Structure and Behaviour in Hardware Verification

K. G. W. Goossens
Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, U.K.

Abstract. In this paper we review how hardware has been described in the formal hardware verification community. Recent developments in hardware description are evaluated against the background of the use of hardware description languages, and also in relation to programming languages. The notions of structure and behaviour are crucial to this discussion.

1 Introduction

Hardware has long been described using hardware description languages (HDLs). More recently, in the field of hardware verification logic-based notations have been used. In this paper we explore how the relationship between the structure and behaviour of circuits has been perceived over time in the formal verification field. The structure of this paper is as follows: we give our view of HDLs and simulation prior to the advent of formal methods, then we comment on formal logic methods used to describe and reason about hardware. Connections with conventional programming languages are also explored.

Hardware Description Languages and Simulation

The first rôle of HDLs was to document hardware designs and facilitate communication between designers [11, 36]. It was soon realised, however, that these descriptions could be used to *simulate* the realisations of the designs they described [34]. The shift from the use of HDLs as documentation to their use as behavioural descriptions is important. A structural description of the physical realisation of the system has been replaced by the behavioural description of the design of the system.
In the former situation there is an explicit understanding that every construct in the language stands for, or represents, a real hardware component. (In fact, in [54] an HDL was given a semantics in these terms. See also PMS [36].) In the presence of simulation, however, an HDL description requires a model that defines the behaviour of the basic components of the language. The gap between an HDL description and the behaviour of one of its implementations is filled by the simulation model, or model of hardware. Features that the model abstracts away from cannot be reasoned about, and if the model is unrealistic or incorrect, the behaviours associated with an HDL program are also invalid. While this has always been clearly understood in areas such as device modelling where simulation programs have been used extensively [53] and system-level modelling, this was not always so obvious in formal hardware verification [17].

A separate development addressed the need to document and design systems at higher levels of abstraction. Behavioural notations such as ISP [60], closer to conventional programming languages, were defined for this purpose. By definition, this type of description does not relate to any particular implementation. Simulation of higher-level descriptions is less contentious than that of structural descriptions because the former do not relate to an underlying physical implementation via a model like the latter. Note that a design written in a behavioural HDL can only be interpreted indirectly, using a simulator.

The two distinct developments of simulation and emphasis on behaviour, together with the ability to generate structural descriptions from low-level behavioural hardware descriptions using synthesis tools, diffused the original intention of hardware description languages: to document circuit implementations. Formal hardware verification started from these premises, and it is therefore not surprising that structure and behaviour were not cleanly separated until recently.[1] In the remainder of this section we review how hardware has been described in the formal hardware verification community until recently. Some research explicitly addressing these issues is then discussed.

Structure and Behaviour in Formal Hardware Verification

Where proof assistants have been used in the hardware verification community, the following schema has generally been employed:

$$\vdash \textit{implementation} \text{ IMPLEMENTS } \textit{specification}$$

The relation IMPLEMENTS expresses that the implementation satisfies the specification. IMPLEMENTS has been interpreted as equivalence ($\leftrightarrow$ or $=$), and implication ($\rightarrow$). Although more sophisticated notions have been investigated [7], logical implication is used predominantly. Nearly always *implementation* is a relation between input and output signals, describing the behaviour of the design under consideration. This behavioural description of the implementation is commonly regarded as a structural description. However, in purely structural descriptions there is no behavioural information: `and(x,y,z)` means only that in the corresponding place in the implementation there is 'a piece of hardware commonly called an AND gate.'

In the approach taken by researchers using the Boyer-Moore theorem prover the circuit description *b-and x y* already denotes a particular behaviour — that normally associated with an AND gate. Consider the following representative example from [12] below. The description has been broken down into small components that we associate immediately with their usual gate-level implementations but the description remains behavioural.

(defn b-not a) (if (equal a F) T F)
(defn b-and a b) (if (and (boolp a) (boolp b))

[1] Although, of course, a major reason for formal hardware verification was the early realisation that simulation alone would not be feasible for the verification of hardware [11]. Note that mathematical logic may be considered as a sufficiently expressive behavioural HDL not to require animation.

(and (equal a T) (equal b T)) F)
(defn b-nand a b) (b-not (b-and a b))

The example consists of a composition of constants that already have an interpretation. We insist on commencing with the uninterpreted syntax of a structural language; behaviour is a secondary concept, and is provided in an explicit manner [23, 24]. This highlights the fundamental difference between the structure of hardware and the behaviour of the hardware, when it is abstracted using a particular model. In the Boyer-Moore system only Brock and Hunt have used this approach [12]. Their work is discussed in Section 4. Other Boyer-Moore work provides interpretations such as the one given above; the hardware description is a recursive function which is intended to model the behaviour of the design. The use of tail recursion to represent the advance of time was introduced by Hunt [35], and has generally been used by hardware verification research based on the Boyer-Moore theorem prover.

