Strongly-typed Theory of Structures and Behaviours

Keith Hanna and Neil Daeche

University of Kent, Canterbury, Kent, CT2 7NT, United Kingdom

Abstract. This paper describes an approach to capturing the relation between circuits and their behaviours within a formal theory. The method exploits dependent types to achieve a rigorous yet theoretically simple connection between circuits (treated as graphs) and their behavioural specifications (treated as predicates). An example is given of a behavioural extraction function and it is shown how a type for modules can be defined that is sufficiently fine to guarantee that the behaviour of a module will satisfy its behavioural specification. The method is discussed in relation to VHDL and in relation to formal synthesis, (a process whereby one starts with a behavioural specification and, using an interactive goal-directed approach, ends up with a circuit and a formal proof that it satisfies the given behavioural specification).

1 Introduction

The aim of hardware verification is to establish that the behaviour of a given circuit satisfies a given behavioural specification. The problem divides naturally into two tasks:

- Firstly, given a (well-formed) circuit and given the behavioural specifications of each of its component parts, determining the behavioural specification of the overall circuit, and
- Secondly, showing that this computed behavioural specification is stronger than the given behavioural specification.

With most approaches to formal verification the structure of the circuit is represented only *informally* (for example, as a circuit diagram) and so this means that the first of the above tasks can only be undertaken informally. For many purposes, for example, when working with an idealised, two-valued, voltage-driven digital logic, this approach is perfectly adequate. This is because the *intension* (or form) of the term whose *extension* (or value) describes a (well-formed) circuit's behavioural specification corresponds closely to the structure of the circuit. Briefly stated, the behavioural specification of the overall circuit (described as a predicate on the signals at the circuit's ports) may be obtained by taking the conjunction of the terms that describe the behavioural specifications of the individual components and using existential quantification to hide any internal signals. Since the relation between structure and behaviour is so simple, there is little opportunity for error in inferring the relationship and so little is lost by adopting an informal approach. With other kinds of digital logic (for example, tri-state logic) and with circuit structures that are parametrized, then the relation between structures and behaviours is rather more complex and there are significant gains to be had in formalising the relationship.

In this paper we will be describing an approach to relating structure and behaviour that exploits the very fine type structure that a *dependently-typed* higher-order logic offers. (Dependent types are explained below.) As we shall show, the particular merits of the approach are its rigorous natures and its relative theoretical simplicity. In order to place it in the context of conventional CAD methods, we will describe it using concepts and terminology taken from VHDL (such as *entity*, *interface* and *implementation*). Later, we will show how it relates to aspects of VHDL and how it can be mechanically translated to and from VHDL.

2 Background

As background to this paper we briefly[1] review two earlier approaches (with which this paper shares many features) to the problem of relating structure and behaviour within a formal framework.

The principal features of the first approach [HD86] are:

- The formalism used is a typed higher-order logic.
- Circuits (in general, hierarchically structured) are *directly* specified within the logic in terms of their essential properties.
- The type discipline of the logic is used to enforce the well-formedness of circuits.
- There is a *formal* link between the structure of circuits and their behaviour.

Whilst this approach is both direct and simple, it does, however, have some serious limitations, namely:

- Circuits are specified by *properties* and not as *values*. This is generally inconvenient since often an exact (ie, 'categorical') description of a circuit is required rather than a 'loose' one.
- The approach is valid only for idealised, voltage-driven 2-state logic.
- Since the theory is axiomatic in nature, errors in specifications may give rise to inconsistency.

A very much more ambitious approach to relating structure and behaviour is described in [BHY92]. The principal features of this second approach are:

- A *module* (that is, a circuit) is represented by a constant, basically a list comprising the module name, lists of the names of its input ports and its output ports, and an occurrence list specifying the internal structure of the module.
- These datastructures are similar in form to the abstract syntax of the representation of the corresponding modules in a conventional hardware description language.
- Tri-state logic is catered for.
- Generic circuit modules (for example, modules parametrized by their word length) are represented by functions that, when applied to a concrete value of the generic parameter, yield a concrete module representation.

[1] A version of this paper giving a much fuller review of these two earlier approaches is available from the authors.

- The formalism used is the Boyer-Moore logic (a weakly typed, first-order system with a LISP-like syntax).

Whilst the approach has been used to conduct some impressive, large-scale correctness proofs, it is, however, not without some disadvantages. Perhaps the three main ones are:

- the complexity of its structure-to-behaviour mapping function, DUAL-EVAL (which results in a general lack of transparency and in increased complexity of proofs);
- the absence of strong typing (which makes descriptions of objects and functions verbose and difficult to follow);
- the absence of a user-accessible metalanguage by which deduction in the Boyer-Moore system may be directed.

It was, nevertheless, the description of this general approach and of the possibilities it opened up which reawakened our interest in formally linking structures and behaviours. The method described in the following sections inherits features from both this approach and from the earlier one outlined above.

3 Dependent Types

The main theme of this paper is the way in which a *dependently-typed* logic may be used to allow the structure and behaviour of circuits to be intimately related to each other within a strongly-typed formal logic. The concepts we describe were actually tested using the VERITAS logic and so we begin by summarising, very briefly, its main features. We note, however, that the same approach could equally well be developed in other dependently-typed logics such as NUPRL [C86] or ISABELLE [PT90] and even (though with some limitations) by modelling a dependently-typed logic within an ordinary polymorphically-typed one, as in [JM92].

