SPECIFICATION AND VERIFICATION USING HIGHER-ORDER LOGIC: A CASE STUDY

F.K. Hanna and N. Daeche

Rapporteur: Mr. A.M. Koelmans

Specification and Verification using Higher-Order Logic: A Case Study

F K Hanna and N Daeche

University of Kent
Canterbury, Kent, U.K.

Higher-order logic is an ideal computer hardware description language: it allows the behaviour of digital devices to be rigorously specified, and it allows the correctness of digital system designs to be formally verified. The VERITAS approach to specification and verification is based on the use of higher-order logic. As a case study of this approach, a behavioural specification for an edge-triggered D-type flipflop is formulated, and a description is given of the stages involved in computationally verifying the correctness of a commonly used implementation of this kind of flipflop.

1. Introduction

Contemporary computer hardware design languages (CHDL's) have been designed to achieve many aims, including the ability to:

- describe the *structure* of digital systems (possibly hierarchically),
- describe the *behaviour* of digital systems (possibly at different levels of abstraction),
- *simulate* the behaviour of assemblages of digital components,
- *transform* structural or behavioural descriptions into other forms (for example, into wiring lists, or into waveform diagrams, etc),
- generate *fault diagnostic* information, etc.

One aim, however, which by and large they have *not* set out to achieve is that of allowing *reasoning* about the properties of structures, the properties of behaviours, or of the interrelationship between structures and behaviours. Alternatively stated, little emphasis has been given to the definition of the *semantics* of CHDL's.

In this respect, CHDL's are similar to many existing programming languages (for example, FORTRAN, LISP, ADA, etc). These languages, although they are seen as being well suited for expressing complex sequences of computation, have semantics which (to the extent that they have been defined at all !) are of immense intricacy. For example, a recent attempt to define the semantics of ADA ran to some several hundred pages...

On the other hand, some recent programming languages (OCCAM being a prime example) do have a well defined, and relatively simple, semantics. This comes about not by accident, but because the design of such languages *started* with semantic considerations, with other aspects of the languages (eg, syntax, implementation, etc) being attended to only *after* the semantics had been rigorously defined. The end result is that it is relatively simple to reason (either intuitively, or formally) about the properties (correctness, termination, equivalence, etc) of programs written in such languages.

We believe that a similar approach can usefully be adopted towards the design of CHDL's. The clarity, simplicity and rigour of definition of the *semantics* of a CHDL (rather than, for example, its syntax, or its built-in timing primitives, etc) should be *the prime consideration* influencing its design.

Once this point of view has been adopted, the question naturally arises as to what notation or formalism is best suited for expressing the semantics of a CHDL. It is not difficult to list the relevant criteria:

- Ideally, the formalism should *already exist*. This means that it will have been well researched, (independently of our intended use for it) and there will be an existing community of 'experts' from whom help may be sought in resolving any tricky technical points.
- The formalism should be both *powerful* and *concise*. In fact, if we intend to use it to reason about arbitrary properties of complex hardware designs, it should be powerful enough to encompass a *large part of mathematics itself.*
- At the same time, the formalism should be not too far removed from the informal modes of expression and intuitive reasoning that digital engineers use most of the time.
- It should allow *partial* descriptions to be expressed and reasoned about. The real world is both indeterminate and of unbounded complexity; our knowledge of it is therefore *inherently* partial. Thus, any *total* description of it must necessarily be an incorrect one.
- Finally, the formalism should be a strictly *formal* one. This implies that reasoning in the formalism can be carried out by *formal inferencing* (or, alternatively expressed, by symbol manipulation). In turn, this implies that the logical soundness of any reasoning carried out in the formalism can be *computationally checked.*

Taken together, this list of criteria is almost prescriptive in suggesting *typed, higher-order predicate logic* [11] as the formalism of choice. As a brief illustration of why *higher*-order logic, rather than just *first*-order logic is appropriate, consider the following sequence of concepts:

- The *voltage* at a port. We can consider a voltage to be a *first-order* entity.
- The *waveform* at a port. A waveform is a function from time to voltage, and hence it is a *second-order* entity.
- The *behavioural specification* of a device. As we shall discuss in §2.3 below, a behavioural specification is a predicate on a tuple of waveforms. It is, therefore a *third-order* entity.
- The relation '*better than*', meaning that one behavioural specification is stronger (ie, more

restrictive) than another. This is a predicate on pairs of behavioural specifications, and hence is a *fourth-order* entity.

- The property '*is a transitive relation*' (a property that we might wish to assert the above relation '*better than*' possesses). This property is a predicate on the above fourth-order relation, and hence it itself is a *fifth-order* entity.

Thus, (and this is not a particularly 'contrived' example) we find that a fairly intuitive property — the transitivity of the above relation — is a fifth-order one. By using higher-order logic we gain the ability to freely introduce and manipulate higher-order entities. In turn, this leads to an easy and natural correspondence between intuitive, everyday concepts and their precise, formal expression within the CHDL.

The reason why a 'typed' species of logic is used is twofold. Firstly, the (informal) notations of science and engineering tend to be typed (eg, *let w be a waveform, t be an instant of time*, etc) and thus typed logic provides a more concise and natural means of expression than does untyped logic. Secondly, like the untyped λ-calculus, 'ordinary' higher-order logic suffers from being inconsistent; the introduction of types is one way of overcoming this defect.

Relation to other formalisms Another feature we might mention in favour of the use of typed higher-order logic is that it includes, as proper subsets, two other formalisms which are also currently being explored as a basis for reasoning about the properties of digital systems. These are:

- The 'functional' subset (essentially Church's typed λ-calculus). Terms in this subset can be 'evaluated', and thus behavioural specifications restricted to this subset can be 'animated'.
- The 'first-order, Horn clause' subset (essentially PROLOG). The truth values of (some) terms expressed in this subset can be mechanically determined, and thus questions posed about properties of behavioural specifications restricted to this subset can be automatically answered.

1.1 Background

We have given the name VERITAS to the general approach to specification and verification of digital systems based on the use of predicate logic. We also use this name variously to describe the overall methodology, the project under which this approach has been developed, and the particular species of logic used. Work on the VERITAS project started in 1981, with the publication [2] of a set of theory presentations characterising, within *first*-order logic, the structural and behavioural properties of discrete logic devices, and the first computationally checked proofs were obtained in 1982. The use of *higher*-order logic for this purpose was first proposed in [3].

Relation to other work The use of logic to describe properties of the physical world and to reason about their interrelation finds its origins with Euclid's informal axiomatisation of geometry in ancient times, and with Hilbert's more formal axiomatisation of the same at the beginning of this century. Since then, propositional logic (Boolean algebra) has underpinned much of the development of digital logic and computers, and, over the last couple of decades, both first-order logic and the λ-calculus have found extensive use in software verification.

Within the field of hardware verification, approaches relating to the one described here have been reported by several authors. Eveking [8] has advocated the use of first-order logic and has developed an approach based on the use of predicate transformers, and has discussed the way that descriptions at different levels of abstraction may be related. The use of higher-order logic (with an emphasis towards specification and verification at higher levels of abstraction than those discussed here) is described [7] by Gordon, and an associated case study by Herbert is presented in [9]. Finally, we mention that the use of predicate logic to reason about circuits at the analogue level is described [6] by Shostak.

1.2 Contents

The main aim of this paper is to present a case study, of moderate complexity, demonstrating the VERITAS approach to the behavioural specification of a digital device, and to the formal verification of a proposed implementation of it. The remainder of this paper is organised as follows:

- We begin with a discussion of our approach to *partially describing* waveforms, and introduce the concept of a '*wavespec*' — that is, a waveform specification.
- We illustrate (as a simple example) how, using higher-order logic, the behaviour deemed to characterise a NAND gate can be specified.
- As a more complex example, we specify the behaviour of an edge-triggered, D-type flipflop (we abbreviate the name of this device to 'D-flipflop').
- We describe a proposed implementation, in terms of NAND gates, of a D-flipflop, and *attempt* to describe, at an intuitive level, how it works. This turns out not to be easy ...
- We introduce the notions of *formal verification*, and of *computational theorem proving*.
- As a case study in formal verification, we describe the various stages involved in computationally constructing a formal proof asserting the correctness of the above proposed implementation of the D-flipflop.
- Finally, we indicate the practical significance of the above result, and summarise our main conclusions.

Although this paper is largely self-contained, it does not attempt to cover all aspects of the subject. A different perspective on the use of higher-order logic for specification and verification will be found in [4], with an emphasis on the formal specification of structures (ie, circuit diagrams). In addition, in [5] can be found a full and detailed account of the formal logic used here and a description of the VERITAS Theorem Prover, together with a tutorial exposition on the latter.

1.3 Notation

The mathematical notation used in this account is, with one or two minor exceptions, one of the notations used by the computational implementation of the system (as described later). There are only three features that need comment:

- The *application* of a function f to a set of arguments $x_1, x_2, \ldots, x_n$ is denoted (in the 'applicative' style) by $(f \quad x_1\ x_2 \ldots x_n)$ rather than by $f(x_1, x_2, \ldots, x_n)$.
- It is often convenient to represent multi-argument functions in higher-order (or 'curried') form; for example $((F\ x_1)\ x_2)$ instead of $(f\ x_1\ x_2)$.
- The relative indentation of multi-line expressions is used, along with parentheses, to indicate grouping.

2. General Approach

The methodological aspects of the VERITAS approach are a little more fully described in [4]; here we have space to present only a very brief and superficial summary. Overall, the approach involves defining a *theory* which characterises the universe of discourse of digital engineering. A theory is defined by a *signature*; this is simply a list of the *symbols* that can occur in terms and the *axioms* that can be used in constructing *derivations* (or 'proofs') of theorems within the theory. The intention is that:

- Each symbol should correspond to some entity in the universe of discourse.
- Each axiom should describe some relation between these entities that is known (or asserted) to be true.

In practice, it is convenient to define a signature in a structured way, as a series of *extensions* of a primitive signature. This therefore gives rise to a corresponding nested series of theories. In the following subsections, we summarise the principal features of each of these theories and indicate their intent. The most primitive theory in the sequence is that of pure higher-order logic. The symbols introduced include the type *bool* (comprising the two propositional truth values *true* and *false*) together with the logical connectives ($\wedge$, $\vee$, $\neg$, etc) and equality (ie, '='). The axioms introduced define certain fundamental properties of higher-order logic (for instance, reflexivity of equality, extensionality, etc).

2.1 Theory of Numbers and of Time

The first important extension of the signature of pure higher-order logic defines the theory of Natural Numbers, introducing the type *nat* (of natural numbers), various arithmetic operators (*succ*, +, −, etc), and a set of axioms that characterise number theory. As an example, the following axiom describes induction:

$$\begin{array}{l} \vdash \forall P : nat \rightarrow bool. \\ \quad (P \quad 0) \\ \rightarrow \\ \quad\quad \forall n : nat. (P \quad n) \rightarrow (P \quad (succ \quad n)) \\ \quad \rightarrow \\ \quad\quad \forall n : nat. (P \quad n) \end{array}$$

(Note that, since this axiom involves quantification over the set of all properties P, it is an intrinsically higher-order one.)

One feature of this theory to which we need to draw attention is that the operator '$-$' denotes *natural subtraction* (that is, $a - b = 0$ if $b \geq a$). This operator shares only some of the properties of 'ordinary' subtraction (as commonly defined on the integers). As a simple example, consider the relation $(a + b) - c = (a - c) + b$ Whilst this relation is true if $b = 0$ or if $a \geq c$, it is *not* true in general.

