The Formalisation of a Hardware Description Language in a Proof System: Motivation and Applications[1]

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.
Email: kgg@dcs.ed.ac.uk

Abstract

Hardware description languages (HDLs) are a notation to describe behavioural and structural aspects of circuit designs. We discuss why it is worthwhile to give a formal semantics for an HDL, and why we have encoded such a semantics in a proof system. We outline the subset of the hardware description language ELLA[2] which we use, its formal structural operational semantics, and its embedding in the higher-order logic proof system LAMBDA[3]. Finally we discuss applications of this approach which include the ability to prove results about the simulation mechanism, formal symbolic simulation, various synthesis techniques, and transformational design.

Keyword Codes: B.7.2; F.3; I.2.3
Keywords: Integrated Circuits, Design Aids; Logics and Meaning of Programs; Deduction and Theorem Proving

1 Structure and Behaviour in Hardware Verification

Hardware description languages (HDLs) are a notation to describe designs of hardware. There is a wide spectrum of HDLs [23]; examples are DDL [8], ELLA [27], and VHDL [24]. Hardware description languages are used extensively in industry. However, academic research into the formal verification of hardware [32] predominantly uses notations based on higher-order logic [17]. This diminishes the chances of industrial adoption of verification methodologies based on academic research. By providing a formal basis for a widely used HDL, and building a methodology which is compatible with it, we hope this barrier will be reduced. In the following we give our perception of concepts such as structure and behaviour which play an important rôle in hardware design and description. We argue why the use of a formal semantics for an HDL is beneficial.

[1] To appear in the Proceedings of the XIII Conference of the Brazilian Computer Society, Florianopolis, Brazil, September 1993.

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

[3] LAMBDA is a product of Abstract Hardware Limited.

The rôle of HDLs has changed over time. The function of early HDLs was to document hardware implementations; only structural information was needed to describe components and their interconnections. The realisation that these HDL descriptions could also be used to *simulate* the hardware component which they documented was a conceptually very important step.[4] Simulation entails providing a circuit description with input stimuli, and computing its outputs according to some model. A *behaviour* was now associated with the structural description. The use of HDLs as documentation was vastly expanded to include design and debugging. To aid this new rôle behavioural language constructs were added to allow circuits to be described in terms of their behaviour only; no implementations is implied. (Note that conventional programming languages may be considered (and used) as a behavioural subset of an HDL because no hardware implementation is specified by a conventional program – it exhibits an input-output behaviour only.) HDLs could now be used throughout the design process. During the early stages in the design process designs are given in an abstract, behavioural manner, with no implementational bias. In later stages a design would consist of structural descriptions, reflecting the actual hardware implementation. In both cases, simulating the HDL description, should give the same behaviour.

A large body of research exists in the area of formal semantics for conventional programming languages [9, 22, 29, 26]. Formal semantics are essential to be able give a clear, unambiguous definition of a programming language. Properties about individual programs, as well as properties of classes of programs may be proven using formal semantics. As we saw previously, a conventional programming language may be regarded (somewhat unfairly) as the behavioural subset of a hardware description language. However, due to the prevalence in academia of notations other than industrial HDLs, formal semantics for HDLs were not considered until recently [3, 12, 28, 31, 16]. We now explain how hardware has been described in most of the hardware verification literature, and why this has obfuscated the boundary between the structure and behaviour of a circuit. The use of an HDL with associated formal semantics, in contrast, clarifies these issues [13].

We show how a simple `NAND` gate may be described using an HDL and using higher-order logic to illustrate problems with the latter approach. Using the hardware description language ELLA [27] two `NOT` gates, an `AND` gate, and an `OR` gate may be described (with minor variations) as follows:

```
FN NOT (BOOL: x) -> BOOL: CASE x OF true:  false ELSE true ESAC.
FN SLOWNOT (BOOL: x) -> BOOL: DELAY (NOT x).

FN AND (BOOL: x y) -> BOOL:
       CASE (x,y) OF (true,true):  true ELSE false ESAC.

FN OR (BOOL: x y) -> BOOL:
      CASE (x,y) OF (false,false):  false ELSE true ESAC.
```

[4] The behaviour of an HDL description and the corresponding implementation in hardware are related through a *model of hardware*. Features which the model abstracts away from (for example by regarding data signals as boolean values) cannot be reasoned about. If the model is unrealistic or incorrect, the behaviours associated with an HDL program are also invalid [6]. Prior to simulation of HDL descriptions no model was required.

