2)\ n \end{array}$$

This abstract specification states that if the value v is stored in the latch at time t then the values v and $\neg v$ will be held stable on the outputs q and $\overline{q}$ from time $t{+}2$, for as long as both s and r are held low.

This specification defines the behaviour of the latch only for well behaved set and reset inputs, s and r. The predicate RS_spec does not state, for example, what the behaviour of the device will be if s and r are high simultaneously for some period of time. While s and r are both high, the antecedant 'store $s\ r\ t\ v$' will be false; the specification will therefore be satisfied during this period, regardless of the values of q and $\overline{q}$. Thus, the partial specification RS_spec leaves unspecified—i.e. unconstrained—the values of the outputs q and $\overline{q}$ during periods of time when both s and r are high.

The design description RS_imp, however, does *not* leave unconstrained the values of q and $\overline{q}$ during periods when both s and r are high; the values of these outputs are well defined at all times by the equations for the two nor-gates that implement the latch. To reflect this difference between RS_imp and RS_spec, the correctness statement is formulated as an implication:

$$\forall s\, r\, q\, \overline{q}.\ \mathsf{RS_imp}(s, r, q, \overline{q}) \supset \mathsf{RS_spec}(s, r, q, \overline{q})$$

That is, RS_spec is a behavioural abstraction of RS_imp that agrees with RS_imp on the values of q and $\overline{q}$ whenever the inputs s and r are 'well behaved'. The proof of this correctness statement is straightforward; the main step is an induction on the variable n in the definition of RS_spec.

This simple example, in which a partial specification defines the device's output signals for only well behaved input signals, is typical of correctness proofs involving behavioural abstraction, and shows how using logical implication to formulate correctness makes it possible relate such simplified specifications to more detailed design descriptions.

6 Formalising Data Abstraction

In the examples given above, values present on the ports of the devices have been modelled by booleans—i.e. values of type '*bool*'. Other logical types, however, can also be used to model the range of values on the ports of a device. For example, a type of 8-bit words, '*word8*', can be introduced into the logic and used to specify the behaviour of the serial-in parallel-out shift register shown below.

The behaviour of this device can be specified by a predicate Sreg(in, out), where in is a function of type '$num \to bool$' and out is a function of type '$num \to word8$'. Here, the logical type '*word8*' is used as an abstract data type in the specification of the shift register.

With such data types, correctness statements involving data abstraction can be formulated by using abstraction functions that map the data types of implementation descriptions to those of abstract specifications. There are two distinct tasks involved in using data abstraction in the correctness proof of a device:

- Constructing the logical types that will be used as abstract data types to describe the behaviour of the design and its specification.

- Defining abstraction mappings from the data types of the design description to those of the specification, and proving that the operations performed by the device on the detailed level data types correctly implement the specified operations on the abstract level data types.

The first of these will not be discussed in detail here. New logical types can be consistently introduced into the logic by constructing them from 'subsets' of existing types. The newly introduced types can then be characterised by a set of theorems—derived from their construction—that serve as 'axiomatizations' of the types. For details about introducing new types into the logic, see [2,3].

Once logical types have been defined for both the specification and the design description of a device, a correctness statement using data abstraction can be formulated. Consider, for example, a predicate $\mathsf{Imp}(a, b)$ describing the design of a device having values of type $ty1$ on its external ports. If Imp is a sequential device, the variables a and b will be signals of type '$num \to ty1$'. Suppose that the abstract specification for this device is given by a predicate $\mathsf{Spec}(a', b')$, where a' and b' are signals of type '$num \to ty2$'. To prove Imp correct with respect to Spec, we must define an appropriate abstraction function $\mathsf{f}: ty1 \to ty2$ to map values of type $ty1$ on the ports of Imp to values of type $ty2$ on the ports of Spec. Using f, the correctness statement that we want to prove is:

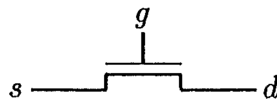
$$\forall a\, b.\ \mathsf{Imp}(a, b) \supset \mathsf{Spec}(\mathsf{f} \circ a,\ \mathsf{f} \circ b)$$

This states that if signals a and b satisfy Imp then the abstracted signals $\mathsf{f} \circ a$ and $\mathsf{f} \circ b$ will satisfy Spec. The function f maps values of type $ty1$ to values of type $ty2$; function composition with f simply translates the *sequences* of values of type $ty1$, given by a and b, to the corresponding sequences of values of type $ty2$.

In this simple example there is a one-to-one correspondence between the signals of the design description Imp and those of the specification Spec. In general, however, this may not be the case. An abstract signal of the specification of a device may be constructed from the values of several signals of its implementation description. An implementation of the Sreg device above, for example, could have eight boolean outputs that are mapped by an abstraction function to the single '$word8$' output of Sreg.