In higher-order logic proof assistants such as LAMBDA [22] and HOL [29], nearly all work has been in terms of similar direct interpretations [32, *e.g.* Section 4]. Exceptions are discussed later. Consider the usual HOL definition of an AND gate: $\vdash \texttt{and}(x, y, z) = (z = x \wedge y)$. It defines a three-place relation between booleans. It may be composed with a similarly defined NOT gate as follows:

$$\vdash \texttt{and}(x, y, a) \wedge \texttt{not}(a, z)$$

Although this looks conspicuously like a structural description it is a behavioural description, composed of the two very simple relational descriptions `and` and `not`. Consider another implementation of a NAND gate:

$$\vdash \texttt{not}(x, a) \wedge \texttt{not}(y, b) \wedge \texttt{or}(a, b, z)$$

These two descriptions are logically equivalent, but are intended to denote structurally different circuits. The identical behaviour (at this level of abstraction) is captured, but the structural distinction is lost. For this reason we introduce a description that is truly structural:

$$\vdash \texttt{P}(\texttt{strand}(\texttt{x}, \texttt{y}, \texttt{a})\ \texttt{\&\&}\ \texttt{strnot}(\texttt{a}, \texttt{z}))$$

There is a considerable difference between the first relational behavioural description, and this purely structural description. `strand` is an object denoting a purely structural AND gate. `&&` is an operator combining structural descriptions, with result type *structural*. The purely structural description is not a truth valued expression, like the relational descriptions: we have to say something about the structural expression, which is what the context `P` indicates. For example, we could give a meaning to the structural description using a semantics, synthesise circuits, *etc.* See Section 4 for more details.

In our opinion a proper separation between the structural and behavioural aspects of a circuit description is crucial. In the remainder of this section we review research that has explicitly addressed this issue.

Research Addressing These Issues

In [33] Hanna and Daeche present the VERITAS hardware verification approach. Theories are used to define new notions such as a theory of *gate behaviours* containing basic gates. It is important to note that only behaviours are defined; there is no mention of structure. For example, if *wf* is the type of waveforms,

$$\vdash \text{ANDBEHAV} : \ \mathit{characteristics} \rightarrow (\mathit{wf} \times \mathit{wf} \times \mathit{wf}) \rightarrow \mathit{bool} \ = \ \mathit{definition}$$

is a parametrised relational definition of the behaviour of an AND gate. The association of structure with behaviour can only be completed after a theory of *simple structures* has been given. This theory defines the structural aspects of a circuit. Elements of a type correspond to implementations; subtypes are used to axiomatise input and output ports, components, and interconnections. Projection functions are used to extract characteristics from structural entities. For example, we use the function $\mathit{in}_i : \mathit{andgate} \rightarrow \mathit{inport}$ to obtain the ith input port of an AND gate. We associate an AND gate behaviour ANDBEHAV, as defined in the gate behaviour theory, with a particular simple structure g of type *andgate* as follows:

$$\vdash \forall g : \mathit{andgate}.\ \text{ANDBEHAV}\ (\mathit{characteristics}\ g)\ (\mathit{in}_1\ g)\ (\mathit{in}_2\ g)\ (\mathit{out}\ g)$$

This axiom states that every purely structural AND gate g with its particular properties, in this case *characteristics* g, input and output ports, satisfies the behaviour of an AND gate as axiomatised by ANDBEHAV. Finally, a theory of *compound structures* defines composite structures, properties of which, such as subgates and their interconnections, are again obtained by applying projection functions. The behaviour of composite structures may be derived from the behaviours of subcomponents. This work is a good example of the separation of structure and behaviour. It is distinctive in its use of projection functions to extract the composition of non-simple structures. Usually subcircuits are combined explicitly using composition and hiding operators (*e.g.* CIRCAL [45] and LCF_LSM [30]).

Wang [61] describes a Hardware Synthesis Logic which also maintains a clear distinction between structure and behaviour. Circuit structures are composed in a simple structural algebra, called the implementation language, containing a structural connective &, which is comparable to `&&` introduced previously. A logic called the specification language is used to reason about properties of implementations and about specifications. The calculus is independent of a particular specification logic, although a higher-order logic is used in the example below. The implementation and specification languages are related through a so-called construction logic, which contains some inference rules and axiom schemas. The latter define, using the specification language, the behaviour S of basic terms I in the implementation language. This is denoted by the use of the connective in $I \Subset S$. For example:

$$\mathit{Register}(i, c, o) \Subset \forall t.\ o(S\ t) = \mathit{if}\ c\ t\ \mathit{then}\ i\ t\ \mathit{else}\ o\ t$$

The structural conjunction & is preserved by $\models\!\models$, so that the following inference rule is part of the calculus:

$$\frac{\vdash I_1 \models\!\models S_1 \quad \vdash I_2 \models\!\models S_2}{\vdash I_1 \& I_2 \models\!\models S_1 \wedge S_2}$$

Wang proves a number of meta-results relating the implementation, specification, and construction logics.