VERITAS [HDL90, HD92] is a higher-order logic whose distinctive feature is that its type structure includes dependent types, subtypes and datatypes. It is computationally implemented [HDH92] and the notation used in this paper is, with a few minor exceptions, identical to that which the logic both accepts and generates. In the remainder of this section we develop the various definitions and concepts we shall need later on.

Datatypes The natural numbers, *nat*, may be introduced as a *datatype* by

$$\textbf{datatype}\ nat = 0 \mid nat'$$

Here, the so-called successor function has been introduced as a postfixed operator (prime). Thus, the first few numbers may be defined by $1 \triangleq 0'$, $2 \triangleq 1'$, and so on.

All terms in the logic are total. Function may be defined by primitive recursion over datatypes. For example, the addition function (defined as a binary operator '+') may be defined by

$$\textbf{define}\ n + 0 = n \mid n + m' = (n + m)'\ \textbf{end}$$

Functions may also be defined by abstraction as for example in $twice\ n \triangleq n + n$.

Subtypes may be defined by specifying their characteristic predicate. For example, the subtype *bit* of *nat* may be defined as

$$bit \triangleq \{n{:}\, nat \mid n < 2\}$$

Types Types are themselves terms and hence functions may take types as arguments or yield types as their result. For example, the function N defined by $N\ n \triangleq \{m{:}\, nat \mid m < n\}$ takes a number n and yields the subtype of *nat* that comprises numbers in the range 0 to $n-1$. Since types are terms, they themselves have types. The type of all ordinary types is $U0$, the universe of (small) types.

Terms in the logic do not have unique types — rather, the term formation rules of the logic specify that certain term-type combinations (sometimes called 'judgments') are well-formed. For example, all of the following typed terms are well formed: $1{:}\, nat$, $1{:}\, bit$, $1{:}\, N\ (2+2)$.

Vectors Vectors, that is, one-dimensional arrays, may be represented as functions over their index types. For example, the type *byte* defined by $byte \triangleq N\ 8 \rightarrow bit$ describes the type of 8-bit binary words. If B is a symbol of type *byte*, then the elements of B may be accessed by application, thus $B\ 0$, $B\ 1$, up to $B\ 7$. Application in VERITAS may alternatively be denoted by subscription, and so the same collection of bits may also be expressed as B_0, B_1, up to B_7.

Whilst vectors can easily be defined by λ-abstraction, it is convenient to be able to describe literal vectors using conventional mathematical notation. Thus, we will often write[2], for instance, either $v \triangleq \begin{bmatrix} a \\ b \\ c \end{bmatrix}$ or $v \triangleq \langle a, b, c\rangle$ to describe the vector v whose components are $v_0 = a$, $v_1 = b$, $v_2 = c$.

Dependent Types Dependent types [C86] are a natural generalisation of the ordinary × and → types that allow finer, more expressive types to be defined. A type of the form $[x{:}\, S] \rightarrow T_x$ is a *dependent function type* (sometimes called a Π-type). An application consisting of a function $f{:}\,([x{:}\, S] \rightarrow T_x)$ applied to a term $a{:}\, S$ is a term $f\ a$ whose type is T_a. Thus, the type of an application *depends on* the value of the argument. If, as a special case, the type T_x is independent of x, then an ordinary (i.e., non-dependent) function type results.

In a similar way, *dependent product types* may be defined. These have the form $[x{:}\, S] \times T_x$. A term of this type has two components; the first is of the form $a{:}\, S$ and the second (whose type depends on the value of the first) is of the form $b{:}\, T_a$. A typical example of a dependent product type is the type $[n{:}\, nat] \times N\ n$, and a typical element of this type is $(5, 3)$, since $5{:}\, nat$ and $3{:}\, N\ 5$.

Records In addition to vectors whose elements are all of the same type one can also, by using dependent types, define vectors whose elements are of *heterogeneous* types. Such vectors we term *records.* Suppose that $f{:}\,(N\ n \rightarrow U0)$ is the function that defines the type f_i of the i^{th} element of a record with n components. Then the type of the overall record is $[i{:}\, N\ n] \rightarrow f\ i$. Notice that this type itself is rather similar[3] to a vector and so we introduce the notation $\Pi\langle \ldots \rangle$ to abbreviate such types. For

[2] The "vertical" form of vector notation is not at present supported by computational implementations of the logic.

[3] Technically, the only difference is that it contains a Π binder rather than a λ one.

example, given three typed terms, $a{:}\,s$, $b{:}\,t$ and $c{:}\,u$, the type of the record $\langle a, b, c\rangle$ may be written as $\Pi\langle s, t, u\rangle$.

4 Relating Structures and Behaviours

Our aim is to be able to describe, with clarity and precision, the structures of circuits, the behaviour of circuits and the relation between structures and behaviours. We will do this by developing, within the VERITAS logic, a strongly-typed Theory of Structures and Behaviours (abbrev. TSB).

The relation between structures and behaviours is, of course, strongly influenced by the particular kind of digital technology by which the circuit is implemented. In order to illustrate the general principles in as simple a setting as possible, we shall describe the development of the theory for an ideal *two-level, voltage-driven, fully synchronous* technology. We note, however, that the real gains from this approach only become manifest when working with non-ideal, tri-state technologies. We have developed the theory for such a technology; it is alluded to in a later section.