6.1 An Example

The following example illustrates the use of data abstraction in the correctness proof of a CMOS inverter. We first give the specifications of the components of the inverter, beginning with the N-type transistor:



which can be specified by the predicate Ntran, defined as follows:

$$\mathsf{Ntran}(g, s, d) \equiv \forall t.\ g(t){=}\mathsf{H} \lor (g(t){=}\mathsf{Z} \land g(t{-}1){=}\mathsf{H}) \supset s(t){=}d(t)$$

Here, we are using the three-valued type '*tri*', with values H (*high*), L (*low*) and Z (*floating*), to model the values present on the lines g, s and d. The predicate Ntran asserts that the values at the source s and the drain d will be equal whenever 1) the gate g is high or 2) the gate is floating and the gate was high at the previous moment of time. Thus, Ntran specifies a transistor that 'stores' the value on its gate for one unit of discrete time.[2]

The specification of a P-type transistor is similar to Ntran. It is:

$$\mathsf{Ptran}(g,s,d) \equiv \forall t.\, g(t){=}\mathsf{L} \lor (g(t){=}\mathsf{Z} \land g(t{-}1){=}\mathsf{L}) \supset s(t){=}d(t)$$

This specification asserts that the source and drain are connected if the gate is driven low or the gate is floating and was low at the previous instant of time.

Two other components are used in the inverter: Pwr and Gnd. These model the power and ground nodes respectively, and are defined by:

$$\mathsf{Pwr}(w) \equiv \forall t.\, w(t) = \mathsf{H} \qquad \text{and} \qquad \mathsf{Gnd}(w) \equiv \forall t.\, w(t) = \mathsf{L}$$

Using these parts, the standard design of an inverter can be described by the predicate Imp, defined as follows:

$$\mathsf{Imp}(i,o) \equiv \exists p\, g.\, \mathsf{Pwr}(p) \land \mathsf{Gnd}(g) \land \mathsf{Ptran}(i,p,o) \land \mathsf{Ntran}(i,g,o)$$

The predicate Imp defines the behaviour of the inverter in terms of values of type '*tri*'. Suppose, however, that we wish to show that this implementation satisfies a specification Inv, defined in terms of boolean values on the external ports:

$$\mathsf{Inv}(i,o) \equiv \forall t.\, o(t) = \neg i(t)$$

Here, the signals i and o have type '*num→bool*'; in the implementation, however, they have type '*num→tri*'. To abstract from three-valued to boolean signals, we will define a data abstraction function $\mathsf{f}: \mathit{tri} \to \mathit{bool}$ as follows:

$$\mathsf{f}(v) = \varepsilon b.\, (b \Rightarrow v{=}\mathsf{H} \mid v{=}\mathsf{L})$$

From this definition, it follows that $\mathsf{f}(\mathsf{H}) = \mathsf{T}$ and $\mathsf{f}(\mathsf{L}) = \mathsf{F}$. The use of the ε-operator makes the value of $\mathsf{f}(\mathsf{Z})$ be '$\varepsilon b.\mathsf{F}$', i.e. a fixed but unknown value of type '*bool*'.

Using f, we can formulate the correctness of Imp as follows:

$$\mathsf{Def}(i) \land \mathsf{Imp}(i,o) \supset \mathsf{Inv}(\mathsf{f} \circ i,\, \mathsf{f} \circ o)$$

This correctness statement asserts that if the condition Def holds of the input i then Imp is correct with respect to the specification Inv. The condition '$\mathsf{Def}(i)$' is a *validity condition* on the correctness of the inverter, defined by:

$$\mathsf{Def}(i) \equiv \forall t.\, i(t){=}\mathsf{H} \lor i(t){=}\mathsf{L}$$

The condition $\mathsf{Def}(i)$ ensures that the input i is never floating—which must be the case for the predicate Inv to represent a valid abstraction of the inverter's behaviour. If the inverter is used in an environment in which $\mathsf{Def}(i)$ is satisfied, we know that the abstraction is valid and we can use the simplified behaviour represented by Inv in place of the more detailed behaviour given by Imp. This can help to simplify the descriptions of designs containing inverters that are always strongly driven.