All these gates are described in terms of their behaviour; they are primitive components. In contrast consider two different implementations of a NAND gate:

```
FN NAND1 (BOOL: x y) -> BOOL:
BEGIN
       MAKE NOT: not.
       MAKE AND: and.
       JOIN (x,y) -> and.
       JOIN and -> not.
OUTPUT not
END.

FN NAND2 (BOOL: x y) -> BOOL:
BEGIN
       MAKE OR: or.
       JOIN (NOT x,NOT y) -> or.
OUTPUT or
END.
```

Whereas the NOT, OR, and AND gates are described in terms of their behaviour, the NAND gates are defined structurally, by combining smaller components. Their behaviour can be derived from the behaviour of their subcomponents. In particular, NAND1 uses a NOT gate and not the slower, but functionally equivalent, implementation SLOWNOT. This is structural information. Moreover, NAND1 and NAND2 are exhibit the same behaviour but have a different internal structure.
Using higher-order logic, an AND gate may be described as follows [17]:

$$\vdash \mathtt{and}(x, y, z) = (z = x \wedge y)$$

It defines a three-place relation between booleans such that the output z is the boolean conjunction of the two inputs x and y. It may be composed with a similarly defined NOT gate as follows:

$$\vdash \exists z.\ \mathtt{and}(x, y, z) \wedge \mathtt{not}(z, a)$$

Although this looks conspicuously like the structural description NAND1, it is in fact a behavioural description, composed of the two very simple relational descriptions and and not. To see why this description is not sufficiently discerning, consider the equivalent to the NAND2 description:

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

However, we can prove trivially that

$$\vdash \exists z.\ \mathtt{and}(x, y, z) \wedge \mathtt{not}(z, a) \leftrightarrow \exists a, b.\ \mathtt{not}(x, a) \wedge \mathtt{not}(y, b) \wedge \mathtt{or}(a, b, z)$$

Thus two so-called structural descriptions in higher-order logic may be proven identical, although they are intended to denote structurally different circuits. This clearly shows that using logic in this manner to give structural hardware descriptions is inadequate.

The reason why a hardware description language does not suffer from the same problem is that we *derive* the behaviour of an HDL fragment using simulation (in the informal case) or using its *semantics* (in a formal setting). Thus `NAND1` and `NAND2` describe two structurally different circuits with the same (derived) behaviour.

Following the example of conventional programming languages we give a formal semantics for a subset of the hardware description language ELLA. An outline of the "paper semantics" of picoELLA (as our ELLA subset is called) is given in the next section. The reasons for, and principles of embedding picoELLA in LAMBDA [11], the higher-order logic proof assistant we use, are given in Section 3. Our work combines *post hoc* verification, simulation, synthesis, and transformations in one framework, as described in Section 4.

2 picoELLA and Its Formal Semantics

The industrial hardware description language ELLA [27] is relatively small and has a clean simulator model. Although it is possible to give direct semantics for most of the language [2], we chose to use a minimal subset, called picoELLA, into which all of ELLA may be translated. We will only give a flavour of this subset; for aspects such as undefined, or 'don't know' values (*?type* below) consult [13, Chapter 3]. picoELLA is a very simple functional language with the following abstract syntax:

$$
\begin{array}{rcl}
pgm & ::= & \texttt{TYPE}\ decl\ \texttt{IN}\ pgm \mid expr \\
decl & ::= & type = cons_1 \mid \ldots \mid cons_n \mid type = type_1 * type_2 \\
expr & ::= & const \mid name \mid expr[1] \mid expr[2] \mid (expr_1, expr_2) \mid \\
 & & \texttt{LET}\ name = expr_1\ \texttt{IN}\ expr_2 \mid \texttt{LET INIT}\ const\ \texttt{REC}\ name = expr_1\ \texttt{IN}\ expr_2 \mid \\
 & & \texttt{IF}\ expr_1\ \texttt{MATCHES}\ pat\ \texttt{THEN}\ expr_2\ \texttt{ELSE}\ expr_3 \mid \texttt{DELAY}\ (const, expr) \\
pat & ::= & type \mid cons \mid (pat_1, pat_2) \mid pat_1 \mid pat_2 \\
const & ::= & ?type \mid cons \mid (const_1, const_2)
\end{array}
$$