The next extension of the signature defines a theory of time. Specifically, it introduces three data types

time	— describing *instants* of time
dur	— describing non-negative *durations* of time
intvl	— describing *intervals* of time

and a collection of functions between them. In particular, the function

$$I : \ time \times dur \rightarrow intvl$$

is used for constructing intervals. The interval $(I \quad t \quad d)$ is defined as the semi-open interval starting at t, and of duration d.

2.2 Theory of Waveforms

The behaviour of a digital system is defined in terms of the waveforms present at its ports. Although these are often idealised as being perfect, two-valued 'binary' waveforms, in reality they are *analogue* waveforms that are only in a 'binary' state during certain intervals of time. The general approach followed by the VERITAS axiomatisation is not to pretend that reality is other than it is, but simply to axiomatise *only* those properties of it that we actually require.

In the theory of Waveforms, defined by the next extension of the signature, two types are introduced; these are wf the set of *analogue* waveforms, and C, a type containing the two elements c_f and c_t, representing the two 'digital' constants. (Note: these digital constants, (pseudonyms *0*, *1*, or *low*, *high*, etc) should not be confused in any way with the propositional truth values $true$, $false$ of type *bool* introduced above).

Wavespecs One of the foundations upon which the entire VERITAS approach to specifying behaviours rests is in the use of a particular predicate

$$const : (intvl \times C) \rightarrow (wf \rightarrow bool)$$

This predicate (the fact that it is 'curried', or higher-order, is unimportant) is defined by the term

$$((const \quad i \quad c) \quad w)$$

meaning that throughout an interval of time i, the analogue waveform w assumes the digital value c. This is sketched in Fig. 1a. Note the *partial* nature of *const*, and in particular, the fact that it makes no assertion whatsoever concerning the behaviour of w outside the interval i. It is convenient to introduce a graphical notation, as shown in Fig. 1b, for expressing assertions of this general form.

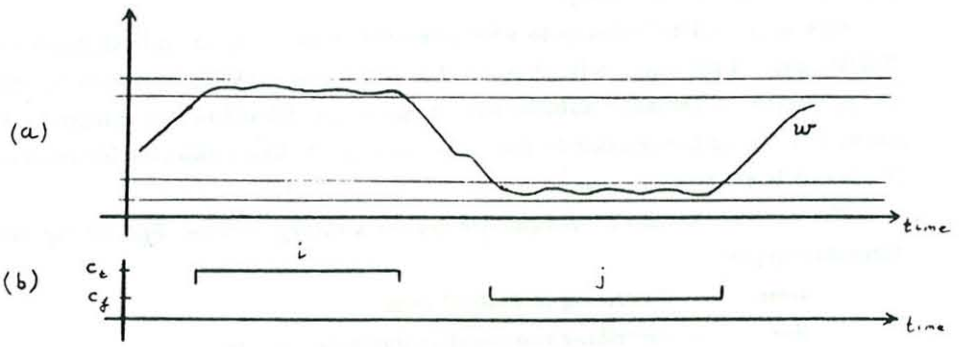

Fig 1.

(a) Both the assertion $((const \quad i \quad c_t) \quad w)$ and the assertion $((const \quad j \quad c_f) \quad w)$ are true.

(b) This shows the convention adopted for depicting these assertions graphically.

We call a predicate defined on a waveform (or, more generally, on a collection of waveforms) a *wavespec* (for 'waveform specification'). For example, the term $(const \quad i \quad c_t)$ is a wavespec. As we shall see, wavespecs are of fundamental importance in the VERITAS approach.

2.3 Theory of Gate Behaviours

The theory of Gate Behaviours (defined by a further extension of the above signature) contains the definitions for the *behavioural* (as distinct from the *structural*) characteristics implied by common usage of the terms 'AND gate', 'OR gate', etc. Insofar as these are pure *definitions*, the reader is free to decide the extent to which they succeed in capturing what is generally meant by these terms.

We will illustrate the general approach adopted for characterising device behaviours by taking the behaviour of a 2-input NAND gate as an example: other gate behaviours follow a similar pattern, as do other device behaviours (e.g. RAM's, ALU's, etc).

We begin by noting that, as a NAND gate has three ports, the general form of the definition will be a predicate on three waveforms (ie, a wavespec on a 3-tuple of waveforms). Because NAND gates have differing 'speeds', the definition must also incorporate some *timing parameters*. We choose to use a tuple of four durations to characterise the speed; this allows a much 'tighter' definition than is normally used for specifying the characteristics of gates. As an aside, we note that there is, of course, no reason why a 'looser' definition (ie, one with fewer parameters) should not be used if so desired, but then, because less would be known about the gate's behaviour, it would not be possible to infer so much about the properties of systems employing such gates.

The symbol we use for characterising the behaviour of a NAND gate is

$$NAND\_BEHAV : dur^4 \rightarrow (wf^3 \rightarrow bool)$$

As before, it turns out that it is convenient to express it as a *higher-order* function. The meaning we wish the term

$$((NAND\_BEHAV \quad d_1 \quad d_2 \quad d_3 \quad d_4) \quad w_{in_1} \quad w_{in_2} \quad w_{out})$$

to have is (approximately):

"The three waveforms w_{in_1}, w_{in_2} and w_{out} exhibit the behavioural characteristics of a NAND gate, with propagation and hold times (for *low* inputs) of d_1 and d_2 respectively, and propagation and hold times (for *high* inputs) of d_3 and d_4 respectively."

Let us now propose a definition for this predicate. We will first illustrate its intent in terms of waveform sketches, and then evolve the corresponding formal definition.

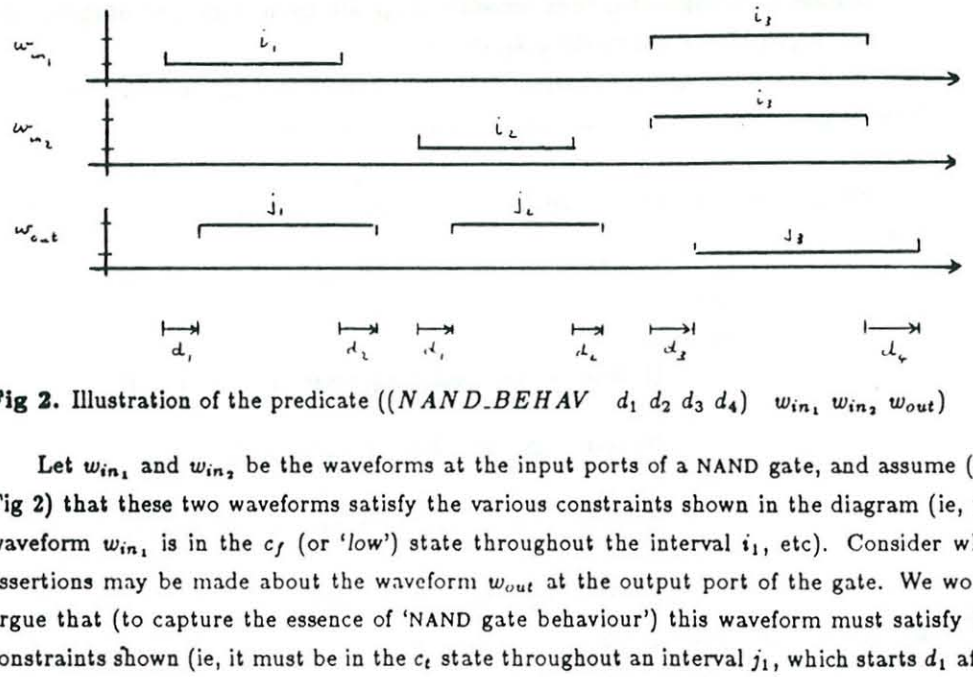

Fig 2. Illustration of the predicate $((NAND\_BEHAV \quad d_1\ d_2\ d_3\ d_4) \quad w_{in_1}\ w_{in_2}\ w_{out})$

Let w_{in_1} and w_{in_2} be the waveforms at the input ports of a NAND gate, and assume (see Fig 2) that these two waveforms satisfy the various constraints shown in the diagram (ie, the waveform w_{in_1} is in the c_f (or '*low*') state throughout the interval i_1, etc). Consider what assertions may be made about the waveform w_{out} at the output port of the gate. We would argue that (to capture the essence of 'NAND gate behaviour') this waveform must satisfy the constraints shown (ie, it must be in the c_t state throughout an interval j_1, which starts d_1 after i_1 starts and finishes d_2 after i_1 finishes, etc).

There are three interesting points about this definition. First, it is a *partial* definition of the behaviour: it places bounds on what is required, but it does not artificially overconstrain it. For example, it does not imply idealised rise and fall transients. Secondly, it is *non-strict* for 'low' inputs: for example, during interval i_1 it does not impose *any* requirements on the behaviour of waveform w_{in_2}. Thirdly, it does not state what the *actual* rise and fall times of the analogue waveforms must be: it simply specifies bounds on them.

Now let us make this definition precise. First, it is useful to introduce, as a subsidiary definition, a function $\Delta : dur^2 \rightarrow (intvl \rightarrow intvl)$ that 'propagates' intervals. The interval

$$((\Delta \quad d_1 \quad d_2) \quad i_1)$$

is *defined* to be the interval which starts d_1 later that i_1 starts and finishes d_2 later than i_1 finishes (it is, in fact, the interval j_1 of Fig(2)). Using this function, we can define the 'low-mode' aspects of the $NAND\_BEHAV$ predicate, as follows:

$$\begin{array}{l} \forall i : intvl. \\ \quad ((const \quad i \quad c_f) \quad w_{in_1}) \vee ((const \quad i \quad c_f) \quad w_{in_2}) \\ \quad \rightarrow \\ \quad ((const \quad ((\Delta \quad d_1 \quad d_2) \quad i) \quad c_t) \quad w_{out}) \end{array}$$

or, in narrative form:

"**For any** interval i, **if** waveform w_{in_1} is in state c_f throughout i, **or** waveform w_{in_2} is in state c_f throughout i, **then** waveform w_{out} will be in state c_t throughout the interval starting d_1 after i and finishing d_2 after i."

The corresponding 'high-mode' aspects is identical, save only that the roles of c_f and c_t are interchanged, that d_3, d_4 appear in place of d_1, d_2, and that $\vee$ (disjunction) is replaced by $\wedge$ (conjunction).

We can now present the actual definition of the behavioural predicate $NAND\_BEHAV$; it is

$$\begin{array}{l} ((NAND\_BEHAV \quad d_1 \quad d_2 \quad d_3 \quad d_4) \quad w_{in_1} \quad w_{in_2} \quad w_{out}) \\ =_{def} \\ \forall i : intvl. \\ \qquad ((const \quad i \quad c_f) \quad w_{in_1}) \vee ((const \quad i \quad c_f) \quad w_{in_2}) \\ \qquad \rightarrow \\ \qquad ((const \quad ((\Delta \quad d_1 \quad d_2) \quad i) \quad c_t) \quad w_{out}) \\ \quad \wedge \\ \qquad ((const \quad i \quad c_t) \quad w_{in_1}) \wedge ((const \quad i \quad c_t) \quad w_{in_2}) \\ \qquad \rightarrow \\ \qquad ((const \quad ((\Delta \quad d_3 \quad d_4) \quad i) \quad c_f) \quad w_{out}) \end{array}$$

In narrative form, this reads (approximately) as:

"The predicate $NAND\_BEHAV$ (that is deemed to characterise the behaviour of a NAND gate) applied to a 4-tuple of durations d_1, d_2, d_3, d_4 (representing the timing parameters of the gate), in turn applied to a 3-tuple of waveforms $w_{in_1}, w_{in_2}, w_{out}$ (representing the waveforms at the ports of the gate) **is defined as meaning** that **for any** interval of time, i, the two following assertions hold

(1) **if** input waveform w_{in_1} is in state c_f throughout i, **or** input waveform w_{in_2} is in state c_f throughout i, **then** output waveform w_{out} will be in state c_t throughout the interval starting d_1 after i starts, and finishing d_2 after i finishes, **and**

(2) **if** input waveform w_{in_1} is in state c_t throughout i, **and** input waveform w_{in_2} is in state c_t throughout i, **then** output waveform w_{out} will be in state c_f throughout the interval starting d_3 after i starts, and finishing d_4 after i finishes.