[2]This model of transistor behaviour is, of course, very much simplified—but it will serve for the purposes of this example. For a better transistor model see, for example, [11]

7 Formalising Temporal Abstraction

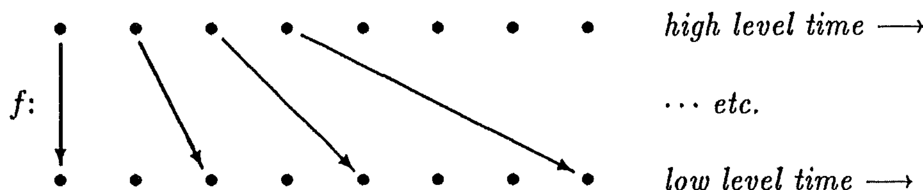
Temporal abstraction involves relating formal specifications that describe hardware behaviour using different notions of discrete time. This type of abstraction is used when a design description gives more detail about how a device behaves over time than is desired for its abstract specification. With the mechanism of temporal abstraction, information about a device's behaviour at moments of time that are not of interest can be hidden from the abstract specification, allowing the specification to concentrate on how the device behaves at significant or 'interesting' points of time.

In the simplest case of temporal abstraction, each single unit of discrete time at the abstract level of description corresponds to several units of time at the more detailed level of description. Here, the abstract specification of a device defines its behaviour at fewer points of time than the design description—i.e. the grain of discrete time is 'coarser' at the abstract level than at the detailed level.

To express this in logic, we must specify for each single *unit* or step of 'high level' time a corresponding *interval* or sequence of steps of 'low level' time. Suppose, for example, the unit of time from t' to $t'{+}1$ at the abstract level of description corresponds to the interval of time from t to $t{+}n$ at the more detailed level of description. In this case, high level time t' corresponds with low level time t and high level time $t'{+}1$ corresponds with low level time $t{+}n$. This correspondence can be specified formally by defining a mapping f from points of high level time to points of low level time such that:

$$f(t') = t \quad \text{and} \quad f(t'{+}1) = t{+}n$$

We can use such a mapping $f{:}num{\rightarrow}num$ to specify any correspondence between successive units of high level time and contiguous intervals of low level time:



Not every function of type $num{\rightarrow}num$, however, specifies a valid temporal abstraction; we require that the time mapping function be *increasing*. This ensures that if time t_2 comes after time t_1 at the abstract level, then this relationship also holds between the corresponding points of time at the detailed level. The requirement that a time mapping, f, be increasing can be expressed using the predicate Incr, defined as follows:

$$\mathsf{Incr}(f) \equiv \forall n\, m.\ (n > m) \supset (f(n) > f(m))$$

We now consider how an increasing temporal abstraction function, such as f above, can be used to formulate correctness statements that involve specifications at different grains of time. Suppose that $sig{:}num{\rightarrow}bool$ is a signal involved in the detailed level description of a design. The function sig gives detailed information about the sequence of values present on a port of the device. The device's abstract specification will involve a corresponding signal, $sig'{:}num{\rightarrow}bool$ say, that gives the port's value at just those

points of time that are significant at the abstract level. If f is an increasing function that specifies the correspondence between high and low level time, then the relationship that must hold between sig' and sig is:

$$\forall t.\ sig'(t) = sig(f(t)) \qquad \text{(i.e. } sig' = sig \circ f)$$

That is, at each instant of high level time the value of the abstract signal sig' must be the same as the value of the detailed signal sig at the corresponding instant of low level time.

Given that this correspondence must hold between the signals of an implementation description and those of an abstract specification, the required correctness relation is formulated as follows. Suppose that $\mathsf{Imp}(a, b)$ is the implementation description of a device, $\mathsf{Spec}(a, b)$ is its abstract specification, and f is a term that denotes a mapping of type $num \rightarrow num$. The specification Spec is related to the design description Imp by the temporal abstraction given by f if we can prove a correctness statement of the following form:

$$\mathsf{Incr}(\mathsf{f}) \wedge \forall a\, b.\ \mathsf{Imp}(a, b) \supset \mathsf{Spec}(a \circ \mathsf{f},\ b \circ \mathsf{f})$$

This states that whenever the signals a and b satisfy the temporally detailed specification Imp, then the abstracted signals $a \circ \mathsf{f}$ and $b \circ \mathsf{f}$ will satisfy the temporally abstract specification Spec.

The predicate Imp specifies the values allowed on the ports of the device at each instant of fine-grained time; the predicate Spec specifies the desired behaviour in terms of the values allowed on the ports at only some of these points of time. The function denoted by f defines the sequence of low level time points that lie between each unit of high level time. Proving this correctness statement involves showing that if a and b have the intermediate values allowed by Imp then the values of a and b at the abstracted time points will be allowed by Spec.

Notice that this correctness relation is formulated as an implication, rather than an equivalence. This reflects the idea that there may be several non-equivalent methods of implementing the device specified by Spec; the implementation in which the ports a and b take on the intermediate values defined by Imp is only *one* such method.