2 Programming Language Semantics

The structure versus behaviour issues discussed above have been investigated for conventional programming languages using formal semantics. Three types of semantics have been proposed to give meaning to programs; axiomatic [20], denotational [51], and operational [49]. The three types of semantics may be viewed as progressively more concrete, and therefore suited to different applications [51]. Axiomatic semantics map programs directly onto properties characterising their behaviours. Denotational semantics map programs onto functions, from which input-output behaviours may be derived. Operational semantics allow a behaviour to be derived through the sequence of transitions a program may perform.

In Section 3 axiomatic and denotational approaches to hardware description are presented, whereas Section 4 contains operational methods. In both sections informal, partially formal, and formal methods are distinguished.

3 Extracting Behaviour From Circuit Descriptions

The intuitive solution to the structure-behaviour division is to *extract a behaviour* from a circuit description directly. We have a function `behaviour` : *structural* $\rightarrow$ *bool*. In other words, `behaviour` maps a hardware description to a logical formula characterising its behaviour. For example:

$$\texttt{behaviour}\ (\texttt{delay}(\texttt{c}, \texttt{in}, \texttt{out})) \ = \ (\mathit{out}\ 0 = \texttt{c} \ \wedge \ \forall t.\ \mathit{out}(\texttt{S}\ t) = \mathit{in}\ t) \qquad (1)$$

Here `delay(c,in,out)` is an HDL description for a unit transport delay. Let us first assume that this equation is entirely outside a proof system. This definition raises the following question: what is the relation between `in` and *in*? The former is a syntactic structural object, whereas the latter is part of the formal system in which the behaviour is expressed. The situation is clarified by giving explicit types to the various components:

$$\begin{aligned} \texttt{behaviour} &: \mathit{structural} \rightarrow \mathit{bool} \\ \texttt{delay} &: (\mathit{value} \times \mathit{name} \times \mathit{name}) \rightarrow \mathit{structural} \\ \texttt{in} &: \mathit{name} \\ \mathit{in} &: \mathit{signal} = \mathit{time} \rightarrow \mathit{value} \end{aligned}$$

We would like behaviour functions to always produce formulae that are consistent, *i.e.* do not contain contradictions. If this were not the case, a particular circuit for which an inconsistent behaviour description was produced would satisfy

any specification. We note that in principle the range of the behaviour function may be anything, as long as it allows us to express our intuitions about the behaviour of circuits. If the result is truth-valued then the behaviour function could be called axiomatic, or denotational otherwise.

The definition of `behaviour` could be an entirely informal exercise, but rather than using an *ad hoc* implementation of the manipulation of behaviours later work advocated mapping the extracted behaviour into a proof system. This lead to a clean separation of conceptually different processes, namely the extraction of the behaviour and the formal reasoning about this behaviour. We may view Equation 1 in this light; the right hand side could be inside the proof system. One fundamental problem remains: the behaviour function itself resides outside the proof system. This means that we cannot reason about it within the proof system. In particular, we will have to accept the correctness of the implementation of the behaviour function in good faith. The HDL description is also informal, which means we cannot reason about structural terms. We can only use the behaviour of the design, and no structural aspects, inside the proof system. This becomes a problem where we want to reason about general properties possessed by all, or a set of circuits. The solution is to move the behaviour function into the proof system also. For example, some hardware models may satisfy the property that for every input an output exists. It is preferable to prove a general theorem of the form

$$\vdash \forall e : \mathit{structural}.\ \forall i : \mathit{const}.\ \exists o : \mathit{const}.\ \mathit{simulation}\ e\ i = o$$

rather than a number of instantiations. It is important to note that the type *structural*, representing terms of type circuit, resides inside the proof system. Thus the structural circuit description may be manipulated independently from its behaviour; we discuss this in more detail in Section 4. Whether formal or informal, there is a real separation between the description of the circuit and its behaviour.

The remainder of this section refers to research that has some aspect of explicitly relating structural descriptions to behaviour.

Informal Behaviour Extraction Functions

Early research into hardware verification was informal and rather *ad hoc*. Most efforts took the form of a software system that given a hardware description and a specification would try to show their equivalence. From a historical perspective we may consider these efforts as primitive behaviour extraction functions. In the late 1970s and early 1980s a number of efforts were directed at *functional abstraction*; this is to the process of extracting a behaviour from a circuit description [38, 4, 39].

Pitchumani and Stabler [48] used a Floyd-Hoare style semantics to give a definition for a register transfer-level HDL. The language which is described in [48] does not have an explicit notion of time. Rather, time is introduced in the semantics through the use of a distinguished variable t that represents time. It may be used in pre- and post-conditions, but not in programs. This precludes assignments to the time variable, but does allow temporal information to be given

in the specification. Consider the NULL statement with its conventional semantics $\{P\}$ NULL $\{P\}$. When time is involved this becomes $\{P[t+1/t]\}$ NULL $\{P\}$. Thus NULL has no effect other than to pass time.