Type definitions declare a number of constructors, or a non-recursive binary tuple type. Patterns *pat* are used in the `IF` statement which is a generalised multiplexor. If the output of circuit $expr_1$ matches pattern *pat* the `IF` delivers the result of $expr_2$, else that of $expr_3$.[5] The `LET` construct defines a local *name* for the result of circuit $expr_1$. Use of *name* in $expr_2$ corresponds to a fan-out of the signal, whereas substituting $expr_1$ for all occurrences of *name* in $expr_2$ would duplicate the hardware component $expr_1$. The recursive `LET` allows *name* to be used in $expr_1$ too. This corresponds to *feedback*, an example of which is given below. The term `INIT` *const* ensures unique typing and guarantees that all simulations terminate. A discrete and linear time base, isomorphic to the natural numbers, is introduced into the language by the `DELAY` construct. `DELAY(`c_t`,`e`)` is a unit delay of the circuit e. The result of evaluating e at time t will be output by `DELAY(`c_t`,`e`)` at the *next* time step $t+1$. At the *current* time t, the value c_t is the result. This shows that the state of the delay (*i.e.* the constant c_t) is explicit in the description of the delay. picoELLA dispenses with a memory or store which other languages

[5] Actually, if the output of $expr_1$ is insufficiently defined to decide between the two branches an undefined value is output. This is crucial for the semantic model [13, Chapter 3].

require to save the state from one time step to the next. Only an enviroment for use within one time step is needed. However, as a result of the explicit representation of the state, a new circuit description must be evaluated at each time step. The result of an evaluation consists therefore of a value output together with a description of the circuit at the next time step. The type of the simulation (or dynamic semantics) is therefore $(\mathit{environment} \times \mathit{expression}) \rightarrow (\mathit{value} \times \mathit{expression})$. State transition functions of state machines have a similar type $(\mathit{value} \times \mathit{state}) \rightarrow (\mathit{value} \times \mathit{state})$, as do process algebras such as CCS [25].

As an example, consider the following circuit which implements a parity checker. It returns `hi` at time $t+1$ if there have been an even number of `hi`'s on the input signal `input` during the closed time interval $[0, t]$. At time zero it outputs `hi`.

```
LET INIT ?bit
    REC xor = DELAY (hi, IF (input,xor) MATCHES (hi,lo)|(lo,hi)
                         THEN hi ELSE lo)
IN xor
```

Few HDLs and programming languages have a precise definition. In practice the simulator or compiler serves as this definition, but this leads to problems when different implementations present conflicting outputs. A *formal semantics* may be used to give a mathematical description of the behaviour of an HDL, *i.e.* how a simulator should behave.[6] General properties, such as termination of any simulation within a finite number of steps, may be proved about the semantics (and hence the behaviour of conforming simulators). Some HDLs that have been formalised include FUNNEL [28], SILAGE [16], and various subsets of ELLA [12, 1, 2, 21], and VHDL [31, 30]. The remainder of this section describes the semantics for picoELLA [13] which takes the form of a structural operational semantics [26]. It comprises a *static semantics* describing which programs are well-typed, and a *dynamic semantics* defining the run-time behaviour of well-typed programs. We will not discuss the static semantics here; it suffices to say that it is relatively straightforward. picoELLA dynamic semantics rules (17 in total) fall into two categories; those dealing with time, and those dealing with the evaluation of expressions within one time step. The reduceSeqCons rule belongs to the first class:

$$\frac{\Gamma' \vdash \ \mathit{expr}_t \Rightarrow \ o_t, \ \mathit{expr}_{t+1} \qquad \mathit{tl}, \ \Gamma \vdash \ \mathit{expr}_{t+1} \Rightarrow \ \mathit{tl}', \ \mathit{expr}_{t+N}}{i_t :: \mathit{tl}, \ \Gamma \vdash \ \mathit{expr}_t \Rightarrow \ o_t :: \mathit{tl}', \ \mathit{expr}_{t+N}}$$