It is significant that this formulation of the temporal abstraction relation between Imp and Spec does not state that

- Spec is *a* temporal abstraction of Imp

but rather states that

- Spec is *the* temporal abstraction of Imp *given by* f

That is, the exact correspondence between time scales, given by the value of f, is an integral part of the statement of the correctness relation between Imp and Spec. There are several reasons why this information must be retained in the formulation of the correctness relation. One of these is to make it possible to 'compose' the correctness results that link several levels of structural and temporal abstraction, yielding one correctness

statement relating the lowest level design description to the highest level abstract specification.

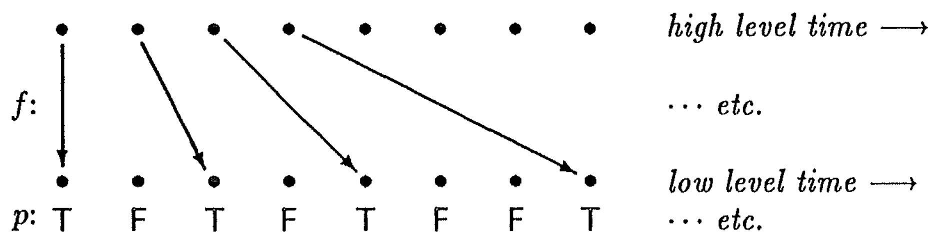
In the example given above, both signal a and signal b were abstracted using the same temporal abstraction function f. In general, however, it is not necessary that the same abstraction function be used for every signal of a device. For example, some signals may be abstracted at points of time corresponding to rising edges of a clock, while others are abstracted at points of time corresponding to falling edges. If, however, different abstraction functions are used for different signals of a device in the same correctness statement, there are certain reasonablity conditions that must be imposed on the combination of functions used, beyond simply requiring that each of them be increasing. These constraints are imposed order to prevent apparent 'computation' being done by the abstraction step itself—which is only supposed to remove or restructure information.

7.1 Constructing Temporal Abstraction Functions

The first step in the formulation and proof of a correctness relation involving temporal abstraction is to construct the appropriate abstraction function mapping high level time to low level time. This can often be done most easily by simply identifying those points of low level time that will correspond to points of high level time—i.e. by identifying those points of low level time that will be 'mapped to' by the temporal abstraction function.

The advantage of defining abstraction functions this way is that some of the signals of the implementation description can themselves be used to identify the points of low level time that will correspond to points of high level time. In synchronous systems, the appropriate points of low level time can often be identified by the value of the clock; in asynchronous systems, handshaking signals can be used for the same purpose.

To construct an abstraction function f from high level time to low level time, it is sufficient to define a function $p{:}num{\rightarrow}bool$ that is true of just those points of low level time that are to correspond to points of high level time:



From such a function p, it is possible to derive the abstraction function f as follows. If the term 'Timeof p n' denotes the point of low level time at which p is true for the nth time, then f can be defined by:

$$\forall t.\, f(t) = \mathsf{Timeof}\; p\; t \qquad \text{(i.e. } f = \mathsf{Timeof}\; p\text{)}$$

It remains to define formally the function Timeof.

7.1.1 Defining Timeof

We first observe that the value of the term 'Timeof p n', as informally described above, may be undefined for some values of p and n. If the signal p is not true at an infinite number of time points, then there will exist some N such that p is true only N times. For all $n > N$, there will not exist points of time at which p is true for the nth time and 'Timeof p n' will be undefined. Thus, the value of 'Timeof p' will be a partial function. Such partial functions, however, are not directly definable in the logic; all functions defined must be total. The definition of Timeof will therefore be based on a *relation*, Istimeof, which can be defined directly.

The term 'Istimeof p n t', will have the meaning 'p is true for the nth time at time t'. The formal definition of the relation Istimeof is done by primitive recursion on the variable n. In the case where n is zero, the definition is:

$$\mathsf{Istimeof}\, p\, 0\, t \equiv p(t)\ \wedge\ \forall t'.\, t'<t \supset \neg p(t')$$

That is, time t when p is true for the first (i.e. the 0th) time if p is true at time t and is false at all times before t.

For the case when when p is true for the $(n+1)$th time, the definition is:

$$\begin{array}{l}\mathsf{Istimeof}\, p\, (n+1)\, t \equiv \\ \quad p(t)\ \wedge\ \exists t'.\, t'<t\ \wedge\ \mathsf{Istimeof}\, p\, n\, t'\ \wedge\ \forall t''.\, (t'<t'' \wedge t''<t) \supset \neg p(t'')\end{array}$$

meaning that t is the time when p is true for the $(n+1)$th time if p is true at time t, and there exists a time t' before t when p is true for the nth time, and p is false at all times between t' and t.

This primitive recursive definition captures the idea that 'p is true for the nth time at time t'. There is no guarantee, however, that such times t exist for all values of p and n. In order to use this definition, it is necessary to show that if the signal p is true 'infinitely often' then for all n there is a unique time t at which p is true for the nth time.