Partially Formal Behaviour Extraction Functions

In [8] Borrione and Paillet recognise the need for a formal system to unambiguously express the semantics of an HDL. They outline the design of a system to translate VHDL descriptions to a representation of their behaviour in a proof system. The behaviour is represented by a set of simultaneous functional equation, in the Boyer-Moore and REVE proof systems.

Boulton [10] describes a behaviour extraction function from a subset of ELLA[2] to the HOL proof assistant. The behaviour function and its abstract syntax tree input are outside the formal part of the HOL proof system. ELLA constructs are mapped to high-level behaviours in HOL. For example, consider the `case` statement in ELLA:

```
⟦case in of lo: hi, hi: lo⟧ =
CASE ⟦in⟧ [OF [⟦lo: hi⟧; ⟦hi: lo⟧]] (UNLIFT UU) =
CASE in [OF [CONST lo, SIGNAL LIFT_hi;
             CONST hi, SIGNAL LIFT_lo]] (UNLIFT UU)
```

The behaviour function $[\![\cdot]\!]$ gives a semantics to the structural description `case in of lo: hi, hi: lo`. `CASE` and `OF` are HOL functions that, given the subcomponents' behaviours ⟦`lo: hi`⟧ and ⟦`hi: lo`⟧, represent the behaviour of the whole `case` statement. Because this behaviour function is itself not part of HOL, the `case` statement is informal and the variable `in` has no explicit relation to *in* (*cf.* Equation 1).

Other related work includes [58, 19] which describe mapping VHDL into HOL and SDVS respectively. SILAGE has also been given a HOL semantics as above [27]. In [47] behaviours of CASCADE descriptions are mapped into the Boyer-Moore and TACHE theorem provers. Recently Umbreit has used LAMBDA to map VHDL programs onto formally defined ML descriptions [57].

Formal Behaviour Extraction Functions

In [41] Melham describes a formal behaviour function in HOL. He defined an abstract data type representation of CMOS circuit descriptions inside the HOL proof assistant. Part of this data type is given below.

$$circ ::= \texttt{pwr}\ str \mid \texttt{ntran}\ str\ str\ str \mid \texttt{join}\ circ\ circ \mid \ldots \tag{2}$$

`join` $c\ c'$ is structural composition, comparable to `&&` introduced earlier. Switch-level model and threshold model semantics were defined using primitive recursion functions. A fragment of the former is:

$$\begin{aligned} &\vdash \texttt{Sm}\ (\texttt{pwr}\ p)\ e &&= (e\ p = T) \\ &\vdash \texttt{Sm}\ (\texttt{ntran}\ g\ s\ d)\ e &&= (e\ g \supset\ (e\ d = e\ s)) \\ &\vdash \texttt{Sm}\ (\texttt{join}\ c_1\ c_2)\ e &&= \texttt{Sm}\ c_1\ e \wedge \texttt{Sm}\ c_2\ e \end{aligned} \tag{3}$$

[2] ELLA is a trademark of the Secretary of State for Defence, United Kingdom.

$\texttt{Sm} : \mathit{circ} \rightarrow (\mathit{str} \rightarrow \mathit{bool}) \rightarrow \mathit{bool}$ is the function mapping circuits with environments to a formula describing their switch-level behaviour. The term $e : \mathit{str} \rightarrow \mathit{bool}$ is the environment, mapping strings *str*, denoting wire names, to their values. As we briefly indicated in Section 3 because the data type expressions are ordinary proof system terms we may quantify over structural descriptions. This feature was used to relate the switch-level and threshold models of hardware formally, *i.e.* as a theorem in HOL.

In [5] Basin uses the NUPRL proof assistant [18], which implements a constructive type theory. He uses the *proofs-as-circuits* paradigm, which is an adaptation of the *propositions-as-types* idea. A constructive proof contains computable evidence, *e.g.* a circuit, of the truth of the proposition it proves. Different proofs correspond to different implementations. Proving

$$>> \ \forall i, o.\ \exists c.\ S(i, o, c)$$

entails exhibiting a witness c that satisfies the specification $S(i, o, c)$. ($>>$ is NUPRL's judgement.) There is no guarantee, however, that realisation c has a particular form, or *intention*; we only know that it has behaviour, or *extension*, S. We would like c to be a circuit description, not just any old proof term. To force the realisation to have a particular form, or to be at a particular level of abstraction, a type of circuit terms is introduced. This type *trans* is a recursively defined data type. An interpreter $\mathit{Interp}_{\mathit{trans}} : \mathit{trans} \rightarrow \mathit{env} \rightarrow \mathit{bool}$ is defined to give a meaning to these terms. *trans* and $\mathit{Interp}_{\mathit{trans}}$ correspond to Melham's *circ* (Equation 2) and Sm (Equation 3) respectively.