There are a number of points to make in connection with this definition.

- It is quite a complicated definition (although, with familiarity, it is readily comprehensible). However, *we would maintain that this complexity is intrinsic to the digital designers notion of the behaviour of a* NAND *gate.*
- One point we have deliberately glossed over is the *partial* nature of Δ, the 'propagate' function. In particular, we would *not* wish to be able to apply Δ to a *null* interval and have it yield a *non-null* interval. For example, we would not wish the interval defined by

$$((\Delta \quad 2 \quad 5) \quad (I \quad t \quad 0))$$

to be the interval $(I \quad t+2 \quad 3)$. In the present context, such behaviour (corresponding to 'pulse-stretching' a null pulse into a non-null one) would be distinctly non-physical! The axiomatic definition of Δ has been carefully contrived to avoid such a meaning. (In this respect, it differs from the definition of Δ as used in [4], in which (because all intervals were defined as being *closed* rather than semi-open, as here) it was necessary to 'guard' explicitly against such a possibility at every usage of Δ, a feature which gave behavioural definitions a somewhat clumsy appearance.)

3. Specification of a D-flipflop

In this section we will first describe, in informal terms, how (we believe) an 'edge-triggered, D-type flipflop' (or, in short, a D-flipflop) is meant to behave. We will then make this description more precise by recasting it as a collection of wavespec diagrams. Finally, we will formulate it as a wavespec *definition*.

3.1 Informal description

A D-flipflop (see Fig. 3a) has three ports: a 'strobe' (or clock) port (S), a 'data-in' port (D) and an 'output' port (Q). It is typically described in textbooks and manufacturer's specification sheets by a set of idealised waveform diagrams such as those shown in Fig. 3b. Data (on the D port) is strobed on the rising edge of a strobe pulse and transferred shortly afterwards to the Q port, where it remains until the next strobe pulse. A manufacturer's data sheet (eg [10]) will specify certain conditions that have to be satisfied, and certain parameters that are guaranteed.

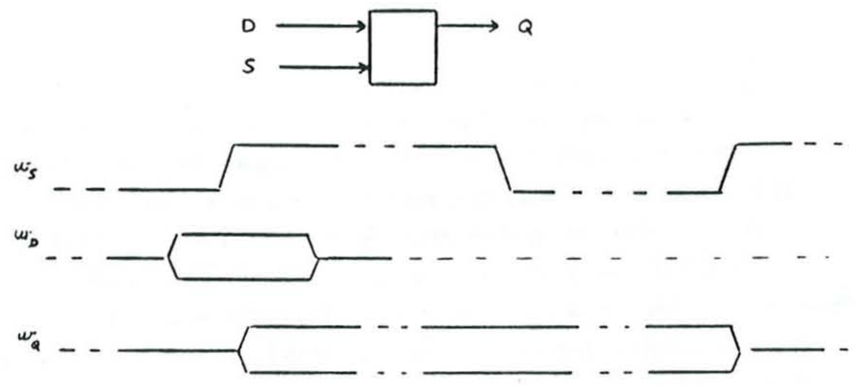

Fig 3.

(a) Graphical representation of a D-flipflop. S is the '*strobe*' (or *clock*) port, D is the '*data*' port, and Q is the '*output*' port.

(b) A typical 'idealised' behavioural description for a D-flipflop.

In the first category will (typically) be found maximum strobe rates and minimum data setup and hold times, and in the second category will (typically) be found maximum propagation delays.

3.2 Wavespec diagram description

Now consider a rather more careful description of the same device. As before (with the NAND gate), we emphasise that we are now attempting to formalise a *definition* of an already existing concept: the reader must decide an the extent to which he or she thinks we succeed in capturing the intent of the existing usage of this term. In passing, we note that we could (very easily!) devise a more complicated (or 'tighter fitting') definition than the one we are about to propose (for instance, we *could* decide to associate *different* timing parameters with different polarities of signals, etc). In the interests of keeping the exposition clear, however, we have deliberately chosen the most simple (yet realistic) definition possible. The definition we propose is of the form

$$\forall c : C.$$
$$(z_S \quad w_S) \wedge ((z_D \quad c) \quad w_D) \rightarrow ((z_Q \quad c) \quad w_Q)$$

Before describing the significance of this expression as a whole, first consider its component wavespecs, z_S, z_D etc. These are shown diagrammatically in Fig. 4

- z_S is a wavespec on the strobe waveform. It asserts that it must be *low* throughout the interval of duration a_1, and be *high* throughout an interval of duration of $a_3 + d$ (where d is a duration of arbitrary extent), and be *low* throughout an interval of duration e (where e is of arbitrary extent). Elsewhere the strobe waveform is unrestricted. Informally, we would say that this assertion describes a 'rising edge' on the strobe waveform, (with the

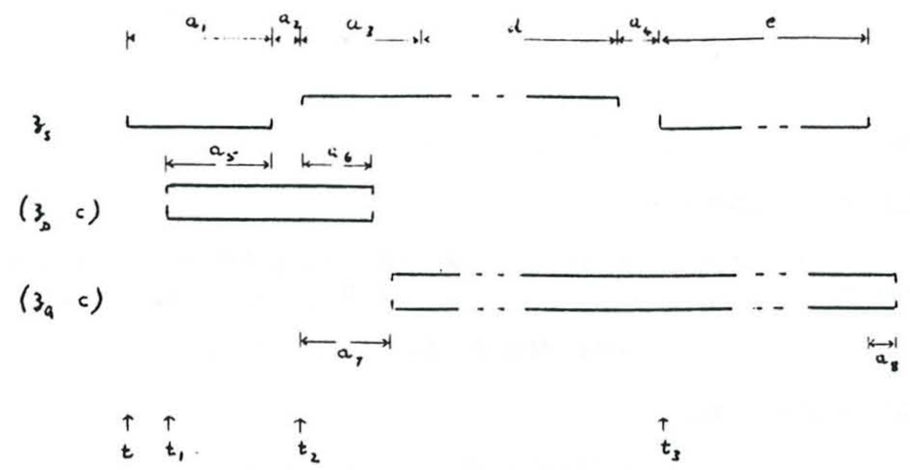

Fig 4. Wavespec definition of a D-flipflop. The timing parameters $\vec{a}$ correspond (loosely) to the following:

a_1	min strobe *low* time
a_2	max strobe rise time
a_3	min strobe *high* time
a_4	max strobe fall time
a_5	data setup time
a_6	data hold time
a_7	max output propagation delay
a_8	min output hold time

duration a_2 describing the maximum allowable rise time) followed by (if e is nonzero) a 'falling edge' (with maximum fall time a_4). Note carefully that the specification says nothing about the detailed 'shape' of either transition.

- $(z_D \quad c)$ is a wavespec on the data waveform. It asserts that the data waveform must be constant, at value c (where c is either c_t or c_f) throughout an interval which overlaps the 'rising edge' of the strobe waveform. The two duration a_5 and a_6 correspond to the required 'setup' and 'hold' times for the D-flipflop.
- $(z_Q \quad c)$ is a wavespec on the output waveform. It asserts that the output waveform w_Q is constant, at value c, throughout the interval shown. The two durations a_7 and a_8 describe the (maximum) propagation time and (minimum) hold time of the D-flipflop.

With these definitions, we can now appreciate the general form of the flipflop specification

$$\forall c : C.$$
$$(z_S \quad w_S) \wedge ((z_D \quad c) \quad w_D) \rightarrow ((z_Q \quad c) \quad w_Q)$$

In words "**For all** values of c (namely, either c_t or c_f), **if** w_S, the strobe waveform, satisfies wavespec z_S, **and** w_D, the data waveform, satisfies wavespec (z_D c), **then** w_Q, the output waveform, will satisfy wavespec (z_Q c)".

In passing, we note that, as a special case, the duration e may be zero: if this is so, then the definition will not require there to be a well-defined falling edge on the strobe waveform.

3.3 Formal definition

Now that we have described (informally) our understanding of the term "*The behaviour of* a *D-flipflop*", let us propose a formal definition for it. To this end, we introduce a new symbol

$$DFF\_BEHAV : dur^8 \rightarrow (wf^3 \rightarrow bool)$$

with the intention that

$$((DFF\_BEHAV \quad \vec{a}) \quad w_S \quad w_D \quad w_Q)$$

should express the assertion: "*The three waveforms w_S (the strobe waveform), w_D (the data waveform)* and w_Q *(the output waveform) exhibit the behavioural characteristics of* a *D-flipflop with timing parameters $\vec{a}$* ". Formally, we define this symbol by the axiom

$$\begin{array}{l}
\vdash \ ((DFF\_BEHAV \quad \vec{a}) \quad w_S \quad w_D \quad w_Q) \\
\quad =_{def} \\
\quad \forall c : C. \\
\quad\quad \forall t : time, \ d, e : dur. \\
\quad\quad\quad (z_S \quad w_S) \wedge ((z_D \quad c) \quad w_D) \rightarrow ((z_Q \quad c) \quad w_Q) \\
\quad\quad WHERE \\
\quad\quad\quad (z_S \quad w) \ = \\
\quad\quad\quad\quad\quad ((const \quad (I \quad t \quad a_1) \quad c_f) \quad w) \wedge \\
\quad\quad\quad\quad\quad ((const \quad (I \quad t_2 \quad a_3 + d) \quad c_t) \quad w) \wedge \\
\quad\quad\quad\quad\quad ((const \quad (I \quad t_3 \quad e) \quad c_f) \quad w) \\
\quad\quad\quad ((z_D \quad c) \quad w) \ = \\
\quad\quad\quad\quad\quad ((const \quad (I \quad t_1 \quad a_5 + a_2 + a_6) \quad c) \quad w) \\
\quad\quad\quad ((z_Q \quad c) \quad w) \ = \\
\quad\quad\quad\quad\quad ((const \quad ((\Delta \quad a_7 \quad a_8) \quad i) \quad c) \quad w) \\
\quad\quad WHERE \\
\quad\quad\quad i = (I \quad t_2 \quad a_3 + d + a_4 + e) \\
\quad\quad\quad t_3 = t_2 + a_3 + d + a_4 \\
\quad\quad WHERE \\
\quad\quad\quad t_1 = t + a_1 - a_5 \\
\quad\quad\quad t_2 = t + a_1 + a_2
\end{array}$$

(**Note**: The locally defined symbols t_1, t_2, t_3 and i were indicated in Fig. 4)

As with the definition of $NAND\_BEHAV$, we note that this definition is moderately complex. It is, however, just about the *simplest possible* definition that captures the digital designer's notion of what constitutes the behaviour of a D-flipflop. Further, we note that whilst it fully specifies it, it does *not* overspecify it. This is an important point: overspecification will in general limit the range of possible implementations and therefore unnecessarily increase implementation costs.