The condition that p must be true at an infinite number of points can be formalised by the predicate Inf defined by:

$$\mathsf{Inf}(p) \equiv \forall t.\, \exists t'.\, t'>t \wedge p(t')$$

The predicate Inf can be used to express the fact that if p is true infinitely often then the time at which p is true for the nth time exists for all n:

$$\mathsf{Inf}(p) \supset \forall n.\, \exists t.\, \mathsf{Istimeof}\, p\, n\, t \qquad (1)$$

The proof of this statement proceeds by induction on n and use of the well ordering property of natural numbers.

It is also the case that Istimeof defines a *unique* time t for each value of n. Formally:

$$\forall t_1\, t_2.\, \mathsf{Istimeof}\, p\, n\, t_1 \wedge \mathsf{Istimeof}\, p\, n\, t_2 \supset (t_1 = t_2) \qquad (2)$$

Given these theorems, the relation Istimeof can now be used to define the function Timeof. Using the ε-operator, described in Section 3.1, the definition of Timeof is given by the following equation.

$$\forall p\, n.\ \mathsf{Timeof}\ p\, n = \varepsilon\, t.\ (\mathsf{Istimeof}\ p\, n\, t)$$

That is, 'Timeof $p\,n$' denotes some time, say t, such that 'Istimeof $p\,n\,t$' is true or, if no such time exists, 'Timeof $p\,n$' denotes an arbitrary natural number. With the use of the ε-operator, this definition makes the term 'Timeof p' always denote a total function; 'Timeof $p\,n$' is defined for all values of n, even when the signal p is true at only a finite number of points.

If, however, the signal p is true infinitely often, then for all n there will exist a time t such that 'Istimeof $p\,n\,t$' is true, and this time will be unique. Thus, if $\mathsf{Inf}(p)$ holds, 'Timeof $p\,n$' will in fact denote the unique time at which p is true for the nth time, as desired. More formally, an immediate consequence of (1) is:

$$\mathsf{Inf}(p) \supset \mathsf{Istimeof}\ p\, n\ (\mathsf{Timeof}\ p\, n)$$

which gives the following easily-proved lemma:

$$\mathsf{Inf}(p) \supset p(\mathsf{Timeof}\ p\, n)$$

Using (2) the following two lemmas can also be proven:

$$\mathsf{Inf}(p) \supset (\mathsf{Timeof}\ p\, n) < (\mathsf{Timeof}\ p\, (n+1))$$

$$\mathsf{Inf}(p) \supset \forall t.\ (\mathsf{Timeof}\ p\, n) < t \wedge t < (\mathsf{Timeof}\ p\, (n+1)) \supset \neg p(t)$$

These lemmas can be conveniently collected together using the predicate Next defined by:

$$\mathsf{Next}\ t_1\ t_2\ sig \equiv t_1 < t_2 \wedge sig(t_2) \wedge \forall t.\ (t_1 < t \wedge t < t_2) \supset \neg sig(t)$$

giving the following theorem

$$\mathsf{Inf}(p) \supset \forall n.\ \mathsf{Next}\ (\mathsf{Timeof}\ p\, n)\ (\mathsf{Timeof}\ p\, (n+1))\ p \tag{3}$$

This theorem states that if p is true infinitely often then 'Timeof p' is well defined and denotes the desired function from high level time points to low level time points.

7.1.2 Using Timeof to Define Abstractions

Having formally defined the function Timeof and shown that it is well defined for functions p that are true at an infinite number of points, it is possible to use Timeof to define temporal abstraction functions for statements of correctness.

A temporal abstraction function that maps high level time to low level time is just an increasing function $f{:}num{\rightarrow}num$. Any such function can be defined using Timeof and an appropriate function p that indicates the points of low level time that are abstracted to points of high level time.

Formally, we have that:

$$\forall f.\ \mathsf{Incr}(f) \equiv \exists p.\ \mathsf{Inf}(p) \wedge f = \mathsf{Timeof}\ p$$

This theorem follows from the definition of Incr and the properties of Timeof given by theorem (3).