The VERITAS approach, discussed in the introduction, corresponds to an axiomatic approach fully within a proof system.

4 Deriving Behaviour via a Semantics

In the previous section we showed how behaviour could be extracted directly from circuit descriptions. This is a high-level approach with no indication of an underlying model of how the behaviour is arrived at. Industrial HDLs usually have a simulator to animate hardware descriptions. It makes sense not to state properties directly about circuit descriptions, but to derive properties using the simulator. That is, we take a more operational stance. Taken at face value, this would seem to imply that we can only derive properties using simulation; exactly what we are trying to get away from. This is not the case, however: if we provide an operational semantics for the HDL, we may prove *general properties* about the simulator model. For example, we can characterise the domain on which the simulator is a total function. An operational semantics gives us a firm mathematical grip on the simulator model.[3] Often we can prove more detailed properties using operational semantics than with other types of semantics because we can refer to the simulation method.

As with behaviour functions earlier, we can define an operational semantics on paper or use a proof system. In this case, however, there is no half-way stage:

[3] Particular implementations of this algorithm may still be incorrect. [25] shows how formal simulation overcomes this problem.

either everything is on paper or everything is in a proof system. The reason for this is clear when we consider a fragment of an operational semantics.

$$\texttt{opsem}\ env\ (\texttt{wire}\ n) = env\ n$$
$$\texttt{opsem}\ env\ (\texttt{parcomp}\ (c_1, c_2)) = (\texttt{opsem}\ env\ c_1, \texttt{opsem}\ env\ c_2)$$
$$\texttt{opsem}\ env\ (\texttt{mux}\ (c_1, c_2, c_3)) = \mathit{if}\ \texttt{opsem}\ env\ c_1\ \mathit{then}\ \texttt{opsem}\ env\ c_2\ \mathit{else}\ \texttt{opsem}\ env\ c_3$$

`wire` n returns the value on the wire n, `parcomp` is parallel composition, and `mux` a multiplexor. Although it is conceivable to map from outside into a proof system this does not really make sense because the same objects and types occur in both the domain and range of the semantics. This was not necessarily the case for the axiomatic behaviour function of Equation 1.

The discussion that follows applies equally to 'paper' and embedded operational semantics. The difference is of a more pragmatic nature; it is possible to use an operational semantics on paper but it quickly becomes tedious and error-prone.

To embed an operational semantics in a proof system circuits, input and output values, and the semantic rules must be encoded. Auxiliary objects such as environments and wire names are also needed. The structure and behaviour of hardware are kept separate by providing a structural description language, which is given a meaning through the use of a semantics (*cf.* Section 3). Operational semantics relate a circuit and its inputs to an output according to some simulation model. A type of the semantics could be the following (for simplicity we allow only one input):

$$\texttt{opsem} : (\mathit{structural} \times \mathit{value}) \times \mathit{value}$$

Here the concept of state is missing; most circuits contain latches, which retain a value between clock cycles. Adding an explicit state yields the following (*cf.* [59]):

$$\texttt{opsem} : (\mathit{structural} \times \mathit{state} \times \mathit{value}) \times (\mathit{value} \times \mathit{state})$$

An alternative view is to dispense with the state, and evolve the circuit itself so that the state is part of the circuit description [23]:

$$\texttt{opsem} : (\mathit{structural} \times \mathit{value}) \times (\mathit{value} \times \mathit{structural})$$

Introduced by Milner [43], this type was used by Gordon as the basis for LCF_LSM [30]. State transition functions of state machines have a similar type. This view is also common in process algebras such as CCS [44], CIRCAL [45], and HOP [26], which use labelled transition systems.

After the structural aspects of the HDL have been defined they can be manipulated using proof system facilities. This leads to a number of possible applications: we may quantify over circuits, expressing properties that hold for all or particular classes of circuits. Circuits expressions can contain free variables, corresponding to plug-in components [23]. Circuits may be operated on by transformation functions, which may be proven correct [24], or be the result of formal hardware synthesis functions [12]. Interactive synthesis, perhaps based on the operational semantics rules [24], or refinement-based strategies [21] is

also possible. Operational semantics based formal simulation is another powerful application [25]. It may be very useful to have multiple semantic functions emphasising different aspects of the structural description [12, 46].

Embeddings of hardware notations are more powerful but also harder to use than extracting the behaviour directly because one has to resort to the semantics to obtain any behavioural information (see *e.g.* [9, Section 7.8]).

Recently a number of HDLs have been given formal semantics. With a few exceptions, these have all been paper exercises [55, 2, 62], although [3, 56] provide computer support. Below we discuss work in conjunction with proof systems.