Standard Definitions As a final point, we note that, in formulating this definition, several choices had to exercised, as anyone starting off to frame such a definition *ab initio* will very soon discover! Thus, whilst we believe that the proposed definition is a reasonable one, it is not unique: many subtle variations are possible. One might envisage, one day, a forward looking standards organisation (BSI, IEEE, ISO, etc) taking on the task of formulating a set of 'standard' definitions for the elements of digital systems, much as Whitworth did for screw threads, or Bourbaki did for mathematics. In the shorter term, one might envisage individual companies each evolving their own 'house style', a shared set of definitions that would be available to anyone in the company who was involved in specification, implementation or verification. Such a 'database' (which would, in time, include derivations of useful lemmas as well) would be a valuable resource.

4. Implementation

Now that we have defined the behaviour that (at least in our eyes) a D-type flipflop is *meant* to exhibit, we will consider a possible implementation of one (there is, of course, an infinity of means by which such behaviours *could* be realised). The implementation we choose to examine is shown in Fig. 5. We came across it in the reference manual for a ULA manufacturer's cell library. Prior to that the same circuit may be found in [10] as a 'functional block diagram' for a SN7474 integrated circuit.

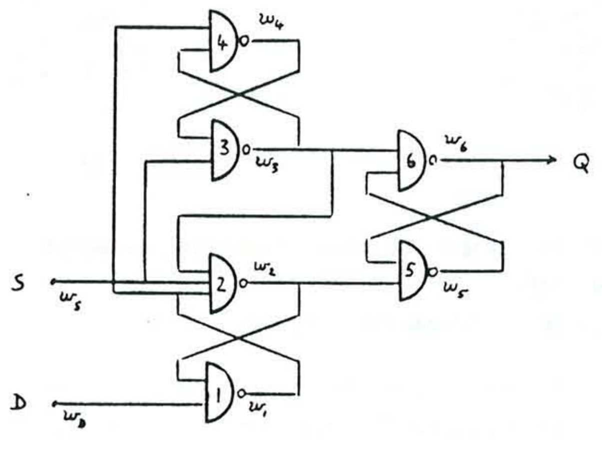

Fig 5. The circuit of a widely used implementation of a D-flipflop.

4.1 How it appears to work

Before undertaking (in §5) a formal proof that this circuit actually exhibits the behaviour of a D-flipflop, it naturally makes sense to try to gain some *intuitive* understanding of how the circuit operates. We say 'try' because, perhaps surprisingly in view of the apparent simplicity of the circuit, this is very difficult! It turns out, on analysis, that the *modus operandi* of this circuit is far from simple: in fact, it is unusually complex, and (so the authors found) difficult to understand intuitively*.

It turns out that the circuit operation can be understood most easily by considering the circuit as two subcircuits, of which the second can be treated as an ordinary set-reset flipflop (see Fig. 6a). This then allows the operation of the circuit to be appreciated in higher-level terms. Lest this manœuvre seem so commonplace as barely to warrant comment, we remark that the analogous substitution *cannot* be performed on either of the other two cross-coupled NAND gate pairs. For example, it turns out that it is *not* possible to explain the operation of the circuit in terms of Fig. 6b. This is because *other* properties of this particular NAND gate pair are required, over and above the fact that it behaves as a set-reset flipflop.

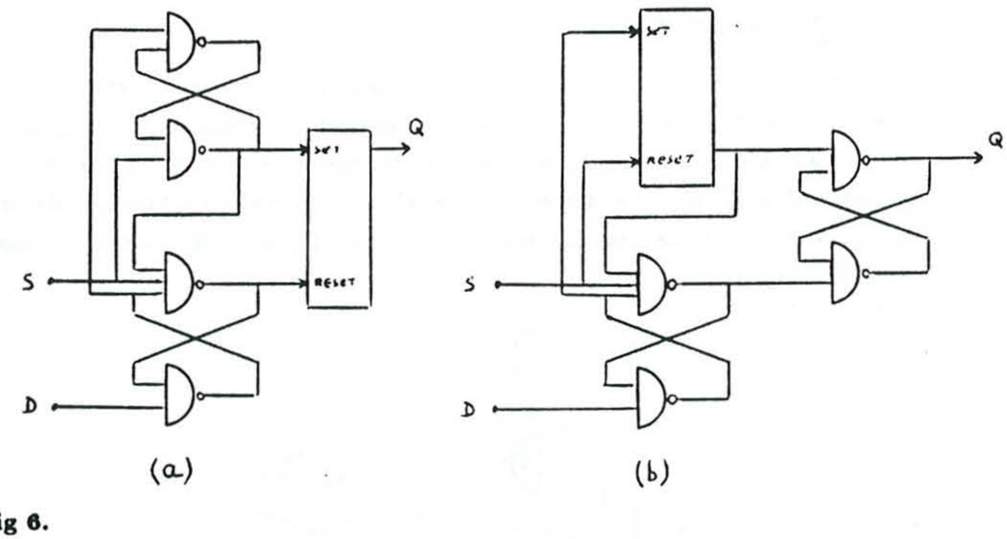

Fig 6.

(a) **The circuit shown in Fig 5 split into two subcircuits, the second of which has been replaced by a set-reset flipflop.**

(b) **An equally plausible, but unsatisfactory substitution.**

We will now offer a (very abbreviated) explanation, couched in the informal, everyday jargon of digital engineering, for the operation of the circuit shown in Fig. 6a. We start off

* If, like most people, you find this remark difficult to accept at face value, read the rest of this account, then set it aside, and attempt, within (say) *one working day*, to come up with a carefully justified account of 'how' the proposed implementation is intended to function ...

with the interesting observation that the mode of operation of the circuit is *essentially different* for each of the two polarities of data input. Understanding how the circuit operates when it latches a *high* data input gives little help in the understanding how it operates in latching a *low* data input. The following informal explanation deals only with the former case: we leave the latter case (which turns out to be a little more complex) as an exercise for the interested reader.

4.2 Informal explanation

In order to illustrate the following explanation, we have arbitrarily** chosen certain timing parameters for the NAND gates, and certain timing parameters for the overall flipflop specification. Using these parameters, we have then constructed (see Fig. 7) a set of wavespecs which illustrate (for a *high* data input) the operation of the circuit. In the following, highly informal, account we treat these wavespecs as *if* they were waveforms. The labelling $(a, \ldots, f)$ on the time axis of this diagram corresponds to the following itemisation:

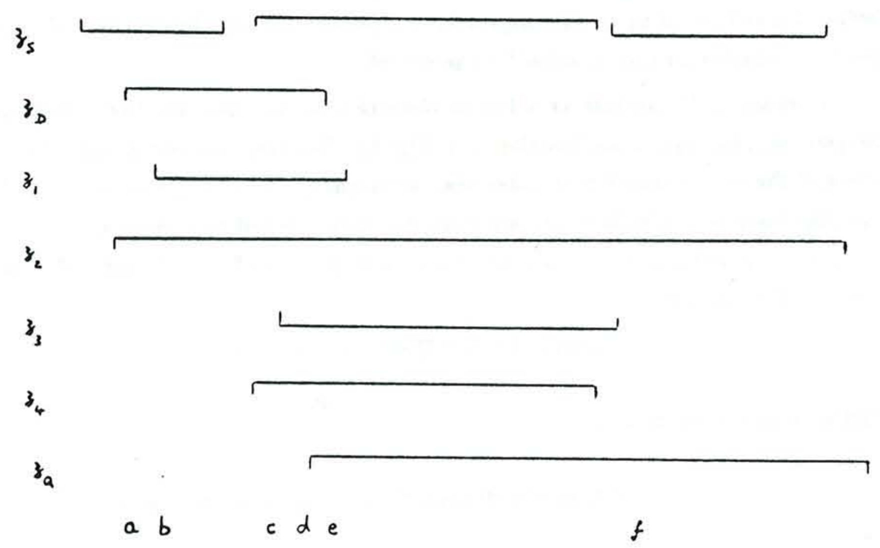

Fig 7. A set of wavespecs compatible with the circuit shown in Fig 5.

(a) Initially, w_S (the strobe waveform) is *low*. Thus w_2 (the output of gate 2) is *high*.

** As it happens (see §7.4) the circuit only functions 'correctly' for certain combinations of timing parameters, so it is *not* an entirely arbitrary choice!

(b) Then, w_D (the data waveform) becomes *high*. Gates g_1/g_2 latch this state. w_1 goes *low*.

(c) Then w_S (the strobe waveform) goes *high*. This drives w_3 *low*. Gates g_3/g_4 latch this state.

(d) w_2 remains *high* and w_3 is *low*. This *sets* the output latch g_5/g_6, with w_Q *high*

(e) w_D (the data waveform) becomes undefined. w_1 also becomes undefined. However, w_3 remains latched *low*, and hence maintains w_2 *high*. Thus output latch g_5/g_6 remains *set*.

(f) w_S (the strobe waveform) goes *low*. This change races through g_2 and g_3: assuming it is a sufficiently rapid transition, w_2 remains *high* and thus the output latch g_5/g_6 remains *set*, with w_Q *high*. (Most of the other gates go into an undefined state).

Again, we emphasise the highly informal, and indeed quite unconvincing, nature of this account. In fact, this example provides an excellent illustration of the *raison d'etre* for formal verification.

4.3 Formal specification of the implementation

Elsewhere ([4]) we have explained how, using higher-order logic, the *structure* (ie, the circuit diagram) of a device may be specified, in the form of an axiomatic theory, and how (by formal inferencing) the corresponding behavioural specification may be obtained from it. Thus, we will not dwell at all upon this aspect in the present account but rather take as our starting point a *behavioural* specification for the circuit.

In order to frame this as a formal *definition* we assume that the waveforms are named $w_S, w_D, w_Q, w_1, w_2, \ldots, w_5$ (as shown in Fig. 5). Further, we assume that the timing parameters of the component NAND gates are described by $\vec{n}$, a 4-tuple of durations. Under these assumptions, we can write down a set of assertions satisfied by these waveforms. Each component NAND gate imposes a relation between various of these waveforms. For example, gate g_1 imposes the relation

$$((NAND\_BEHAV \quad \vec{n}) \quad w_2 \quad w_D \quad w_1)$$

and gate g_2 the relation

$$((NAND3\_BEHAV \quad \vec{n}) \quad w_3 \quad w_S \quad w_1 \quad w_2)$$

The relation imposed by all the gates together is the conjunction of all the component relations.

At this point we introduce a new symbol to denote the wavespec (or, more precisely, the *weakest* wavespec) satisfied by the waveforms. Because the component wavespecs are parameterised by $\vec{n}$, so also this overall wavespec will be. Thus, we introduce the symbol

$$IMPL\_BEHAV \quad : \quad dur^4 \rightarrow (wf^8 \rightarrow bool)$$

(a parameterised wavespec for the behaviour of the implementation) with definition

$$
\begin{array}{l}
((IMPL\_BEHAV \ \ \vec{n}) \ \ w_S \ \ w_D \ \ w_Q \ \ w_1 \ \ w_2 \ \ w_3 \ \ w_4 \ \ w_5) \\
=_{def} \\
\quad ((NAND\_BEHAV \ \ \vec{n}) \ \ w_2 \ \ w_D \ \ w_1) \\
\wedge \\
\quad ((NAND3\_BEHAV \ \ \vec{n}) \ \ w_3 \ \ w_S \ \ w_1 \ \ w_2) \\
\wedge \\
\quad ((NAND\_BEHAV \ \ \vec{n}) \ \ w_4 \ \ w_S \ \ w_3) \\
\wedge \\
\quad ((NAND\_BEHAV \ \ \vec{n}) \ \ w_1 \ \ w_3 \ \ w_4) \\
\wedge \\
\quad ((NAND\_BEHAV \ \ \vec{n}) \ \ w_Q \ \ w_2 \ \ w_5) \\
\wedge \\
\quad ((NAND\_BEHAV \ \ \vec{n}) \ \ w_3 \ \ w_5 \ \ w_Q)
\end{array}
$$

The waveforms associated with the circuit shown in Fig. 5 (ie, the proposed implementation of a D-flipflop) will therefore (assuming the tuple of timing parameters of each of the component NAND gates is $\vec{n}$) satisfy the wavespec

$$(IMPL\_BEHAV \ \ \vec{n}) \ \ : \ \ wf^8 \rightarrow bool$$

The above definition will be used in the next section

5. Formal verification

We now move onto the second major theme of this paper, that of the 'formal verification' of the correctness of the proposed implementation of the D-flipflop. In principle, the task is a very simple one.