With this theorem in mind, we can use Timeof to construct an alternative formulation of temporal abstraction as follows. If p is a term that denotes a function indicating the points of low level time that are to be abstracted to points of high level time, then the correctness relation that must hold between a specification Spec and an implementation Imp is:

$$\mathsf{Inf}(\mathsf{p}) \wedge \forall a\, b.\ \mathsf{Imp}(a, b) \supset \mathsf{Spec}(a \circ (\mathsf{Timeof}\ \mathsf{p}),\ b \circ (\mathsf{Timeof}\ \mathsf{p})) \tag{4}$$

This states that Imp is correct with respect to the temporally abstract specification Spec if whenever a and b satisfy Imp then the abstract signals constructed by sampling a and b when p is true will satisfy Spec. If 'when' is an infix operator defined as follows:[3]

$$\forall s\, t.\ s \ \mathsf{when}\ t = s \circ (\mathsf{Timeof}\ t)$$

then this correctness statement can be written:

$$\mathsf{Inf}(\mathsf{p}) \wedge \forall a\, b.\ \mathsf{Imp}(a, b) \supset \mathsf{Spec}(a\ \mathsf{when}\ \mathsf{p},\ b\ \mathsf{when}\ \mathsf{p})$$

Since $\forall p.\ \mathsf{Inf}(p) \supset \mathsf{Incr}(\mathsf{Timeof}\ p)$, the correctness statement (4) implies a correctness statement of the old form:

$$\mathsf{Incr}(\mathsf{f}) \wedge \forall a\, b.\ \mathsf{Imp}(a, b) \supset \mathsf{Spec}(a \circ \mathsf{f},\ b \circ \mathsf{f})$$

with f=Timeof p. Furthermore, since every increasing function f can be expressed using 'Timeof p' with an appropriate function p, this alternative form of correctness relation for temporal abstraction is essentially equivalent to the original one.

The following example shows how Timeof can be used in the correctness proof of a unit delay register, and describes some ways in which the ideal form of correctness given above must be modified in practice.

7.2 An Example

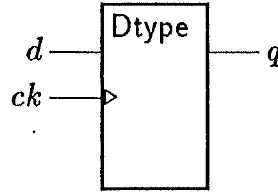
A commonly used register-transfer level device is the unit delay:

which can by specified in higher order logic by:

$$\mathsf{Del}(in, out) \equiv \forall t.\ out(t{+}1) = in(t)$$

[3] In previous work, I have called this operator 'Abs'. The mnemonicly superior name 'when' was suggested by the work of Halbwachs, Lonchampt and Pilaud [5].

The unit delay device is a temporal abstraction; its implementation is specified at a more detailed grain of time. One possible implementation is the rising edge triggered D-type flip flop:



which, ignoring detailed timing information,[4] can be specified by the predicate Dtype defined as follows:

$$\mathsf{Dtype}(ck, d, q) \equiv \forall t.\, q(t{+}1) = (\mathsf{Rise}\; ck\; t \Rightarrow d(t) \mid q(t))$$

where the term 'Rise $ck\; t$' has the meaning 'the clock, ck, rises at time t':

$$\mathsf{Rise}\; ck\; t \equiv \neg ck(t) \wedge ck(t{+}1)$$

We want to show that the D-type flip flop is a correct implementation of the register-transfer level unit delay. Using the formulation of correctness for temporal abstraction described above, we must prove a correctness statement of the following form:

$$\mathsf{Inf}(\mathsf{p}) \wedge \forall ck\; d\; q.\, \mathsf{Dtype}(ck, d, q) \supset \mathsf{Del}(d\, \mathsf{when}\, \mathsf{p},\; q\, \mathsf{when}\, \mathsf{p})$$

where p is a term that denotes a function of type '*num→bool*'. It is not possible, however, to prove a correctness statement of this form; for we can show that:

$$\neg\exists p.\, \mathsf{Inf}(p) \wedge \forall ck\; d\; q.\, \mathsf{Dtype}(ck, d, q) \supset \mathsf{Del}(d\, \mathsf{when}\, p,\; q\, \mathsf{when}\, p)$$

The trouble is that the abstract time scale for Del must be defined in terms of the value of the clock, ck. Informally, Dtype implements a unit delay by sampling its input when the clock rises and holding this value on the output until the next rise of the clock. In this way the D-type delays by one clock period the *sequence* consisting of the values of the signal d at the times of successive clock rises.

This suggests that the temporal abstraction function for this proof should map successive points of high level time to the successive points of low level time at which the clock rises. Using 'Rise ck', the required temporal abstraction function is given, for each value of ck, by the term 'Timeof(Rise ck)'. Using this, we can reformulate the correctness statement for the Dtype implementation of Del as:

$$\begin{aligned}&\forall ck.\, \mathsf{Inf}(\mathsf{Rise}\; ck) \wedge \\ &\quad \forall d\; q.\, \mathsf{Dtype}(ck, d, q) \supset \mathsf{Del}(d\, \mathsf{when}\, (\mathsf{Rise}\; ck),\; q\, \mathsf{when}\, (\mathsf{Rise}\; ck))\end{aligned}$$

[4]For clarity, a greatly simplified specification of Dtype is used in this example. For D-type flip flop specifications that include information about detailed timing behaviour see [6] or [8].