Compiler Correctness in Proof Systems

Correctness proofs of compiler (algorithms) in proof systems use the same techniques as those for embedding HDLs in proof systems. To reason about programs their syntax and semantics have to be encoded in the proof system.

Milner and Weyhrauch used the Stanford LCF proof checker to check the correctness of a simple compiler algorithm [42]. The source and target languages were axiomatised in the system through the use of constructors and destructors. Aiello *et al.* encoded a denotational semantics for Pascal in the Stanford LCF in a similar manner [1]. Using the Edinburgh LCF [28] Cohn proved a compiler correct with respect to the denotational semantics of imperative source and target languages [16]. Other research involving compiler correctness proofs using proof systems includes Sokolowski's LCF work [52]. Joyce verified a compiler using HOL with as target machine a non-idealised formally verified computer Tamarack [31] taking into account finite storage [37]. A group at Computational Logic [6] has used the Boyer-Moore theorem prover to verify a code generator, assembler and linker to a verified microprocessor FM8502.

Embedded Hardware Description Notations

Brock and Hunt [12] describe a simple combinatorial logic hardware description language in the Boyer-Moore theorem prover. This is the earliest research known to us that defines an operational semantics for an HDL in a proof system. Circuits are encoded as list constants, which are interpreted by a semantic function. For example, a full adder is described as follows.

```
'(half-adder (a b) (sum carry) (((sum) (b-xor a b))
                                ((carry) (b-and a b)))
'(full-adder (a b c ) (sum carry)
             (((sum1 carry1) (half-adder a b))
              ((sum carry2)  (half-adder sum1 c))
              ((carry)       (b-or carry1 carry2))))
```

The circuit *half-adder* is defined as having two inputs *a* and *b*, and two outputs *sum* and *carry*. *b-xor* and *b-and* represent primitive XOR and AND gates respectively. A well-formedness predicate is defined to check that these definitions are purely combinatorial. The output value of the circuit description is computed by an operational semantics, which is encoded as a total recursive function. The

conceptual type of the semantic function is as follows:

$$heval: \ name \rightarrow signalenv \rightarrow circuitenv \rightarrow value\ list$$

name consists of the name of the top-level component and its inputs. The environment *circuitenv* contains the definitions of non-primitive functions, such as *half-adder*, and *signalenv* is used to store the values of input, output, and internal variables such as *sum1*. To evaluate the half adder with inputs x and y with values F and T respectively, we use:

(heval '(half-adder x y) (list (cons 'x F) (cons 'y T)) (list '(half-adder (a b)...

A recent extension to this work allows sequential circuits with delayed feedback loops and explicit state holding components [13].

Goossens [23] describes the embedding of a formal static and dynamic operational semantics for a subset of the industrial HDL ELLA [50]. The HDL contains unit delays, generalised multiplexors, and allows both delayed and delayless feedback loops. In common with other work he defines a data type to define the abstract syntax of the HDL. Due to restrictions of LAMBDA version 3.2 the operational semantics is given as a function that is defined structurally on abstract syntax terms. This limits proofs to structural induction on program terms. A number of meta-level results such as the totality and monotonicity of the simulator model are proved [24].

The same approach is used by van Tassel to embed a VHDL subset in HOL [59]. Again an abstract data type is used to represent program terms, but here a HOL package to define inductive relations is then used to derive a rule induction principle from a relational semantics. This is a more general induction than Goossens' structural induction on the abstract syntax of programs [14] and the fixed-point (or computational) induction [40] in LCF. LAMBDA version 3.2 permits only functional semantics, whereas the HOL system allows more general relational semantics.

More recently a spate of embeddings in HOL has been reported [15].

5 Conclusions

In this article we attempted to illustrate the evolution of the separation of structural and behavioural aspects in formal hardware verification. Although behavioural hardware descriptions have shortcomings in this respect, their ease of manipulation compared to operational semantics based approaches is an advantage [24]. Due to the use of the underlying logic relational hardware description (*e.g.* [32]) is especially efficient.

References

1. L Aiello, M Aiello, and R Weyhrauch. The semantics of Pascal in LCF. Memo STAN-CS-74-447, Stanford Artificial Intelligence Laboratory, Computer Science Department, Stanford University, August 1974.
2. H Barringer, G Gough, and B Monahan. Operational semantics for hardware design languages. In P Prinetto and P Camurati, eds., *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 313–334. North Holland, June 1991.