Logic levels We will take the two logic levels to be identified with the values 0 and 1 of type *bit*. We will assume that all inputs are two-level signals and that all latches are in a two-level state at time $t = 0$. Given these assumptions, and under the further assumption that the circuit is *well-formed* (a property that we will be formally defining) then we can assume that all signal levels within the circuit remain as two-level ones. Throughout, we will identify time with the natural numbers, that is, $time \triangleq nat$.

Relation to VHDL So as to allow the TSB approach to be related to contemporary engineering practice, we will adopt certain features of VHDL[4] and, later, we shall describe how TSB may be translated into a dialect of VHDL. The ability to undertake such translation is essential [B92] if a formal method is ever to be able to be used in industrial design.

Following the general VHDL approach, we shall work in terms of *design entities* (or *entities* for short) which we will consider as having two aspects, an *interface* that describes an entity's external aspects, and an *implementation* that describes its internal structure. We shall extend VHDL, however, by requiring the interface to describe not only the entity's *structural* aspects but also its *behavioural specification*.

4.1 Interfaces

The interface that an entity (in a voltage-driven technology) presents to the external world consists of a set of *ports* each of which has a *mode* (that specifies whether it is an input or output port) and has a *base type* and which carries a *signal*. If the base type of the port is b, then the signal at the port will be of type $time \rightarrow b$. The behaviour of the entity is characterised by a predicate defined on the signals at its ports.

[4] VHDL is a widely used hardware description language [VHDL87] distinguished by a syntax that is somewhat prolix and a semantics that whilst not particularly expressive is nevertheless, when viewed formally, found to be of Byzantine complexity ...

We formalise the notion of an interface by introducing a type *interface*. In framing the definition for this type we shall aim (by exploiting dependent types) to capture the exact relationship between, on the one hand, the number and types of the ports of the entity and, on the other hand, the type of its behavioural predicate. Formally, an entity interface is a 4-tuple consisting of:

- A number, np, specifying the number of ports. Given this number we shall assume that the ports are identified with elements of the set $N\ np$ and that we refer to ports by their *indices* rather than by their names. Thus if, for example, np had the value 4, then the ports would be labelled by indices 0 through to 3.
- A port-mode mapping, pm, from the port index set $N\ np$ to the two-element type *mode* defined by the declaration **datatype** $mode = input \mid output$.
- A port-type mapping, pt, from the port index set $N\ np$ to the type $U0$ of (small) types. This mapping will represent the types of the signal levels (eg, *bit*) at the ports. Thus, the type of the signal (as distinct from the type of the signal levels) at port 2 would be $time \rightarrow pt_2$.
- A behavioural specification, *spec*, that takes the form of a predicate over the signals at the ports of the entity. The definition we are about to present of the type of *spec* provides a particularly interesting illustration of the use of dependent types. Indeed, without dependent types it would not be possible to express this type at all. We build up the definition in stages. First, note that the signal at port i is of type $time \rightarrow pt_i$. Thus, if the signals at all of the np ports of the entity are bundled together to form a record (ie, a heterogeneously-typed vector), the type of this record is $[i\colon N\ np] \rightarrow (time \rightarrow pt_i)$. Thus, the type of the overall behavioural specification is

$$spec\colon ([i\colon N\ np] \rightarrow time \rightarrow pt_i) \rightarrow bool$$

Thus, the type *interface* of interfaces is given by the following dependent Cartesian product

$$\begin{array}{lll} interface \mathrel{\hat{=}} & [np\colon nat]\times & \text{Number of ports} \\ & (N\ np \rightarrow mode)\times & \text{Port modes} \\ & [pt\colon (N\ np \rightarrow U0)]\times & \text{Port types} \\ & (([i\colon N\ np] \rightarrow time \rightarrow pt\ i) \rightarrow bool) & \text{Behavioural predicate} \end{array}$$

To illustrate this definition, first, as a very simple example, consider the interface (Figure 1(A)) for a one-bit latch

$$\begin{array}{ll} latch\_intf\colon interface \mathrel{\hat{=}} & \\ \quad 2, & \text{Number of ports} \\ \quad \langle input, output\rangle, & \text{Port modes} \\ \quad \langle bit, bit\rangle, & \text{Port types} \\ \quad \lambda r\colon \Pi\langle(time \rightarrow bit), (time \rightarrow bit)\rangle. & \text{Behavioural predicate} \\ \qquad \forall t\colon time.\ (r_1)\ t' = (r_0)\ t & \text{(Recollect that } t' \text{ means } t+1) \end{array}$$

For a second example, this time illustrating a more diverse collection of types, consider the interface (Figure 1(B)) for an ALU unit.

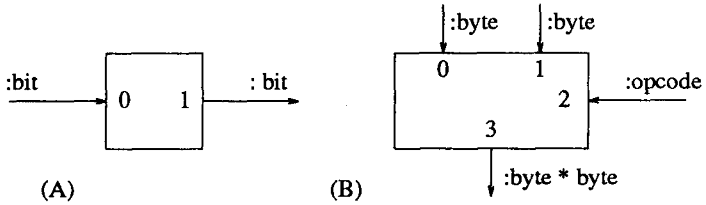

Fig. 1. Typical interfaces. (A) is for a 1-bit latch, and (B) is for a simple ALU.