- On the one hand, we have (for an arbitrary 8-tuple $\vec{a}$ of durations) a wavespec

  $$(DFF\_BEHAV \ \ \vec{a}) \ \ : \ \ wf^3 \rightarrow bool$$

  which describes the relation that we *require to be satisfied* between the triple of waveforms w_S, w_D, w_Q at the ports of a D-flipflop.
- On the other hand, we have (for an arbitrary 4-tuple $\vec{n}$ of durations) a wavespec

  $$(IMPL\_BEHAV \ \ \vec{n}) \ \ : wf^8 \rightarrow bool$$

  which describes the relation that we *know to be satisfied* between the 8-tuple of waveforms $w_S, w_D, w_Q, w_1, \ldots, w_5$ at the ports ('internal' and 'external') of the proposed implementation.

What we need to do, in order to 'verify the design' (that is to prove its correctness w.r.t. the behavioral definition for a D-flipflop) is (roughly stated) to establish that *any* set of waveforms which satisfies the second of these wavespecs also satisfies the first.

Stated a little more carefully, as a formula, this would be:

$$\begin{array}{l} \forall w_S, w_D, w_Q, w_1, w_2, w_3, w_4, w_5 : wf. \\ \quad ((IMPL\_BEHAV \quad \vec{n}) \quad w_S \quad w_D \quad w_Q \quad w_1 \quad w_2 \quad w_3 \quad w_4 \quad w_5) \\ \quad \rightarrow \\ \quad ((DFF\_BEHAV \quad \vec{a}) \quad w_S \quad w_D \quad w_Q) \end{array}$$

Of course, this formula is *not* going to be true for arbitrary values* of $\vec{a}$ (the 8-tuple of timing parameters of the flipflop) and $\vec{n}$ (the 4-tuple of timing parameters of the component NAND gates). Thus, we will postulate a relation

$$R \quad : \quad dur^8 \times dur^4 \rightarrow bool$$

that will define the necessary relationship between $\vec{a}$ and $\vec{n}$ if the above formula is to be true. We shall call R the '*timing relation*'. Using R, we can now state the actual theorem that we wish to establish

$$\begin{array}{l} \forall \vec{a} : dur^8, \ \vec{n} : dur^4. \\ \quad (R \quad \vec{a} \quad \vec{n}) \\ \quad \rightarrow \\ \quad \forall w_S, w_D, w_Q, w_1, w_2, w_3, w_4, w_5 : wf. \\ \qquad ((IMPL\_BEHAV \quad \vec{n}) \quad w_S \quad w_D \quad w_Q \quad w_1 \quad w_2 \quad w_3 \quad w_4 \quad w_5) \\ \qquad \rightarrow \\ \qquad ((DFF\_BEHAV \quad \vec{a}) \quad w_S \quad w_D \quad w_Q) \end{array}$$

We shall call this (as yet unproven) theorem the *D-flipflop implementation theorem.* In words, it states: "**For any** 8-tuple $\vec{a}$ and 4-tuple $\vec{n}$ of durations, **if** they satisfy the condition $(R \quad \vec{a} \quad \vec{n})$, **then any** set of waveforms that occur at the nodes of the proposed implementation (whose NAND gates have timing parameters $\vec{n}$) will meet the specification of a D-flipflop (with timing parameters $\vec{a}$)."

General Aims The example we have just described is entirely typical of hardware verification. Looking at it, we see that we have, in fact, two distinct but interrelated problems to solve. One is evidently to prove (formally or informally, as the circumstances warrant) the above theorem. The other (not so evident) task, however, is to find a definition for the timing relation R. (Notice carefully that we do *not* imply that such a relation is unique, nor do we imply that a non-trivial one exists.) Although one might imagine that proposing a suitable relation is an easy task, experience contradicts this. It turns out that determining a 'suitable' value for R is as difficult a task as proving the above theorem. In our experience, 'intuitive' attempts at deciding a value for R are *invariably* wrong!

* For example, it is intuitively obvious that a 'fast' flipflop cannot be realised in terms of 'slow' NAND gates.

The best way that we have found to determine R is to begin by leaving it undefined, and then to attempt to prove the desired theorem. During the course of this proof, one will be forced to make certain 'assumptions' in order to carry through the proof. Then, when the (conditional) proof has been achieved, one can gather up all the assumptions (ie, relations between timing parameters) and *chose* a relation R which is

- sufficiently *strong* to imply all the required assumptions, yet
- sufficiently *simple* to be of practical use (no one is going to be interested in a relation that takes a couple of pages to express!).

The proof can then be reworked with this definition of R present as an axiom.

6. Computational Inferencing

In this section, we review the means by which we carry through the two interrelated aims described above, namely:

- Determining a 'suitable' definition for the timing criterion R, and
- Proving the D-flipflop implementation theorem.

As we shall see, this involves constructing a large and complex proof. We undertake this task 'computationally', in order both to ensure the *correctness* of our inferencing (by eliminating the possibility of 'slips', wishful thinking, etc) and to be able to automate the lower-level aspects of it.

The approach we use originally derives from a method pioneered in the Edinburgh ML/LCF project [1]. In this account, we only have space to give a most cursory overview. The approach is very fully described in [5] (which includes an extensive tutorial exposition).

General Principles A *derivation* (or 'proof') is a *tree*. The *nodes* of the tree are labelled with the primitive *rules of inference* of the logic (for example, *modus ponens*, *generalise*, *specialise*, are some of the rules of inference of the VERITAS logic). The *arcs* of the tree are labelled with *theorems*. The *tips* of the tree are labelled with *axioms*. A derivation is not a free-standing entity, but rather it exists only with respect to a *signature*. A signature (sometimes called a 'theory presentation') is essentially a sequence of *declarations* of the *symbols* and *axioms* that may occur in a derivation.

The VERITAS logic is *computationally implemented* within a programming language MV (for 'META-VERITAS'). This is an interactive, typed, purely functional programming language. The implementation of this language includes (amongst others) abstract data types (ADT's) for *signatures*, *terms* and *derivations*. Associated with these ADT's are functions for constructing, manipulating and displaying values of each type. These functions, which have been very carefully defined and implemented, are such as to *guarantee* the syntactic correctness and the logical soundness of any entity constructed. For example, consider the MV function

$$spec\ :\ deriv \times term \rightarrow deriv$$

which implements the '*specialise*' rule of inference. The function takes two arguments; a derivation and a term. The intention is that the first argument should be a derivation of a theorem of the form

$$\vdash \forall x : s. \quad tm$$

and the second a term tm' of type s. In this event, then the value returned by this function call will be a derivation of a theorem of the form

$$\vdash tm[tm'/x]$$

(ie, the term tm with all free occurrences of the symbol x replaced by the term tm', under the assumption that no capture occurs). Incorporated within the implementation of the ADT function *spec* (and completely out of sight to the user) is a rigorous set of checks (on the types and values of the two arguments) that ensures that, unless the resultant derivation would be correct in every respect, an error condition arises. Similar checks are built into each of the other primitive constructor functions.

6.1 High-level Inferencing

Whilst it is, in principle, possible to construct a derivation of any theorem simply in terms of the primitive ADT term and derivation constructors described above, it would be unimaginably tedious to do so. There are three techniques that may be used to reduce the effort required by a user:

- Functions which implement *derived* inference rules may be written in the meta-language. The body of such a function is a complex expression which, each time an application of the function is evaluated, results in the application of many primitive term and derivation constructors. As an illustration, a very simple example of a derived inference rule would be a function which, given a derivation of a theorem of the form $\vdash \forall x.\ \forall y.\ \forall z.\ P_{x,y,z}$ yields a derivation of the theorem $\vdash \forall x, y, z.\ P_{x,y,z}$, that is, it *tuples* a nested sequence of quantifiers.
- A 'database' of useful lemmas (ie, subsidiary theorems) may be built up. In general, the axioms that are chosen to characterise a theory will be both very simple and few in number (the two factors helping to ensure that the truth of the axioms is beyond reasonable doubt). Because they are simple and few in number, they cannot encompass the many arithmetic, algebraic and digital engineering relations that we would like to be able to take for granted. Thus, it is desirable to build up a database of a few hundred generally useful lemmas. To give a simple example, the following lemma

$$\vdash \forall a, b, c : nat. \quad (a \geq b) \wedge (b > c) \rightarrow (a > c)$$

is rep ative of the couple of dozen lemmas concerning arithmetic ordering relations.

- The third technique we mention is that of *goal-directed* inferencing. The idea is a very simple one. It relies upon the fact that if one knows, in advance, the theorem one is endevouring to prove (ie, the *goal*) it is often easy to obtain from it a collection of *subgoals* with the properties (a) that, individually, they are easier to prove than was the original goal, and (b) that, once they themselves have been established as theorems,the original goal may itself (by the use of a derived inference rule) be established as a theorem. The functions that are used for this purpose are known as *tactics*. A typical example of a tactic is one for proving theorems by the use of *induction*. For instance, given a goal of the form

  $$\forall n : nat.\ P_n$$

  (where P_n is some formula in which n occurs free), the tactic will yield two subgoals namely

  $$P_n[0/n] \qquad \text{and} \qquad [P_n] \quad P_n[(succ\ n)/n]$$

  and, once these subgoals have been established as theorems, will use an appropriate derived inference rule to infer the original goal.

We describe the system with which the user interacts in constructing a derivation as a *proof assistant*. Basically, it is a program (written in MV) which allows the user to specify the *top-level* goal, to examine any subgoal (and the declarations of symbols and axioms on its associated signature), and to specify the tactic to be used to achieve each of the subgoals.

It is often useful, especially when undertaking new proofs of an unfamiliar kind, to be able to display, in graphical form, the derivation (or completed part thereof). Thus, the proof assistant program provides the means for doing this; as an illustration of this sort of display, Fig 8 shows a typical derivation (in fact, it is a derivation of the NAND-PAIR lemma that will be used in §7.2 below).

This concludes our very brief look at the logic and the techniques used in its computational implementation. For the remainder of this paper, we will limit ourselves to a relatively informal, high-level description of the goal-directed inferencing process we have used to prove the D-flipflop theorem and obtain the timing criterion R.

7. Proving the D-flipflop Implementation Theorem

So far, we have proposed definitions for the behaviour of a NAND gate and for a D-flipflop, we have illustrated one commonly used implementation of a D-flipflop, we have written down the theorem we would like to establish, and we have outlined the approach we use in computational theorem proving. In this section we shall draw all these threads together and describe how we actually went about computationally proving the theorem and determining the timing relation R.