3. H Barringer, G Gough, B Monahan, and A Williams. A semantics for Core ELLA. Deliverable D2.3b, Department of Computer Science, University of Manchester, November 1992. Formal Verification Support for ELLA, IED project 4/1/1357.
4. H Barrow. Verify: A program for proving correctness of digital hardware designs. *Artificial Intelligence*, 24:437–491, 1984.
5. D Basin. Extracting circuits from constructive proofs. In *1991 Int'l Workshop on Formal Verification in VLSI Design*. January 1991.
6. W Bevier, W Hunt, Jr, J Moore, and W Young. An approach to systems verification. *J. of Automated Reasoning*, 5:411–428, 1989.
7. J Bormann, H Nusser-Wehlan, and G Venzl. Formal design in an industrial research laboratory: Lessons and perspectives. In J Staunstrup and R Sharp, eds., *Workshop on Designing Correct Circuits*, pages 193–213, Denmark, January 1992.
8. D Borrione and J L Paillet. An approach to the formal verification of VHDL descriptions. Technical Report RR 683-I-, IMAG/ARTEMIS, November 1987.
9. R Boulton, A Gordon, M Gordon, J Harrison, J Herbert, and J van Tassel. Experience with embedding hardware description languages in HOL. In V Stavridou, T Melham, and R Boute, eds., *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156. North Holland, June 1992.
10. R Boulton, M Gordon, J Herbert, and J van Tassel. The HOL verification of ELLA designs. Technical Report 199, University of Cambridge Computer Laboratory, August 1990.
11. M Breuer. General survey of design automation of digital computers. *Proceedings of the IEEE*, 54(12):1708–1721, December 1966.
12. B Brock and W Hunt, Jr. The formalization of a simple hardware description language. In L Claesen, ed., *Applied Formal Methods For Correct VLSI Design*, pages 778–792, Amsterdam, November 1989. Elsevier Science Publishers.
13. B Brock, W Hunt, Jr, and W Young. Introduction to a formally defined hardware description language. In V Stavridou, T Melham, and R Boute, eds., *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 3–35. North Holland, June 1992.
14. R Burstall. Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–44, 1969.
15. L Claesen and M Gordon, eds. *Higher Order Logic Theorem Proving and its Applications*, Leuven, Belgium, September 1992. North Holland.
16. A Cohn. High level proof in LCF. Internal Report CSR-35-78, Department of Computer Science, University of Edinburgh, November 1978.
17. A Cohn. The notion of proof in hardware verification. *J. of Automated Reasoning*, 5(2):127–139, June 1989.
18. R Constable and D Howe. Nuprl as a general logic. In P Odifreddi, ed., *Logic and computer science*, volume 31 of *APIC studies in data processing*, pages 77–90. Academic Press, 1990.
19. I Fillippenko. VHDL verification in the state delta verification system (SDVS). In *1991 Int'l Workshop on Formal Verification in VLSI Design*, January 1991.
20. R Floyd. Assigning meanings to programs. *Proceedings of American Mathematical Society, Symposia in Applied Mathematics*, 19:19–32, 1967.
21. M Fourman and E Mayger. Formally based system design – interactive hardware scheduling. In G Musgrave and U Lauther, eds., *Int'l Conf. on VLSI*, 1989.
22. M Francis, S Finn, and E Mayger. *Reference Manual for the Lambda System*. Abstract Hardware Limited, version 3.2, November 1990.

23. K G W Goossens. Embedding a CHDDL in a proof system. In P Prinetto and P Camurati, eds., *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 359–374. North Holland, June 1991.
24. K G W Goossens. *Embedding Hardware Description Languages in Proof Systems*. PhD thesis, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, December 1992.
25. K G W Goossens. Operational semantics based formal symbolic simulation. In L Claesen and M Gordon, eds., *Higher Order Logic Theorem Proving and Its Applications*, pages 487–506, Leuven, Belgium, September 1992. North Holland.
26. G Gopakakrishnan, R Fujimoto, V Akella, N Mani, and K Smith. Specification-driven design of custom hardware in HOP. In G Birtwistle and P Subrahmanyam, eds., *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 128–170, New York, 1988. Springer Verlag.
27. A Gordon. The formal definition of a synchronous hardware description language in higher order logic. In *Int'l Conf. on Computer Design*, October 1992.
28. M Gordon, R Milner, and C Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.
29. M Gordon. HOL: A proof generating system for higher-order logic. In G Birtwistle and P Subrahmanyam, eds., *VLSI Specification, Verification and Synthesis*, pages 73–128, Boston, 1987. Kluwer Academic Publishers.
30. M Gordon. LCF_LSM. Technical Report 41, University of Cambridge Computer Laboratory, September 1983. Second Printing with Corrections and Additions.
31. M Gordon. Proving a computer correct. Technical Report 42, University of Cambridge Computer Laboratory, 1983.
32. M Gordon. Why higher-order logic is a good formalisation for specifying and verifying hardware. In G Milne and P Subrahmanyam, eds., *Formal Aspects of VLSI Design*, pages 153–177, Amsterdam, 1985. North Holland.
33. F K Hanna and N Daeche. Specification and verification using higher-order logic. In C Koomen and T Moto-Oka, eds., *CHDL 85: 7th Int'l Symposium on Computer Hardware and Description Languages and their Applications*, pages 418–433, Amsterdam, 1985. North Holland.
34. G Hays. Computer-aided design: Simulation of digital design logic. *IEEE Trans. on Computers*, C-18(1):1–10, January 1969.
35. W Hunt, Jr. FM8501: A verified microprocessor. Technical Report 47, Institute for Computing Science. The University of Texas at Austin, December 1985.
36. IEEE computer, December 1974. Special edition on Hardware Description Languages.
37. J Joyce. A verified compiler for a verified microprocessor. Technical Report 167, University of Cambridge Computer Laboratory, March 1989.
38. S Leinwand and T Lamdan. Design verification based on functional abstraction. In *16th Design Automation Conf.*, pages 353–359, San Diego, California, June 1979.
39. J-C Madre and J-P Billon. Proving circuit correctness using formal comparison between expected and extracted behaviour. In *Proceedings of the 25th ACM/IEEE Design Automation Conf.*, pages 205–210, 1988.
40. Z Manna, S Ness, and J Vuillemin. Inductive methods for proving properties of programs. *ACM SIGPLAN Notices*, 7:27–50, 1972.
41. T Melham. Using recursive types to reason about hardware in higher order logic. Technical Report 135, University of Cambridge Computer Laboratory, May 1988.
42. R Milner and R Weyhrauch. Proving compiler correctness in a mechanised logic. In B Meltzer and D Mitchie, eds., *Machine Intelligence*, chapter 3. Edinburgh University Press, 1972.