Assume the declaration **datatype** *op_code* = *add* | *sub* | *mult* | *div*, and assume that a behavioural predicate *alu_behav* has been defined. The interface for the ALU is then defined as

$$\begin{aligned} &\mathit{alu\_intf}\colon \mathit{interface} \stackrel{\wedge}{=} \\ &\quad 4, \\ &\quad \langle \mathit{input}, \mathit{input}, \mathit{input}, \mathit{output}\rangle, \\ &\quad \langle \mathit{byte}, \mathit{byte}, \mathit{op\_code}, (\mathit{byte} \times \mathit{byte})\rangle, \\ &\quad \lambda r.\ \mathit{alu\_behav}\ (r_0, r_1, r_2, r_3) \end{aligned}$$

where the type of the record r is

$$r\colon \Pi\langle(\mathit{time} \to \mathit{byte}),\ (\mathit{time} \to \mathit{byte}),\ (\mathit{time} \to \mathit{op\_code}),\ (\mathit{time} \to \mathit{byte} \times \mathit{byte})\rangle$$

5 Contexts

Before we can go on and describe circuits we need to have a type-secure way of describing the interfaces of the component entities from which particular circuits will be built. For this, we introduce the notion of a *context* — essentially a sequence of interfaces. It is convenient to represent a context by an ordered pair whose first element, *nk*, specifies the number of kinds of interface in the context and whose second element is a vector (of length *nk*) of interfaces. Thus, the type *context* is defined as

$$\mathit{context} \stackrel{\wedge}{=} [\mathit{nk}\colon \mathit{nat}] \times (N\ \mathit{nk} \to \mathit{interface})$$

A typical example of a context (which we shall adopt as our standard context for other examples) is

$$\mathit{std\_cxt} \stackrel{\wedge}{=} 4, \langle \mathit{not\_intf}, \mathit{and\_intf}, \mathit{or\_intf}, \mathit{latch\_intf}\rangle$$

The first element of this pair specifies that the context contains some 4 interfaces; the second element (a vector) of the pair specifies the actual interfaces themselves. As usual, the elements of the vector are accessed by application. For example, the 0^{th} element (a NOT-gate interface) of the context would be referenced by the application $(\mathit{snd}\ \mathit{std\_cxt})\ 0$.

6 Circuits

We use the term *circuit* to mean a set of components connected, via their ports, to a set of *nodes*. Circuits are thus essentially labelled, directed graphs. Figure 2 shows a typical circuit.

We formalise the notion of a circuit by introducing a type *circuit*. As with the definition of the type *interface* above, we aim, by the use of dependent types, to capture the notion with precision. Formally, we define a circuit to be a 6-tuple consisting of:

- A context, $cxt\colon context$, that specifies the set of component interfaces from which the circuit is constructed. We introduce the following subsidary definitions:
  - The number of interface kinds in the context, $nk\colon nat$, defined by $nk \mathrel{\hat{=}} fst\ cxt$.
  - The type, $kind\colon U0$, of interface kinds, defined by $kind \mathrel{\hat{=}} N\ nk$.
  - The kind to interface function, $ki\colon kind \rightarrow interface$, that maps an interface kind to an actual interface, defined by $ki \mathrel{\hat{=}} snd\ cxt$.
- A number, $nc\colon nat$, that specifies the number of components in the circuit.
- A number, $nn\colon nat$, that specifies the number of nodes in the circuit. We introduce the following subsidary definitions:
  - The type $comp\colon U0$ of components, defined by $comp \mathrel{\hat{=}} N\ nc$.
  - The type, $node\colon U0$, of nodes, defined by $node \mathrel{\hat{=}} N\ nn$.
- A function, $ck\colon comp \rightarrow kind$, that specifies the kind of a component. We introduce the following subsidary definition:
  - The component to port-type function, $cp \mathrel{\hat{=}} comp \rightarrow U0$, defined by $cp \mathrel{\hat{=}} N \circ fst \circ ki \circ ck$.
- A function, $nt\colon node \rightarrow U0$, that specifies the node types.
- A function $w\colon [c\colon comp] \rightarrow cp\ c \rightarrow node$ that specifies the wiring of the circuit.

Thus, the type *circuit* is given by the following dependent Cartesian product:

$$\begin{array}{lll}
circuit \mathrel{\hat{=}} & [cxt\colon context]\times & \text{Context} \\
& \mathbf{let}\ nk = fst\ cxt\ \mathbf{in} & \\
& \mathbf{let}\ kind = N\ nk\ \mathbf{in} & \\
& \mathbf{let}\ ki = snd\ cxt\ \mathbf{in} & \\
& [nc\colon nat]\times & \text{Number of components} \\
& [nn\colon nat]\times & \text{Number of nodes} \\
& \mathbf{let}\ comp = N\ nc\ \mathbf{in} & \\
& \mathbf{let}\ node = N\ nn\ \mathbf{in} & \\
& [ck\colon comp \rightarrow kind]\times & \text{Component kind function} \\
& \mathbf{let}\ cp = N \circ fst \circ ki \circ ck\ \mathbf{in} & \\
& [nt\colon node \rightarrow U0]\times & \text{Node types} \\
& ([c\colon comp] \rightarrow cp\ c \rightarrow node) & \text{Wiring}
\end{array}$$

This type definition is not nearly as complicated as it may, at first sight, seem! For example, the TSB representation of the circuit shown in Figure 2. is given by the tuple (cxt, nc, nn, ck, nt, w), where

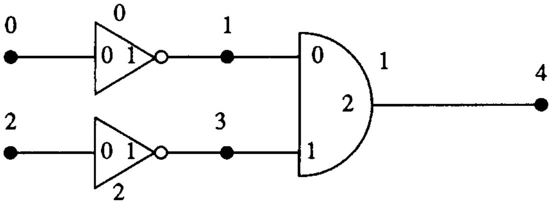

Fig. 2. A circuit consisting of two NOT gates connected to an AND gate.