We have already noted that, even at an *intuitive* level, the mode of operation of the proposed implementation is far from obvious. It will scarcely come as a surprise, therefore,

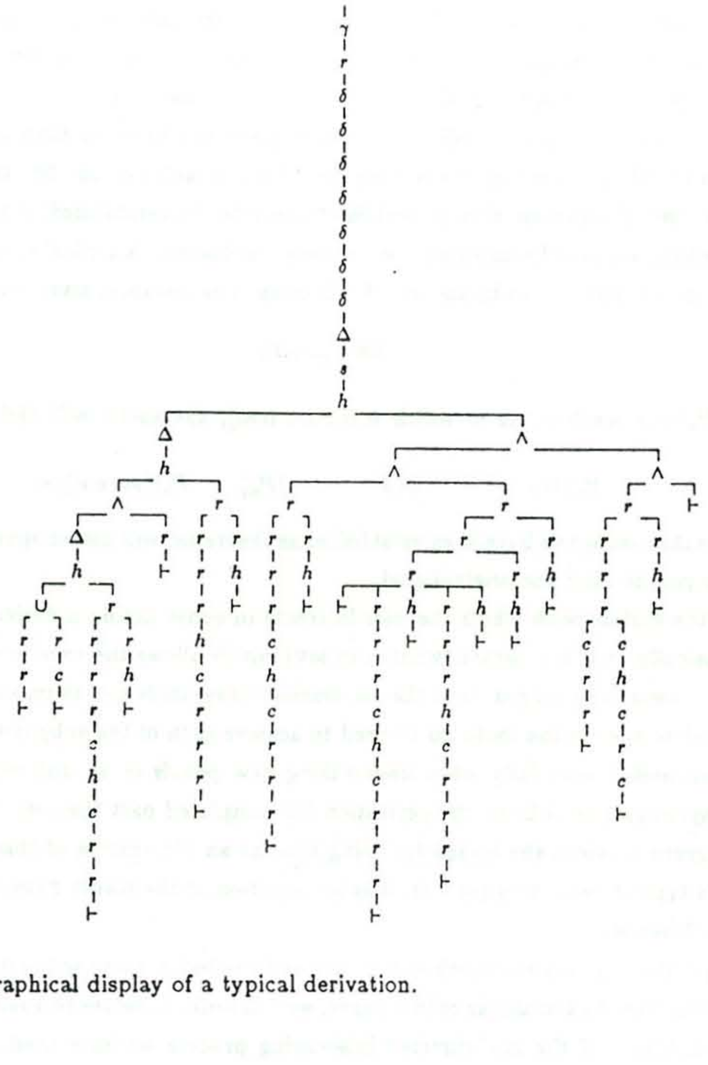

Fig 8. Graphical display of a typical derivation.

that a formal derivation of the D-flipflip implementation theorem is quite complicated. Thus, we are only able to give a general, high-level account of the way we go about constructing this derivation. We mention, however, for anyone who may interested in examining the actual proof, that we are in the process of preparing an annotated step-by-step transcript of it, in the form of a C.A.I interactive document.

Let us start this account of the proof by restating, as our 'goal', the formula we are intending to prove:

$$\begin{array}{l} \forall\, \vec{a} : dur^{8},\ \vec{n} : dur^{4}. \\ \quad (R \quad \vec{a} \quad \vec{n}) \\ \quad \rightarrow \\ \quad \forall\, w_S, w_D, w_Q, w_1, w_2, w_3, w_4, w_5 : wf \\ \qquad ((IMPL\_BEHAV \quad \vec{n}) \quad w_S \quad w_D \quad w_Q \quad w_1 \quad w_2 \quad w_3 \quad w_4 \quad w_5) \\ \qquad \rightarrow \\ \qquad ((DFF\_BEHAV \quad \vec{a}) \quad w_S \quad w_D \quad w_Q) \end{array}$$

As we have explained such a goal is not a free-standing entity, but rather has associated with it a signature. The signature declares all the symbols and axioms that are, in effect, 'within scope', that is, that may be used in constructing a derivation of the theorem. The 'earlier' parts of this signature introduce symbols and axioms associated with propositional logic, arithmetic, instants of time, durations, intervals and waveforms (as briefly described in §2 above). The 'later' (and thus more problem specific) parts of the signature include declarations for the symbols *NAND_BEHAV*, *DFF_BEHAV* and *IMPL_BEHAV*, and their defining axioms (precisely as we have defined them in §2.3 of this account). The signature also includes a declaration for the timing condition R, but, as yet, no axiom defining it.

7.1 First Part of the Proof

We now give an itemised account of the main stages gone through in constructing (in goal-directed mode) a derivation of the D-flipflop implementation theorem. In presenting the following account, we explain that it is highly abbreviated, and hence gives a somewhat idealised view.

(1) The first step, before even calling the proof assistant, is to construct the signature (or, alternately stated, to define the theory presentation) with respect to which inferencing will take place. This is simply done by 'extending' an existing (standard) signature with the definitions (as given earlier) for the symbols *IMP_BEHAV*, and *DFF_BEHAV* and a declaration (but no defining axiom) for the symbol R.

(2) The 'proof assistant' function (in the meta-language) is then called, with the goal (defined w.r.t the above signature) as its argument.

(3) The first couple of steps are concerned with "stripping-off" (by the use of the appropriate tactics) the universally quantified symbols and the hypothesis part of the outer implication. This results in the subgoal

$$\begin{array}{l} \forall\, w_S, w_D, w_Q, w_1, w_2, w_3, w_4, w_5 : wf \\ \quad ((IMPL\_BEHAV \quad \vec{n}) \quad w_S \quad w_D \quad w_Q \quad w_1 \quad w_2 \quad w_3 \quad w_4 \quad w_5) \\ \quad \rightarrow \\ \quad ((DFF\_BEHAV \quad \vec{a}) \quad w_S \quad w_D \quad w_Q) \end{array}$$

The signature associated with this subgoal now includes declarations for the symbols $\vec{a}$ and $\vec{n}$ as 'constants', and for $(R \quad \vec{a} \quad \vec{n})$ as an axiom or 'assumption'.

(3) The universally quantified variables $w_S, \ldots, w_5$ are likewise stripped-off, and then the symbol *IMPL_BEHAV* is replaced by the RHS of its definition (remember that, in eventually constructing the derivation, exactly the *opposite* of all these moves takes place). This then yields a subgoal of the form

$$\begin{array}{l} ((NAND\_BEHAV\ \ \vec{n})\ \ w_2\ \ w_D\ \ w_1) \\ \wedge \\ \quad \vdots \\ \wedge \\ ((NAND\_BEHAV\ \ \vec{n})\ \ w_3\ \ w_5\ \ w_Q) \\ \rightarrow \\ ((DFF\_BEHAV\ \ \vec{a})\ \ w_S\ \ w_D\ \ w_Q) \end{array}$$

(5) Again, the hypothesis part of this formula is stripped-off, so that the six *NAND_BEHAV* clauses appear as axioms in the signature. Then *DFF_BEHAV* is replaced by its definition. This yields the subgoal

$$\begin{array}{l} \forall\, c : C. \\ \quad \forall\, t : time,\ d, e : dur. \\ \qquad (z_S\ \ w_S) \wedge ((z_D\ \ c)\ \ w_D) \rightarrow ((z_Q\ \ c)\ \ w_Q) \\ \qquad WHERE \\ \qquad (z_S\ \ w) \ \ = \ \ \ldots \\ \qquad \quad \vdots \\ \qquad WHERE \\ \qquad t_1 \ \ = \ \ t + a_1 - a_5 \\ \qquad t_2 \ \ = \ \ t + a_1 + a_2 \end{array}$$

(6) Case analysis on c is carried out: this yields two subgoals. One is for c_t and the other for c_f (ie, for *high* and for *low* data inputs respectively). Here, we will consider only the former case. The case analysis is followed by stripping-off the universally quantified variables, replacing the terms $(z_S\ \ w_S)$ and $((z_D\ \ c_t)\ \ w_D)$ by their definitions and then stripping these off (so that they too become axioms). This results in the subgoal

$$\begin{array}{l} ((const\ \ ((\Delta\ \ a_7\ \ a_8)\ \ i)\ \ c_t)\ \ w_Q) \\ WHERE\ \ i \ \ = \ \ (I\ \ t_2\ \ a_3 + d + a_4 + e) \\ WHERE\ \ t_2 \ \ = \ \ t + a_1 + a_2 \end{array}$$

This concludes the first part of the proof strategy: we now have as our goal the above assertion involving the 'output' waveform w_Q, and a whole set of assumptions (present as axioms in the signature of the goal), including wavespecs on the 'strobe' and 'data' waveforms, w_S and w_D.

7.2 Specialised Techniques

Before continuing with the second part of this proof, we pause to describe some techniques that we have introduced to facilitate proving goals of this general kind. These techniques are:

- the notion of 'propagating assertions through gates',
- tactics that allow this type of operation to be carried out in a goal-directed manner,
- tactics for performing 'cutting and stitching operations' on intervals,
- a technique for retrospectively deciding what the timing criterion R should be.

We deal with each of these in turn.

Propagating assertions through gates A technique that we use extensively is, expressed informally, that of propagating assertions through gates. We illustrate the idea with a simple example. Suppose that w_1 and w_2 are the waveforms at the input ports of a NAND gate and that w_3 is the waveform at its output port. Further, suppose that the assertion

$$((const \quad i \quad c_f) \quad w_1)$$

has been established, that is, the w_1 input waveform is known to be in the *low* state during an interval i. Let us (meta-linguistically) name this assertion α_1 (the subscript '1' informally indicating that it relates to waveform w_1). Then, by using the (axiomatic) definition of $NAND\_BEHAV$, we can infer (essentially by *specialisation* followed by *modus ponens*) the assertion

$$((const \quad ((\Delta \quad d_1 \quad d_2) \quad i) \quad c_t) \quad w_3)$$

This assertion (name it α_3) relates to w_3, the output waveform. Intuitively, we can think of assertion α_3 as being obtained by "propagating assertion α_1 through gate g".

Tactics for 'propagating assertions' In order to be able to use the intuitively appealing style of reasoning described above in a *goal-directed* manner, we must be able to express it as a tactic. At first sight, this may seem difficult because goal-directed reasoning inherently proceeds in a 'backwards' direction, whereas the above style of reasoning is inherently 'forwards'. As the following example shows, however, these two conflicting requirements can be reconciled.

Suppose that, for the simple circuit shown in Fig 9, we wish to establish a goal of the form $\alpha_0 \rightarrow \alpha_n$ (where $\alpha_0, \ldots, \alpha_n$ are assertions on waveforms $w_0, \ldots, w_n$ respectively).

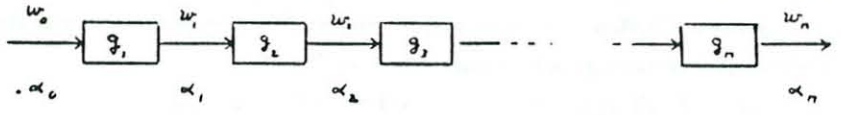

Fig 9 A simple circuit. $\alpha_0, \ldots, \alpha_n$ are assertions involving the waveforms $w_0, \ldots, w_n$ respectively.

The tactic that we use to achieve this operates as follows. First of all, it takes the assertion α_0 and 'propagates' it through gate g_1, thus obtaining a new assertion, α_1 , and a theorem of the form $\vdash \alpha_0 \rightarrow \alpha_1$. The tactic then generates a subgoal of the form $\alpha_1 \rightarrow \alpha_n$. Once this subgoal (which is one step 'easier' to achieve than the original goal) has eventually been achieved (by further subgoaling, etc), it is then a trivial matter for the tactic to infer (essentially by using *modus ponens* and the above theorem) the original goal as a theorem.