Here expr_t is the program at time t, Γ the environment in which the program runs, and $i_t :: \mathit{tl}$ the input stream. Γ' is Γ with input value i_t adjoined (this will be made more precise later). This rule shows that at time t we run the program expr_t with input value i_t in environment Γ'. The output value o_t is added to the output stream, and the new program expr_{t+1} is evaluated with the remainder of the input stream. As explained previously, since the state of the circuit is explicit in its description, we need to evaluate

[6] Note that the implementation of the simulator may use any model, as long the input–output relation obeys the definition. The semantic definition should be clear and simple, not necessarily efficient. In fact, in [14] we show how the semantics may be used as a simulator.

a new circuit at every time step. A typical member of the second category of rules is the reduceTuple rule:

$$\frac{\Gamma \vdash\ expr_1 \Rightarrow\ v_1,\ expr_1' \qquad \Gamma \vdash\ expr_2 \Rightarrow\ v_2,\ expr_2'}{\Gamma \vdash\ (expr_1, expr_2) \Rightarrow\ (v_1, v_2),\ (expr_1', expr_2')}$$

To evaluate a tuple in environment Γ, both subexpressions must be evaluated in the same environment. The rule for the delay shows the use of the embedded state:

$$\frac{\Gamma \vdash\ expr \Rightarrow\ v,\ expr'}{\Gamma \vdash\ \texttt{DELAY}\ (c,\ expr) \Rightarrow\ c,\ \texttt{DELAY}\ (v,\ expr')}$$

The output from the delay is its latched value c. The new description of the delay, to be evaluated at the next time step, contains the output v from $expr$ at the current time step. In other words, it has latched this clock-cycle's output.

Some properties of the semantics are discussed at the end of the next section.

3 Embedding picoELLA in Lambda

The semantics described above, has been encoded in the LAMBDA proof system [11], which implements a polymorphic constructive higher-order logic of partial terms.[7] A large subset of the functional language ML [19] is used to define new data types and operations on data types *within* the logic. The soundness of the system cannot be compromised through new definitions. LAMBDA returns a number of rules axiomatising the new ML data types, such as existence of constructors, (in)equality rules, and a structural induction principle. Rules for functions include rewrite rules, which may be used to define derived rules and tactics. Tacticals can be used to combine tactics into rewrite strategies, symbolic simulation commands, *etc.*

The key to the embedding is the explicit encoding of HDL terms as objects in the proof system. It allows manipulation of circuit descriptions by functions and relations, and quantification over circuits, *e.g.* 'for all circuits c which are well-typed $P(c)$ holds', 'there exists a circuit c which implements specification S'. Indeed, the purely structural terms are given a meaning operationally, by the function `reduce` overleaf, which encodes the dynamic semantics. (An alternative approach is to give structural terms a meaning directly, through a denotational or axiomatic semantics. However, an operational definition allows reasoning about the simulator mechanism, and not just the input-output behaviour. For a detailed discussion see [15].) In the following, we first define a representation of constants and circuit expressions, then the semantic function which interprets them. Some important properties of the semantics are discussed towards the end of this section.

To represent picoELLA constants the ML definition system was used to define the data type *const*:

```
datatype const = Cons of natural * natural |
                 CoTuple of const * const;
```

[7] Note that we use LAMBDA version 3.2. More recent versions uses a different logic.

`Cons(`i`,`t`)` encodes the i^{th} constructor of type t. `Cons(0,`t`)` represents $?t$ which is the undefined, or 'don't know' value of type t. A constant is therefore a constructor or bottom value, or a tuple containing constants. The *pat* data type defined similarly to encode patterns of the `IF` statement. The type representing expressions or circuits is defined as follows.

```
datatype expr = Const of const |
                Tuple of expr * expr |
                Let of expr * expr |
                Var of natural |
                Delay of const * expr |
                If of expr * expr * expr * pat |
                Index1 of expr |
                Index2 of expr |
                LetRec of const * expr * expr;
```

Note that no constructor is present for `TYPE`. Types are dealt with on a meta-level, *i.e.* using LAMBDA's facilities, rather than at the *expr* object level. To embed the `LET` operator the de Bruijn encoding of lambda abstractions is used [7]. The bound variables of lambda expressions are encoded as natural numbers indicating the distance (measured in intervening lambdas) away from the defining lambda. Thus $\lambda x.\lambda y.(x,(x,y))\ a\ b$ would be encoded as $\lambda\lambda(1,(1,0))\ a\ b$. In picoELLA this corresponds to embedding
`LET x = a IN LET y = b IN (x,(x,y))` as
`Let (`*a*`, Let (`*b*`, Tuple (Var 1, Tuple (Var 1, Var 0)))).`