- *cxt* is the standard context *std_cxt* that contains, in order, the interfaces for a NOT gate, an AND gate, an OR gate and a latch.
- *nc*, the number of components in the circuit, is 3.
- *nn*, the number of nodes in the circuit, is 5.
- *ck*, the component kind map, is the vector $\langle 0, 1, 0\rangle$ which indicates that the first component in the circuit is a NOT gate, the second an AND gate, and the third a NOT gate.
- *nt*, the node type function that specifies the type of each node in the circuit, is the vector $\langle bit, bit, bit, bit, bit\rangle$.
- *w*, the wiring function, is the array (i.e., a vector of vectors)

$$\begin{bmatrix} \langle 0,1\rangle \\ \langle 1,3,4\rangle \\ \langle 2,3\rangle \end{bmatrix}$$

This array describes the way that the ports of each component are connected to the nodes. In this case it indicates that port 0 of component 0 is connected to node 0 and port 1 to node 1, and that port 0 of component 1 is connected to node 1, port 2 to node 3, and so on.

Thus, the TSB representation, *cir*, for the circuit in Figure 2 is simply

$$cir \triangleq \langle std\_cxt, 3, 5, \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} bit \\ bit \\ bit \\ bit \\ bit \end{bmatrix}, \begin{bmatrix} \langle 0,1\rangle \\ \langle 1,3,4\rangle \\ \langle 2,3\rangle \end{bmatrix} \rangle$$

As a second example, one with a richer set of types, consider the ALU circuit shown in Figure 3. If we assume the existence of a context, *cxt*, containing interface definitions for the three components then the description of the circuit as a TSB value is

$$alu\_circuit \triangleq \langle cxt, 3, 7, \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \begin{bmatrix} byte \\ byte \\ byte \times byte \\ byte \times byte \\ bit \\ bit \\ op\_code \end{bmatrix}, \begin{bmatrix} \langle 0,1,4,5,2 \rangle \\ \langle 4,5,6 \rangle \\ \langle 2,3 \rangle \end{bmatrix} \rangle$$

(where a, b and c are the indices of the three components in the context cxt).

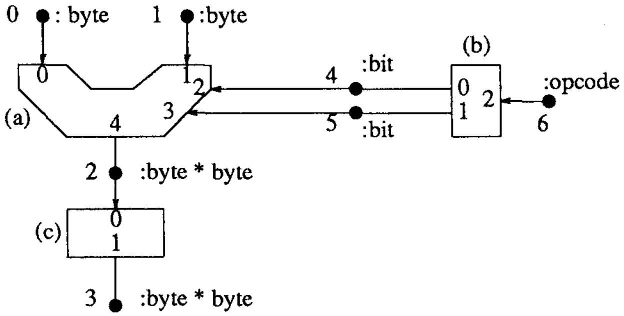

Fig. 3. An ALU circuit.

7 Well-formed Circuits

So far, the definition of the type *circuit* has ensured that the circuits being described are well-formed in a graph-theoretic sense. However, before we can reason about the behaviour of circuits, we need to ensure that they are also well-formed with respect to a particular technology.

For the present case (a two-state, voltage-driven synchronous technology) this involves establishing that:

- Each node is driven by exactly one output port.
- The type of each port is identical to the type of the node to which it is connected.
- There are no asynchronous loops (that is, if all latches are removed, then the remaining circuitry is purely combinational).

Predicates may be defined to check each of these conditions. For example, a good way to test the first condition is to assert the existence of a 'back-wiring' function, $bw\colon node \longrightarrow [c\colon comp] \times cp\ c$ such that, for any node n, the value of $bw\ n$ is a pair (c, p) defining a component c and a port p on that component with the property that: (a) the port p is an output port, and (b) that port is connected (as specified by the wiring function w) to node n. Specified formally, this predicate, $\mathit{well\_driven}\colon \mathit{circuit} \rightarrow \mathit{bool}$ is defined by the expression

$$\begin{array}{ll}
\exists bw\colon \mathit{node} \rightarrow [c\colon \mathit{comp}] \times \mathit{cp}\ c. & \\
\quad \forall n\colon \mathit{node}. & \\
\qquad \mathbf{let}\ (c; p) = bw\ n\ \mathbf{in} & \text{Component and port} \\
\qquad \mathbf{let}\ \mathit{intf} = (\mathit{snd}\ \mathit{cxt})\ (\mathit{ck}\ c)\ \mathbf{in} & \text{Find component interface} \\
\qquad \mathbf{let}\ (\mathit{np}; \mathit{pm}; \mathit{pt}; \mathit{spec}) = \mathit{intf}\ \mathbf{in} & \text{Find port-mode function} \\
\qquad (\mathit{pm}\ p = \mathit{output}) \wedge (w\ c\ p = n) & \text{Check conditions}
\end{array}$$

(where the various subsidiary definitions (such as *node*, *comp*, *cp*, etc) are exactly as defined earlier).