This mechanism is, of course, entirely encapsulated within the body of the 'propagate' tactic (or, more precisely, set of such tactics, since there is one for each kind of gate), and so, to a user, the effect is simply as if he was propagating assertions through the circuit in the same manner that a digital engineer is used to thinking of signals as propagating through a circuit.

Tactics for manipulating intervals There is another specialised group of tactics that we also employ extensively: these are used for 'cutting and stitching' the intervals on which wavespecs are defined. For example (see Fig. 10), given a pair of wavespecs z_1 and z_2 (of the same polarity, and assumed to be qualifying the same waveform) one might wish to assume that the intervals on which they are defined overlap, and hence, taken together, imply the wavespec z defined over the union of the two intervals.

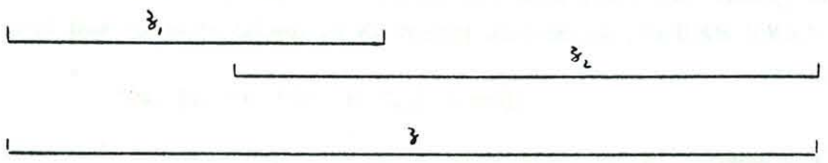

Fig 10 Merging two assertions.

In order to justify this assumption (ie, that the intervals overlap), the proof assistant will generate, as an additional goal, a relation (generally a conjunction of inequalities) that must hold between the two intervals. Because of the number of ways in which intervals may overlap or fail to overlap (including cases where one or other or both intervals are empty), these relations may be unexpectedly complex.

The NAND-PAIR lemma In addition to these specialised tactics, we also make use of a collection of behavioral lemmas. A representative example is one we have named the NAND-PAIR lemma (see Fig. 11). It essentially asserts that a pair of cross-coupled NAND gates behaves like a *set-reset flipflop*, that is to say, it will 'latch' a *low* pulse on one of its inputs, and stay in this state for as long as the other input remains *high*.

In more detail, it asserts that (provided the timing parameters $\vec{b}$ of the flipflop are appropriately related to those of the constituent NAND gates), if the assertions α_{set} and α_{reset} hold, then assertion α_{out} will also hold. In passing, we remark that there are several closely related variants of this lemma.

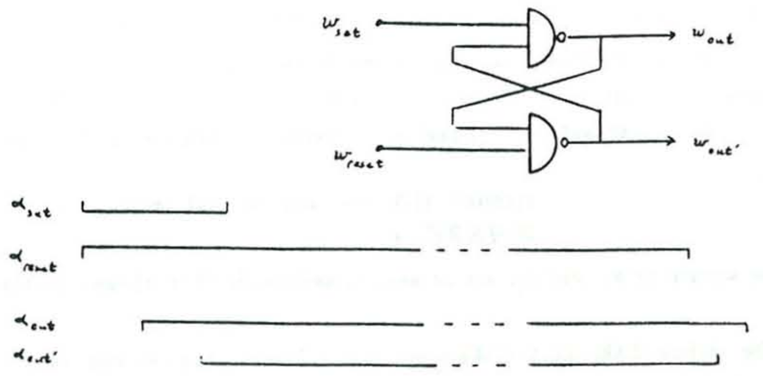

Fig 11 The NAND-PAIR lemma. Assertions α_{set} and α_{reset} imply assertion α_{out}.

Definition of the relation R The net effect of using the various tactics and lemmas we have just been describing is that, as the goal-directed construction of the proof tree proceeds, subsidiary subgoals involving relations between the timing parameters are generated. In general, we do not attempt to satisfy these subgoals immediately, but rather leave them as 'stubs' (ie, as unfinished branches of the goal tree). Then, when all other parts of the goal-tree have been filled in, we gather together the entire set (call it Σ) of such relations and determine a 'suitable' definition for the relation R. Obviously, one possible definition for R would simply be the conjunction of all the relations in Σ, but (because Σ is quite large) such a definition would be far too complex to be of any practical interest. Instead, what we do is to examine Σ carefully and choose a definition for R which is both *reasonably simple* and *sufficiently strong* (ie, strong enough to logically imply each of the relations in R). This takes a degree of skill and judgement. With R chosen, we then incorporate its defining axiom in the signature, and revisit the stubs (ie, unfinished branches) of the goal tree and complete them.

This concludes our summary of the general techniques we use during the second half of the proof. We will now continue the account of the construction of the D-flipflop theorem.

7.3 Second Part of the Proof

The second part of the proof is entirely concerned with 'propagating assertions' through the proposed implementation of the D-flipflop. The overall aim is, briefly stated, to demonstrate that the implementation does, in fact, succeed in 'latching' the input signal. (Recollect that, of the two possible polarities of input signal, only the c_t case is presently under consideration; the c_f case is the subject of a separate proof, not described here.) Stated in a little more detail, the aim is to demonstrate (by goal-directed inferencing) that

under the assumption that the strobe waveform w_S satisfies the strobe wavespec z_S,

and the data-in waveform w_D satisfies the data-in wavespec $(z_D \quad c_t)$, **then** the output waveform w_Q will satisfy the output wavespec $(z_Q \quad c_t)$.

Call these three assertions (or, more precisely, the expanded form of these three assertions, as defined in §7.1 above) α_S, α_D and α_Q respectively. The assertion α_Q is the current goal, namely

$$((const \quad ((\Delta \quad a_7 \quad a_8) \quad i) \quad c_t) \quad w_Q)$$
$$WHERE \quad i \ = \ \ldots$$

and the assertions α_S and α_D are present as axioms (ie, 'hypotheses') in the signature of the goal.

The outline of this part of the proof, which involves the detailed consideration of a series of related wavespecs, is easy to follow if presented in graphical form (see Fig. 12). Essentially, we will be propagating assertions α_S and α_D through the proposed implementation of the D-flipflop (as shown in Fig 5), and demonstrating that they imply assertion α_Q.

In the following description, we use the notation α_i^j to denote the j^{th} assertion relating to waveform w_i.

α_S This assertion specifies the strobe waveform, w_S. It is an *assumption*, that is, it is an axiom present in the signature associated with the current goal.

α_D Similarly, this assertion specifies the data waveform, w_D. Again, it is an assumption.

α_S^1 This assertion is simply the first conjunct of assertion α_S.

α_2^1 This assertion (the *first* one relating to w_2, the waveform on the output of gate $g2$) results from propagating assertion α_S^1 through NAND gate g_2. (*Note:* Less colloquially, we would say that assertion α_2^1 may be inferred from assertion α_S^1 and the axiomatic definition of the wavespec $NAND\_BEHAV$.)

α_2^2 This assertion is simply the assertion α_2^1 restricted to a subinterval whose start has been chosen to coincide with the start of the interval associated with assertion α_D.

α_2^3 This assertion is established using assertions α_D and α_2^2, together with a variant of the NAND-PAIR lemma described above. The two gates which in this case form the pair are g_1 and g_2.

α_1^1 This assertion results from propagating the *two* assertions α_D and α_2^3 through NAND gate g_1. (Compare this with the earlier case where a *single* assertion was propagated through a NAND gate. In the present case, the input assertions are *high* ones, and hence an assertion is needed for *each* of the (two) inputs to the gate. In the earlier case, which involved a *low* assertion, only a single such one was required. This difference is due to the fact that the defining axiom for the wavespec $NAND\_BEHAV$ (see §2.3, earlier) has disjunctive hypotheses for *low* input assertions but conjunctive ones for *high* input assertions.)

α_2^4 This assertion results from propagating the *low* assertion α_1^1 through gate g_2.

α_2^5 This assertion results from 'joining together' assertions α_2^1 and α_2^4. *Note:* We are able to join them together because

(a) The two assertions relate to the same waveform, namely w_2,

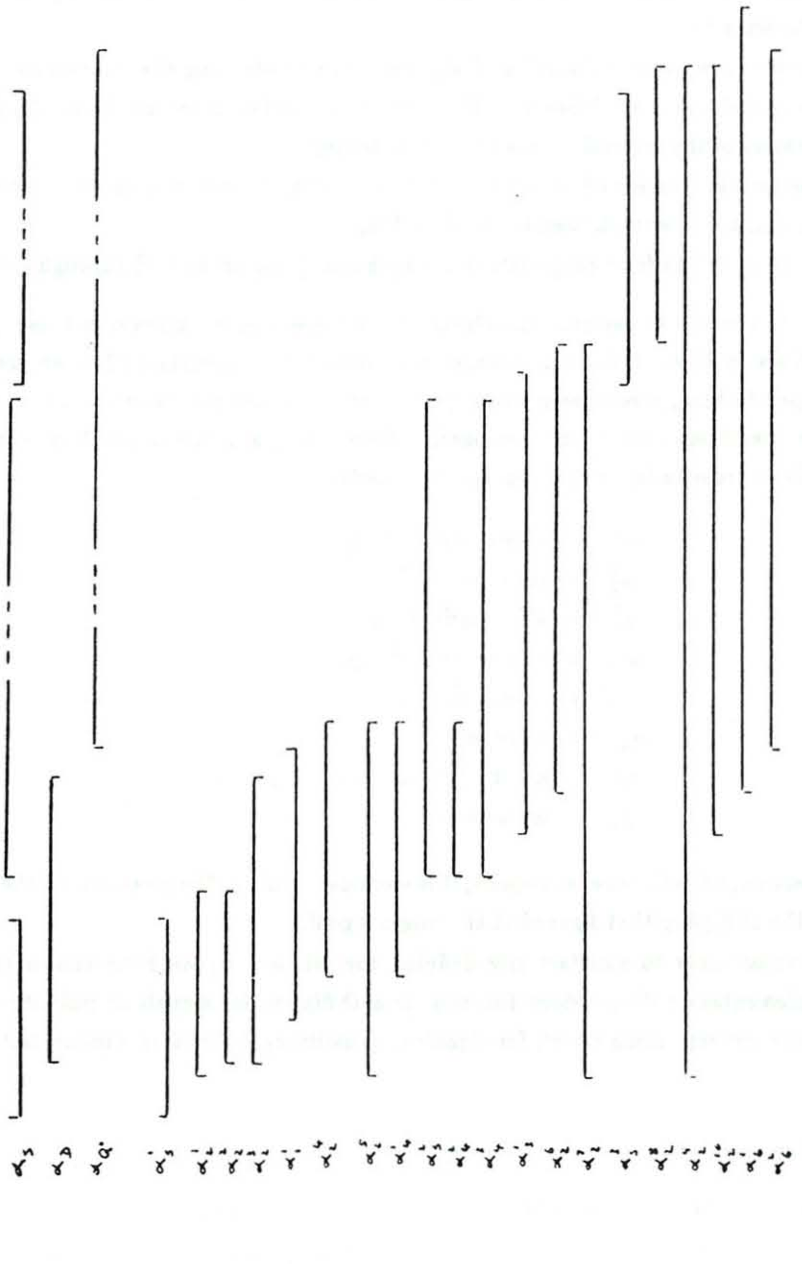

Fig 12 Assertions used in proving the D-flipflop implementation theorem.

(b) Both assertions have the same polarity (in this case, both being *high*), and

(c) The intervals on which the two assertions are defined overlap (or, more precisely, we *require* them to overlap, and, as a result, generate (as further subgoals) constraints on the timing relation, R).

α_4^1 This assertion results from propagating assertion α_1^1 through gate g_4.

α_S^2 This assertion is simply the second conjunct of assertion α_S (one of the hypotheses) on the strobe waveform.

α_4^2 This assertion is a weakened version of α_4^1, obtained by restricting the interval over which the latter is defined to a subinterval. This subinterval is chosen so that its start coincides with the start of the interval over which α_S^2 is defined.

α_4^3 This assertion is established using a variant of the NAND-PAIR lemma (this time, with gates g_3, g_4), together with assertions α_S^2 and α_4^2.