43. R Milner. Processes: A mathematical model of computing agents. In Rose and Shepherdson, eds., *Logic Colloquium 73: Studies in Logic and Foundations of Mathematics*, volume 80, pages 157–173. North Holland, 1973.
44. R Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.
45. F Moller. The definition of CIRCAL. In L Claesen, ed., *Applied Formal Methods For Correct VLSI Design*, pages 178–187, Amsterdam, November 1989. Elsevier Science Publishers.
46. J O'Donnell. Hardware description with recursion equations. In M Barbacci and C Koomen, eds., *CHDL 87: 8th Int'l Symposium on Computer Hardware Description Languages and Their Applications*, pages 363–382. North Holland, 1987.
47. L Pierre. From a HDL description to formal proof systems: Principles and mechanization. In D Borrione and R Waxman, eds., *CHDL 91: 10th Int'l Symposium on Computer Hardware Description Languages and Their Applications*, April 1991.
48. V Pitchumani and E Stabler. A formal method for computer design verification. In *19th Design Automation Conf.*, pages 809–814, 1982.
49. G Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University (DAIMI), 1981.
50. Praxis Systems plc, 20 Manvers Street, Bath BA1 1PX. *The ELLA Language Reference Manual*, issue 3.0, 1986. ELLA is now marketed by R^3 Systems.
51. D Schmidt. *Denotational Semantics, A Methodology for Language Development*. Allyn and Bacon Inc, Boston, 1986.
52. S Sokolowski. Soundness of Hoare's logic: An automated proof using LCF. *ACM Trans. on Programming Languages and Systems*, 9(1):100–120, January 1987.
53. R Spence. Progress in computer-aided circuit design. *CAD*, 1(4):19–24, 1969.
54. E Stabler. System description languages. *IEEE Trans. on Computers*, C-19(12):1160–1173, December 1970.
55. V Stavridou, J Goguen, S Elker, and S Aloneftis. FUNNEL: A CHDL with formal semantics. In P Prinetto and P Camurati, eds., *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 115–137. North Holland, June 1991.
56. V Stavridou, J Goguen, A Stevens, S Eker, S Alonefits, and K M Hobley. FUNNEL and 2OBJ: Towards and integrated hardware design environment. In V Stavridou, T Melham, and R Boute, eds., *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 197–223. North Holland, June 1992.
57. G Umbreit. Providing a VHDL-interface for proof systems. In *EURO-DAC*, pages 698–703, September 1992. IEEE Computer Society Press.
58. J van Tassel and D Hemmendinger. Toward formal verification of VHDL specifications. In L Claesen, ed., *Applied Formal Methods For Correct VLSI Design*, pages 261–270, Amsterdam, November 1989. Elsevier Science Publishers.
59. J van Tassel. A formalisation of the VHDL simulation cycle. In L Claesen and M Gordon, eds., *Higher Order Logic Theorem Proving and Its Applications*, pages 359–374, Leuven, Belgium, September 1992. North Holland.
60. W vanCleemput. Computer hardware description languages and their applications. In *16th Design Automation Conf.*, pages 554–560, San Diego, California, June 1979.
61. L-G Wang. Hardware synthesis logic and its independence. Manuscript. Laboratory for Foundations of Computer Science, Computer Science Department, University of Edinburgh, November 1991.
62. P Wilsey. Developing a formal semantic definition of VHDL. In J Mermet, ed., *VHDL for Simulation, Synthesis and Formal Proofs of Hardware*, pages 243–256. Kluwer Academic Publishers, 1992.