Checking the second of the above conditions (*viz*, that interconnected ports and nodes have identical types) is trivial. Checking the third condition (absence of asynchronous loops) is a more interesting task. A good way to test for this property is to assert the existence of a labelling (using elements of *nat*) on the nodes such that no asynchronous component (that is, any component apart from a latch) has an input port connected to a node labelled with a number greater than or equal to the number labelling a node to which any of its output ports are connected. Such a predicate is not difficult to define.

Given the collection of well-formedness predicates for the technology, we define a subtype *wf_circuit* of the type *circuit* that comprises only those circuits that are deemed to be well-formed

$$\mathit{wf\_circuit} \mathrel{\hat{=}} \{c\colon \mathit{circuit} \mid \mathit{well\_formed}\ c\}$$

8 Behavioural Extraction

The real payoff from the TSB approach with its use of dependent types comes when we consider the task of relating a circuit's behaviour to its structure. Informally, we wish to define a function *behaviour* of type $\mathit{wf\_circuit} \rightarrow \mathit{behav\_spec}$ where the type *behav_spec* signifies a predicate over the record of signals at the nodes of the circuit. Thus, to a closer degree of approximation we require the function to have the type $\mathit{wf\_circuit} \rightarrow \mathit{record\_of\_signals} \rightarrow \mathit{bool}$. Formally, however, we need to capture the relation between the collection of signal base types (as described within *wf_circuit*) and the type *record_of_signals*. We achieve this by giving *behaviour* the dependent type

$$\begin{array}{l}
\mathit{behaviour}\colon \\
\quad [(\mathit{cxt}; \mathit{nc}; \mathit{nn}; \mathit{ck}; \mathit{nt}; w)\colon \mathit{wf\_circuit}] \rightarrow ([i\colon N\ \mathit{nn}] \rightarrow \mathit{time} \rightarrow (\mathit{nt}\ i)) \rightarrow \mathit{bool}
\end{array}$$

That is, it takes a well-formed circuit with nn nodes (whose base types are defined by the vector nt) and it yields a predicate over a *record_of_signals* type. For example, if the node base types for a two-node circuit were $\langle \mathit{bit}, \mathit{byte}\rangle$ then the individual signal types would be $(\mathit{time} \rightarrow \mathit{bit})$ and $(\mathit{time} \rightarrow \mathit{type})$ and so the type of this record would be $\Pi\langle(\mathit{time} \rightarrow \mathit{bit}), (\mathit{time} \rightarrow \mathit{byte})\rangle$.

The definition of the function is really surprisingly simple. We present it first and then discuss it line by line

$$\begin{array}{l}
\mathit{behaviour}\ (\mathit{cxt}; \mathit{nc}; \mathit{nn}; \mathit{ck}; \mathit{nt}; w) \mathrel{\hat{=}} \\
\quad \lambda r\colon ([i\colon N\ \ \mathit{nn}] \rightarrow \mathit{time} \rightarrow (\mathit{nt}\ \ i)). \\
\qquad \forall c\colon N\ \ \mathit{nc} \\
\qquad\quad \textbf{let}\ k = \mathit{ck}\ \ c\ \textbf{in} \\
\qquad\quad \textbf{let}\ \mathit{intf} = (\mathit{snd}\ \ \mathit{cxt})\ k\ \textbf{in} \\
\qquad\quad \textbf{let}\ (\mathit{np}; \mathit{pm}; \mathit{pt}; \mathit{spec}) = \mathit{intf}\ \textbf{in} \\
\qquad\quad \textbf{let}\ s\colon ([i\colon N\ \ \mathit{np}] \rightarrow \mathit{time} \rightarrow (\mathit{pt}\ \ i)) = \lambda j\colon N\ \ \mathit{np}.\ r\ (w\ \ c\ \ j)\ \textbf{in} \\
\qquad\quad \mathit{spec}\ \ s
\end{array}$$

The first line introduces a bound variable r, a record that indicates that the signal at node i is of type r_i. For example, if nt 0 were *bit*, then the signal at node 0 would be of type $\mathit{time} \rightarrow \mathit{bit}$. The next line, $\forall c\colon N\ \mathit{nc}$, specifies that the bound variable c ranges over all components in the circuit. The next line, $k = \mathit{ck}\ c$, defines k as being the interface kind of component c, and the following line defines the interface *intf* of that component. The next line splits *intf* into its four elements, namely *np* (number of ports), *pm* (port modes), *pt* (port types) and *spec* (behavioural predicate).

The penultimate line introduces s as being the record of signals *at the ports of component c*. It is obtained from r (the record of signals *at the nodes of the overall circuit*) by using the wiring function w to specify to which node that port j of component c is connected. The final line, *spec s*, asserts that this record of signals s must satisfy the behavioural specification *spec* (of component c).

A significant fact to notice about this definition is its entire definition is given here — it does not rely upon any subsidiary functions at all.

9 Implementations

In general, the internal structures of circuits are not visible; only their interface is seen. Thus we now need to discuss how to encapulate circuits to give interfaces. This involves no more than defining an association between the ports of the interface and the nodes of the circuit. Formally, we do this by introducing a new type, *implementation*, defined by

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathit{implementation} \mathrel{\hat{=}}} \\
\quad [(\mathit{np}; \mathit{pm}; \mathit{pt}; \mathit{spec})\colon \mathit{interface}] \times & \text{Interface} \\
\quad [(\mathit{cxt}; \mathit{nc}; \mathit{nn}; \mathit{ck}; \mathit{nt}; w)\colon \mathit{circuit}] \times & \text{Circuit} \\
\quad (N\ \ \mathit{np} \rightarrow N\ \ \mathit{nn}) & \text{Port to node wiring function}
\end{array}$$

That is, an implementation is an interface (with *np* ports), a circuit (with *nn* nodes) and a function associating ports with nodes. As with circuits, we need to define a subtype *wf_implementation* of well-formed implementations (essentially identical considerations as before apply) before we can reason about behavioural aspects of implementations.