This states that the clock rises infinitely often, and whenever the signals ck, d and q are in the Dtype relationship, the abstracted signals—i.e. the signals d and q sampled at successive rises of the clock—will be in the Del relationship. Thus there is a 'family' of sampling functions defined by 'Rise ck', giving a temporal abstraction from Dtype to Del for each value of the clock ck.

Because we are not interested in constructing temporal abstraction functions for signals that do not satisfy $\mathsf{Dtype}(ck, d, q)$, this correctness statement can be further modified to have the following form:

$$\begin{aligned} &\forall ck\, d\, q.\, \mathsf{Dtype}(ck, d, q) \supset \\ &\qquad \mathsf{Inf}(\mathsf{Rise}\; ck) \wedge \mathsf{Del}(d\, \mathsf{when}\, (\mathsf{Rise}\; ck),\; q\, \mathsf{when}\, (\mathsf{Rise}\; ck)) \end{aligned}$$

As it stands, however, this correctness statement is false; for Dtype does not constrain the value of ck, and it is not the case that $\mathsf{Inf}(\mathsf{Rise}\; ck)$ holds for all possible values of ck. The correctness statement will therefore be modified to have $\mathsf{Inf}(\mathsf{Rise}\; ck)$ as an assumption, giving:

$$\begin{aligned} &\forall ck.\, \mathsf{Inf}(\mathsf{Rise}\; ck) \supset \\ &\qquad \forall d\, q.\, \mathsf{Dtype}(ck, d, q) \supset \mathsf{Del}(d\, \mathsf{when}\, (\mathsf{Rise}\; ck),\; q\, \mathsf{when}\, (\mathsf{Rise}\; ck)) \end{aligned} \tag{5}$$

which is the final form of the correctness statement for this device.

The proof of (5) is straightforward. The main step is an induction on the number of time steps between adjacent rises of the clock, showing that the value of d that is sampled at a rise of ck is held on q until the next rise. The correctness statement follows easily from this result and the properties of Timeof given by (3).

The assumption that the clock rises infinitely often is a *validity condition* on the temporal abstraction; the correctness statement asserts that Del presents a valid abstract view of Dtype provided the condition '$\mathsf{Inf}(\mathsf{Rise}\, ck)$' is satisfied. To use the Del abstraction, this condition on the clock must be met by the environment in which the Dtype is placed. The validity condition '$\mathsf{Inf}(\mathsf{Rise}\; ck)$' is as unrestrictive as possible, given the simple D-type model used. The clock ck is not required to be regular or to have a minimum clock period; the assumed liveness condition is sufficient for the proof to go through.

This proof gives a very simple example of the most common type of temporal abstraction, where contiguous intervals of low level time correspond to successive units of high level time. Examples involving detailed timing information or several different temporal abstractions in the same proof are more complex but involve the same general approach as this simple example.

There are other types of temporal abstraction that can be used to formulate correctness statements, but that can not be constructed using Timeof. In one form of temporal abstraction, for instance, the occurrence of an event at any time during an interval of low level time corresponds to a signal being true at a point of high level time. This type of abstraction can be used to deal with asynchronous inputs to synchronous systems, hiding from the system's abstract specification irrelevant information about the exact timing of the asynchronous input signals.

8 Conclusion

In hardware verification, designs that are proven correct ought to have formal descriptions that model reality reasonably well. The more accurate the model of device behaviour, the less likely it is for design errors to escape discovery by formal verification. Formal specifications, on the other hand, must be clear and concise, in order to be seen to reflect the designer's intent. This means that most of the details of a device's behaviour must be left out of its formal specification; only the essentials can be included. The idea of abstraction is therefore fundamental to a formalisation of hardware design correctness.

This paper has shown how four basic abstraction mechanisms for hardware verification can be formalised in higher order logic and used to control the size and complexity of device specifications. For clarity, the examples given were kept very simple; each example involved only one simple abstraction step. More substantial examples have been done, in which a series of nested abstractions is used to relate the bottom level design description to the top level abstract specification.

The role of abstraction in hardware verification is not restricted to that of complexity control. Structuring a correctness proof to span several levels of abstraction allows it to serve as structured *documentation* for a design. The top level specification gives a concise description of the essential features of the device; if more information about its operation is required, this is provided by the specifications at the next lower level of abstraction. If still more information is desired, it can be obtained from the next level of abstraction, and so on, right down to the level of primitive components. The abstraction mechanisms that are used in the proof serve to explain how each level of abstraction is related to the next lower level.

Abstractions often involve *validity conditions* that state when a specification represents a valid abstract view of a more detailed design description. Formal correctness proofs make such abstraction validity conditions explicit; for the proofs to go through, they must either be satisfied or included as assumptions. This means that a formal proof clearly documents validity assumptions that might otherwise lead to misunderstandings by remaining implicit. The conditions under which a device's abstract specification reflects the actual behaviour of the device will be clear from the assumptions present in its proof of correctness.