α_3^1 This assertion results from propagating the *high* assertions α_S^2 and α_4^3 through gate g_3.

Whilst we could proceed to describe the remainder of the wavespec diagram in these term, it would be tedious to do so. Indeed, the reader may already have perceived that we are using, in a stereotyped fashion, a relatively small number of techniques (ie, tactics). Thus, we can express these operations rather more succinctly. Here (using a notation which is reasonably self-evident) is the remainder of this sequence of assertions.

$$\begin{aligned}
\alpha_2^6 &= \mathit{propagate}\quad \alpha_3^1 \quad g_3 \\
\alpha_2^7 &= \mathit{join}\quad \alpha_2^5 \quad \alpha_2^6 \\
\alpha_S^3 &= 3^{rd}\ \mathit{conjunct}\quad \alpha_S \\
\alpha_2^8 &= \mathit{propagate}\quad \alpha_S^3 \quad g_2 \\
\alpha_2^9 &= \mathit{join}\quad \alpha_2^7 \quad \alpha_2^8 \\
\alpha_2^{10} &= \mathit{weaken}\quad \alpha_2^9 \\
\alpha_6^1 &= \mathit{nand\_pair}'\quad \alpha_3^1 \quad \alpha_2^{10} \quad g_5 \quad g_6 \\
\alpha_6^2 &= \mathit{weaken}\quad \alpha_6^1
\end{aligned}$$

This final assertion, α_6^2 is the one we require; it is identical with α_Q, the assertion on the output waveform of the D-flipflop that figured in the original goal.

Thus, (subject only to satisfactorily defining the relation R) we have shown that the proposed implementation does indeed function as a D-flipflop for signals of polarity c_t. We remark that the corresponding proof, for signals of polarity c_f, is broadly similar, but a little more complex.

7.4 The Timing Relation

As already described, the end result of undertaking the goal-directed proof construction exercise is not only to prove the D-flipflop theorem, but also to assist in determining a suitable value for R, the timing relation. The actual value that was chosen (and proven correct) for R was

$$
\begin{aligned}
(R \quad \vec{a} \quad \vec{n}) \;=\; & a_1 > 2n_1 + 2n_3 \\
& a_1 \geq a_5 + n_1 \\
& a_3 > 2n_1 + 2n_3 \\
& n_4 + n_2 \geq a_4 + n_1 \\
& a_5 > 2n_1 + n_3 \\
& a_6 > n_1 + n_3 \\
& a_7 \geq n_1 + 2n_3 \\
& n_4 + n_2 \geq a_4 + a_8 \\
& n_2 > 0
\end{aligned}
$$

By way of illustration, we present below a typical 'model' of this relation (ie, a set of values satisfying it). The values we have chosen are representative of a state-of-the-art CMOS gate-array technology.

Timing Parameters for the NAND gate

n_1	=	300 *ps*	low-to-high propagation delay
n_2	=	200 *ps*	high-to-low hold time
n_3	=	300 *ps*	high-to-low propagation delay
n_4	=	200 *ps*	low-to-high hold time

Timing Parameters for the D-flipflop

a_1	=	1400 *ps*	min strobe low time
a_2	=	300 *ps*	max strobe rise time
a_3	=	1300 *ps*	min strobe high time
a_4	=	100 *ps*	max strobe fall time
a_5	=	1000 *ps*	data setup time
a_6	=	700 *ps*	data hold time
a_7	=	900 *ps*	max output propagation delay
a_8	=	300 *ps*	min output hold time

It is of interest to note that

- The 'strobe rise time' parameter, a_2, is (in the definition of R) completely unconstrained: thus, the D-flipflop turns out not really to be 'edge triggered' at all!
- But, conversely, the 'strobe fall time' parameter, a_4, is very strongly constrained; thus, the ability of the D-flipflop to retain data is *critically* dependent upon the rapidity of the trailing edge of the strobe waveform.

So far as we know, this latter fact has not been previously remarked upon.

Functional relationship Although generally there are advantages in describing (as we have done here) the relation between $\vec{a}$ and $\vec{n}$ *relationally*, for some purposes there are advantages in strengthening this relation to a *functional* one. This may take either the form

$$\vec{n} = (F \quad \vec{a}) \qquad \text{or} \qquad \vec{a} = (G \quad \vec{n})$$

The first form specifies the parameters required of the NAND gates in order to realise a D-flipflop with given parameters, whilst the second specifies the parameters of the D-flipflop in terms of those of its component NAND gates (corresponding to, respectively, a top-down *versus* a bottom-up approach). In general, neither F nor G will be total functions.

7.5 Some Statistics

In order to allow the reader to gain an appreciation of the size of the proof of the above theorem, and of the time taken in its creation, we provide a brief summary of the main statistics:

- The overall proof involves about 1600 'high level' inferences (ie, applications of derived inference rules) each corresponding to the selection by the user of a particular tactic.
- These high-level inferences are mapped into about 5000 applications of the 'primitive' inference rules.
- The construction of this proof took about one week of concentrated work by a practiced user.

Note that these figures apply to the proof of the D-flipflop theorem itself; they do not include the proofs of the various lemmas used.

8. Conclusion

The conclusions we draw from the VERITAS project are severalfold; we group them together under various headings. Firstly, as concerns the use of typed, higher-order logic as a computer hardware description language:

- The fact that it allows *predicates* to be introduced means that *partial* (as distinct from *total*) descriptions may be expressed.
- The fact that it allows *higher-order* entities to be introduced means that concepts such as, for example, functional composition or induction, may be expressed in a natural, intuitive style.
- The fact that it has an associated *deductive calculus* means that reasoning ('formal inferencing') can be carried out.
- The fact that higher-order logic is powerful enough to describe mathematics itself means that this reasoning can encompass not just the 'low-level' aspects of the behaviour of digital systems, but also abstract aspects of their behaviour. For example, the properties of number-theoretic transforms based on finite-field arithmetic could be verified.
- Finally, we note that higher-order logic includes as proper subsets both the *applicative programming languages* and also PROLOG. Both of these formalisms have been used for describing aspects of digital systems.

Overall, we would claim that typed, higher-order logic (and syntactic variants thereof) is *the* natural computer hardware description language.

Next, we offer some comments on the nature of low-level digital behavioural specifications.

- These should *always* be partial rather than total. This is because the physical world is inherently indeterminate and our knowledge of it is necessarily imprecise. Any *exact* description must therefore be a wrong one!
- It turns out that the 'temporal' and the 'functional' aspects of low-level behavioural specifications are *inextricably* bound together.
- The predicate *const*, that asserts of an *analogue* waveform that, over a given interval of time, it is constant at a given *digital* state, is of fundamental importance in specifying the behaviour of digital devices.

Our general conclusions on using higher-order logic for formal verification are as follows.

- 'Formal verification', in this context, essentially involves deriving (by the use of formal inferencing, from a set of axioms) a theorem that asserts that the waveforms at the ports of a system satisfy a given property (the behavioural specification of the overall circuit) provided that the waveforms at the ports of each component device of the system satisfy the behavioural specification of that device.
- The process of 'formally verifying' a design is much harder work than is the 'simulation' of the operation of the same device. Unlike simulation, however, the solution obtained from formal verification is valid for *all possible* waveforms and for *all possible* timing parameters.
- Because of this, the theorems expressing the results of formal verification are 'reuseable' items. In particular, this allows the verification of hierarchically structured systems to be undertaken in a structured manner.
- In practice, it is necessary to use *computational assistance* for formal verification in order to eliminate the possibility of wishful thinking, to automate routine patterns of inferencing, and to maintain a secure database of lemmas.
- We have found that an approach to theorem proving derived from the Edinburgh ML/LCF project, in which terms, derivations and signatures are represented as values of secure data abstractions, to be ideal in all respects.

Finally, the conclusions we draw from the particular case study presented here (ie, the specification and verification of a D-flipflop), are as follows:

- When examined in detail, the operation of the proposed implementation of the D-flipflop (which, at first sight appears to be a relatively simple circuit) turns out to be surprisingly complicated.
- The practicability of formally verifying such a circuit has, however, been demonstrated, as also has the practicability (we should rather say the necessity !) of carrying out this task computationally.
- As we have shown, the timing relation R, which relates the timing parameters of the component devices to the timing parameters of the overall circuit, can best be determined by attempting to construct a proof of the verification theorem, gathering together all the 'dangling' subgoals that remain, and then choosing a value of R that is both *strong* enough

to enable the proof to be completed and yet *simple* enough to be of practical utility.

- Finally, in comparing this case study with the many accounts that have appeared in the literature concerning the verification of digital systems, particular note should be taken of the fact that in the approach we have demonstrated:
  - the specifications of both the D-flipflop and its component NAND gates are *partial* ones, and
  - the timing parameters of the component NAND gates have been taken as *variables*, rather than as arbitrary numerical values (eg, unit delays).

  Whilst both these consideration do, of course, greatly complicate the exercise, they are, we believe, the only means by which results of practical use can be obtained.

In overall conclusion, we believe that higher-order logic has been demonstrated to provide an effective basis both for specifying the intended behaviour of digital systems and for formally verifying the correctness of proposed implementations.

Acknowledgment The VERITAS project on digital specification and verification is currently supported under grant GR/C/2287.5 from the UK Science and Engineering Research Council.

9. References

[1] Gordon, M., Milner, R., and Wadsworth, C., "Edinburgh LCF", Springer-Verlag, 1979.

[2] Hanna, F.K., "VERITAS: The Axioms", Internal Report, University of Kent, 1981.

[3] Hanna, F.K., "Overview of the Veritas Project", Internal Report, University of Kent, 1983.

[4] Hanna, F.K., and Daeche, N. "Specification and Verification using Higher-Order Logic", p418, Proc CHDL 1985.

[5] Hanna, F.K., and Daeche, N. "The VERITAS Theorem Prover and its Formal Specification", Internal Report, University of Kent, 1984/5.

[6] Shostak, R.E., "Formal Verification of Circuit Design", p13, Proc CHDL, 1983.

[7] Gordon, M., "Why higher-order logic is a good formalism for specifying and verifying hardware", Internal Report no 77, University of Cambridge, 1985.

[8] Eveking, H., "The Application of CHDL's to the Abstract Specification of Hardware", p167, Proc CHDL 1985.

[9] Herbert, J., "The Application of Formal Specification and Verification to a Hardware Design", p434, Proc CHDL 1985.

[10] Texas Instruments, "Semiconductor Components, Data Book Two",

[11] Enderton, H.B., "A Mathematical Introduction to Logic", Academic Press, 1972.

DISCUSSION

Dr. Schneider questioned the use of the work 'higher-order-logic'. Dr. Hanna answered, that he had met very few logicians who complain. Dr. Schneider responded , "I am not a logician!"

Professor Randell remarked that he thought it odd that it could be possible to ally the notion of 'absolute correctness' (a very precise notion) so that of 'small and simple system' (a not very precise notion).

Dr. Hanna was asked, whether he could be sure that the circuit is not oscillating? Dr. Hanna answered, that we know that the physical circuit works. We are using weak assertions and those satisfy our axioms.

Professor Randell asked, Dr. Hanna whether in his efforts to prove existing designs, had he been able to find errors in them? Dr. Hanna answered, that he had not tried many examples but that he imagined if we did we would. Most designs he had seen were under-specified if one takes a close look.

Professor Lewin said that he knew of flip-flop designs with race conditions that cause both outputs to go high simultaneously. Could Dr. Hanna pick that up? Dr. Hanna answered, that in this case the theorem proven could not come up with an answer.