9.1 Actual behaviour of an implementation

Given that an implementation is well-formed, we can infer its behaviour by using the following function, *behav*. This function is defined in terms of *behaviour* (the function that operates on well-formed circuits). Its definition is not particularly complex

$$
\begin{array}{l}
\mathit{behav}\ \ \mathit{impl} \mathrel{\hat{=}} \\
\quad \mathbf{let}\ (\mathit{intf}; \mathit{cir}; \mathit{pnw}) = \mathit{impl}\ \mathbf{in} \\
\quad \mathbf{let}\ (\mathit{np}; \mathit{pm}; \mathit{pt}; \mathit{spec}) = \mathit{intf}\ \mathbf{in} \\
\quad \mathbf{let}\ (\mathit{cxt}; \mathit{nc}; \mathit{nn}; \mathit{ck}; \mathit{nt}; w) = \mathit{cir}\ \mathbf{in} \\
\quad \lambda v\colon [m\colon N\ \ \mathit{np}] \rightarrow \mathit{time} \rightarrow \mathit{pt}\ \ m. \\
\quad\quad \exists u\colon [n\colon N\ \ \mathit{nn}] \rightarrow \mathit{time} \rightarrow \mathit{nt}\ \ n. \\
\quad\quad\quad (\mathit{behaviour}\ \ \mathit{cir})\ \ u \quad \wedge \\
\quad\quad\quad \forall j\colon N\ \ \mathit{np}.\ \ v_j = u\ \ (\mathit{pnw}_j)
\end{array}
$$

The first line of this definition splits the implemenation *impl* into its three components, namely *intf* (the interface), *cir* (the circuit) and *pnw* (the port-node wiring function). The next two lines split *intf* and *cir* into their components. The next line introduces a record v comprising the signals at the ports of the interface. The next line existentially quantifies (ie, "hides") the record u of signals at the nodes of the circuit. The penultimate line asserts that this record of signals satisfies the behavioural specification of the circuit (as given by the predicate (*behaviour cir*)). The last line asserts the identity of the external signals v with the internal ones, u, as seen through the port-node wiring function.

9.2 Entities

We now have *two* definitions for the behaviour of a (well-formed) implementation:

- Its *behavioural specification* (as given by the *spec* component of its interface);
- Its *actual behaviour* (as inferred by the above function, *behav*, from its internal structure).

In order to bring these two definitions into line, we introduce the notion of an *entity* (as a subtype of *implementation*). An entity is defined to be an implementation whose actual behaviour satisfies the behavioural specification of its interface. Thus we introduce the definition

$$
\mathit{entity} \mathrel{\hat{=}} \{\mathit{impl}\colon \mathit{implementation} \mid \mathit{well\_behaved}\ \ \mathit{impl}\}
$$

where the predicate *well_behaved* is defined by

$$
\begin{array}{l}
\mathit{well\_behaved}\ \ \mathit{impl} \mathrel{\hat{=}} \\
\quad \mathbf{let}\ (\mathit{intf}; \mathit{cir}; \mathit{pnw}) = \mathit{impl}\ \mathbf{in} \\
\quad \mathbf{let}\ (\mathit{np}, \mathit{pm}; [t; \mathit{spec}) = \mathit{intf}\ \mathbf{in} \\
\quad \forall v\colon [m\colon N\ \ \mathit{np}] \rightarrow \mathit{time} \rightarrow \mathit{pt}\ \ m. \\
\quad\quad \mathit{behav}\ \ \mathit{impl}\ \ v \Rightarrow \mathit{spec}\ \ v
\end{array}
$$

The first two lines of this definition extract the component parts of the interface, *intf*, the next line quantifies over the appropriate signal types and the final line asserts that the behaviour of the implementation satisfies the behavioural specification of the interface.

This definition for the type *entity* is perhaps the most important concept in this paper. This type describes both the structural and behavioural properties of an interface and it also describes the structural properties of an implementation whcih is guaranteed, by the type discipline of the logic, to satisfy the interface specifications. The interface type, by virtue of describing both structural and behavioural properties of an interface, provides the means to achieve a complete separation between the concerns of the implementor of an entity and the users of that entity.

9.3 Translation to VHDL

There is a sufficient degree of correspondence between this TSB notion of "entity" and the VHDL one to be able to expresss a TSB entity into a VHDL-liek notation. We have written a function (in STANDARD ML, the metalanguage in which VERITAS is defined rather than in VERITAS itself) that performs this translation. As an example, the translation of a (very simple) TSB entity (a NAND-gate realised using an AND gate and a NOT gate) into VHDL looks like:

```
entity nandgate is
  port (p1 : input bit, p2 : inputbit, p3 : outputbit);
  spec
    (∀t: time. (p3 t) = 1 − (p1 t) × (p2 t))
  end;
end nandgate;

architecture nand_impl of nandgate is
  signal p4 : bit;
begin
  G0; andgate portmap (p1, p2, p4);
  G1; notgate portmap (p4, p3);
end nand_impl;
```

(The alphameric names are specified as separate inputs to the translation function; the TSB notation does not at present include alphameric strings.)