Validity conditions that arise in the correctness proofs of some of the components of a device can sometimes help to simplify the task of writing specifications for other components of the device. For example, the validity condition '$\mathsf{Inf}(\mathsf{Rise}\, ck)$' generated in the Dtype proof simplifies the task of specifying the desired behaviour of clock generator circuits for use with the Dtype device. We know that the clock signal must satisfy this condition; we can therefore take the abstract specification of the clock generator to be just the validity condition itself.

Some abstractions can be seen as specifications of the 'protocol' for communication at the external ports of a device. Consider, for instance, the temporal abstraction given for the D-type flip flop implementation of the unit delay. From the correctness statement, it is clear that values are input to the delay device only at rising edges of the clock. The

abstract specification of this device states that the value input to the device is delayed by one unit of time; the temporal abstraction used in the correctness statement explains what it *means* for a value to be 'input to the device'.

Abstraction mechanisms similar to the ones discussed in this paper are likely to play an important role in the automatic *synthesis* of designs. In this paper, abstraction mechanisms were described as *removing* information from design descriptions to yield abstract specifications. Synthesis of designs involves the opposite process—adding detail to formal specifications until an implementable design description is obtained. The abstraction mechanisms commonly used in post-design verification may also suggest heuristics for design synthesis from formal specifications.

Acknowledgements

Thanks are due to the members of the Cambridge Hardware Verification Group, especially Albert Camilleri, Francisco Corella, Inder Dhingra, Mike Gordon and Jeff Joyce, who made valuable comments on drafts of this paper. I would also like to thank Graham Birtwistle of the University of Calgary for his suggestions. This research is funded by the Royal Commission for the Exhibition of 1851.

References

[1] Camilleri, A.J., Gordon, M.J.C. and Melham, T.F., 'Hardware Verification Using Higher Order Logic,' in: *From HDL Descriptions to Guaranteed Correct Circuit Designs*, Proceedings of the IFIP WG 10.2 International Conference, Grenoble, France, 9-11 September, 1986, Dominique Borrione, ed., North-Holland, Amsterdam, 1987.

[2] Gordon, M.J.C., 'HOL: A Machine Oriented Formulation of Higher Order Logic,' Technical Report No. 68, Computer Laboratory, University of Cambridge, 1985.

[3] Gordon, M.J.C., 'HOL: A Proof Generating System for Higher Order Logic,' to appear in: *VLSI Specification, Verification and Synthesis*, Proceedings of the Workshop on Hardware Verification, Calgary, 12-16 January, 1987, G.M. Birtwistle and P.A. Subrahmanyam, eds., 1987.

[4] Gordon, M.J.C., 'Why Higher Order Logic is a Good Formalism for Specifying and Verifying Hardware,' in: *Formal Aspects of VLSI Design*, Proceedings of the 1985 Edinburgh Conference on VLSI, G.J. Milne and P.A. Subrahmanyam, eds., North-Holland, Amsterdam, 1986.

[5] Halbwachs, N., Lonchampt, A. and Pilaud, D., 'Describing and Designing Circuits by means of a Synchronous Declarative Language,' in: *From HDL Descriptions to Guaranteed Correct Circuit Designs*, Proceedings of the IFIP WG 10.2 International Conference, Grenoble, France, 9-11 September, 1986, Dominique Borrione, ed., North-Holland, Amsterdam, 1987.

[6] Hanna, F.K. and Daeche, N., 'Specification and Verification using Higher-Order Logic: A Case Study,' in: *Formal Aspects of VLSI Design*, Proceedings of the 1985 Edinburgh Conference on VLSI, G.J. Milne and P.A. Subrahmanyam, eds., North-Holland, Amsterdam, 1986.

[7] Hanna, F.K. and Daeche, N., 'Specification and verification of digital systems using higher-order predicate logic,' *IEE Proceedings-E*, Vol. 133, Part E, No. 5, September 1986, pp. 242-254.

[8] Herbert, J.M.J., Ph.D. Thesis, The University of Cambridge, to appear in 1987.

[9] Hoare, C.A.R., 'A Calculus of Total Correctness for Communicating Processes,' *Science of Computer Programming*, Vol. 1, No. 1, 1981.

[10] Leisenring, A., *Mathematical Logic and Hilbert's ε-Symbol*, Macdonald and Co. Ltd., London, 1969.

[11] Winskel, G., 'Lectures on models and logic of MOS circuits,' Proceedings of the Marktoberdorf International Summer School on Logic Programming and Calculi of Discrete Design, July 1986.