10 Discussion

The development of the TSB approach has been described relative to an ideal, voltage-driven, two-state technology. Whilst this has allowed us to focus on the type-theory aspects of the approach, it also tends to somewhat trivialise it. Its full potential only becomes apparent when dealing with tri-state technologies and with parameterized circuitry (a point emphasised in [BHY92] in connection with their HDL notation).

We have developed a "fairly rigorous" TSB theory for tri-state technologies. Its main differences from the present approach is that the signals at the ports are no longer plain voltages but are instead (voltage, Thevenin-impedance) pairs, and are bi-directional in nature. In fact, there is no reason why the same approach should not also be used to reason about properties of completely analogue circuits.

All of the functions described in this paper have (with minor changes) been type-checked using a computational implementation of the VERITAS system. Doing this involved the development of many new "tactics" (functions used for guiding the process of type-checking and theorem proving) and their development was a far from trivial task. However, given these tactics, many definitions of the general kind described in this paper can now be type-checked almost entirely automatically.

As noted in [HLD89, F90] there are many advantages in combining the processes of design and of formal verification, yielding an approach known as *formal synthesis*. We have been exploring using the TSB approach with a formal synthesis design editor. Early signs are that the combination will be a fruitful one. The design process starts off with the proof editor being provided with a TSB context (*alias* VHDL library) and a TSB interface specification. The user, by interactively selecting *techniques* (*alias* tactics), refines the original interface into an interconnected set of simpler ones, and ultimately into primitive ones that can be found in the context. At each stage the proof editor generates the necessary theorems to justify the refinement. The end result of the process is a TSB entity — that is, an interface (identical to the original one) and an implementation whose structure and behaviour are guaranteed to satisfy the interface specification.

An important aspect of formal verification is the idea of relating behaviours at differing levels of abstraction. We believe that similar notions of abstraction will be equally valuable in the structural domain. For example, one may wish to relate the structural views of an adder at the level of individual bits and at the level of busses. There is also interesting work to be done in studying how structural and behavioural abstraction interact with each other.

The comparison between the present approach and its antecedents described earlier in this paper is an interesting one; features are inherited from both. Even if one concedes that the use of higher-order logic with dependent types does bring about aesthetic gains over the development of a similar approach in a simpler logic (for example, Boyer-Moore), it is still, in the authors' opinion, a very open question as to whether these theoretical gains can be translated into practical ones. Thus far, our own computational trials have been limited to very simple circuits and it is not clear how both the computational load and the amount of human guidance required will scale with increasing size of circuit.

As a final point, we note that it may be that the greatest gains arising from developing approaches such as the TSB one described here will not be in supporting formal verification but rather in providing the theoretical framework that will allow future generations of hardware design languages to be developed on a less *ad hoc* basis that at present. If this can be achieved, the gains could be significant.

Acknowledgments The research upon which this paper is based was carried out with sponsorship from International Computers Ltd, from Program Validation Ltd and from the UK Science and Engineering Research Council under Grant GR/F/3668.

References

[B92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, J. Van Tassel. *Experience with Embedding Hardware Description Languages in HOL*, pp 129-156, IFIP Trans. Vol A-10, *Theorem Provers in Circuit Design*, North Holland, 1992.

[BHY92] B. C. Brock, W. A. Hunt, W. D. Young, *Introduction to a Formally Defined Hardware Description Language*, pp 3-35, IFIP Trans. Vol A-10, *Theorem Provers in Circuit Design*, North Holland, 1992.

[C86] R. L. Constable, et al., *Implementing Mathematics with the Nuprl Proof Development System*, Prentice-Hall, 1986.

[F90] M. P. Fourman, *Formal System Design*, pp 191-235, in *Formal Methods for VLSI Design*, ed J. Staunstrup, North Holland, 1990.

[HD86] F. K. Hanna, N. Daeche *Specification and Verification of Digital Systems using Higher-order Predicate Logic*, pp 242-254, Proc IEE, Vol 133, Pt E, No 5, Sept 1986.

[HLD89] F. K. Hanna, M. Longley, N. Daeche, *Formal Synthesis of Digital Systems*, in pp. 153-169 L. J. M. Claesen (Ed), *Applied formal methods for correct VLSI design*, North-Holland, 1989.

[HDL90] F. K. Hanna, N. Daeche, M. Longley, *Specification and Verification using Dependent Types*, pp949-964, IEEE Trans on SE, Vol. 16, No. 9, Sept 1990.

[HDH92] F. K. Hanna, N. Daeche, G. Howells, *Implementation of the Veritas Design Logic*, pp 77-94, IFIP Trans. Vol A-10, *Theorem Provers in Circuit Design*, North Holland, 1992.

[HD92] F. K. Hanna and N. Daeche, *Dependent Types and Formal Synthesis*, pp121-135,Phil Trans. Royal Soc, Series A, (1992), Vol 339,

[JM92] B. Jacobs, T. Melham, *Translating Dependent Type Theory into Higher Order Logic*, Tech. Rep., University of Cambridge, Computer Laboratory, 1992.

[PT90] L. Paulson, T. Nipkow, *Isabelle tutorial and user's manual*, Tech. Rep., University of Cambridge, Computer Laboratory, 1990.

[VHDL87] *VHDL Language Reference Manual*, IEEE standard 1076-1987.