As explained two pages back, the type of the dynamic semantics, defined by the `reduce`, is `reduce:` *const list* `->` *expr* `->` (*const* `*` *expr*), where *const list* represents the value environment.

```
fun reduce l (Let (e1, e2)) =
           let val (c1, f1) = reduce l e1 in
           let val (c2, f2) = reduce (c1::l) e2 in
                        (c2, Let (f1,f2))
           end end |
    reduce l (Var n) = (elem l n, Var n) |
    reduce l (Delay (c, e)) = (c, Delay (reduce l e)) | ...
```

The evaluation of the `LET` statement involves reduces the defining expression, and pushing the value result on the stack l (*cf.* storing it in the environment Γ). Evaluating a name is implemented by a lookup in the environment Γ in the dynamic semantics, and accessing the correct element in the stack l in the embedding.

Using these definitions and derived properties, the operational semantics rules described at the start of this section may be deduced within the proof system. For example, the (pretty-printed) rule `reduceSeqCons` below corresponds to reduceSeqCons shown previously.[8]

[8] All proof system output will be pretty-printed: for example, prefix constructors such as `Let(`e`,`f`)` will be shown infix `LET` e `IN` f. Rules whose name is shown in the `typewriter` font denote the embedded rules, those in roman font the 'paper' rules.

```
⊢ (instream_, env_ ⊢ circ1_ ⇒ (outstream_,circ2_))
⊢ (i1_ :: env_ ⊢ circ_ ⇒ (o1_,circ1_) : t_)
--------------------
⊢ (i1_ :: instream_, env_ ⊢ circ_ ⇒ (o1_ :: outstream_,circ2_))
```

To evaluate a program *circ_* with a non-empty input stream *i1_*::*instream_*, the head of the input stream is pushed onto the environment *env_*. This corresponds Γ' in the paper rules. *circ_* is then processed within this time step. The remainder of the input stream is then evaluated using the new circuit *circ1_*. Finally, the output *o1_* is prepended to the resulting output stream. `reduceDelay` shows how the static and dynamic semantics are used in the same rule:

```
⊢ initial_: t_
⊢ (env_ ⊢ circ_ ⇒ (out_,circ'_): t_)
--------------------
⊢ (env_ ⊢ DELAY (initial_,circ_) ⇒ (initial_,DELAY (out_,circ'_)): t_)
```

The first premise expresses the typing constraint that the constant *initial_* must have the same type *t_* as the expression *circ_*. The second subgoal is a shorthand for both a dynamic semantics inference (_ ⊢ _ ⇒ (_,_)) and a static semantics inference (_ ⊢ _ : _).

It is important to realise that *circ_*, *t_*, *etc.* are *meta-variables*. The `reduceDelay` rule is really a rule schema, which may be instantiated in an infinite number of different ways. When it is applied to a particular `DELAY` statement such as `DELAY (hi,Var 3)`, *initial_* and *circ_*, are unified with `hi` and `Var 3` respectively. This unification is reflected in every place where the variables occur in the rule. The unification works both ways so that meta-variables in a rule are specialised so that the rule applies to the current goal (as in the example below). But meta-variables in the goal may also be made more concrete for the rule to apply. We will see examples of this later on. In LAMBDA meta-variables may be *flexible* or *rigid*. The former are used to stand for some term to be determined as the proof proceeds, the latter require proofs to be schematic in the variable. Rigid variables ensure that a general result, rather than an instantiation of the result, is proved.

Properties of the Semantics

To increase the confidence in the correctness of the encoding of the semantics in the proof system we proved a number of properties about the embedded semantics which we also proved for the 'paper' version. These properties are of interest because they hold for all circuits which are well-typed (according to the static semantics). We proved that the reduction function is total, *i.e.* for all inputs an output will be produced. This is a non-trivial observation because feedback loops need not be broken by delays, which allows circuits which are meta-stable. Consider a `NOT` gate with its output fed into its input:

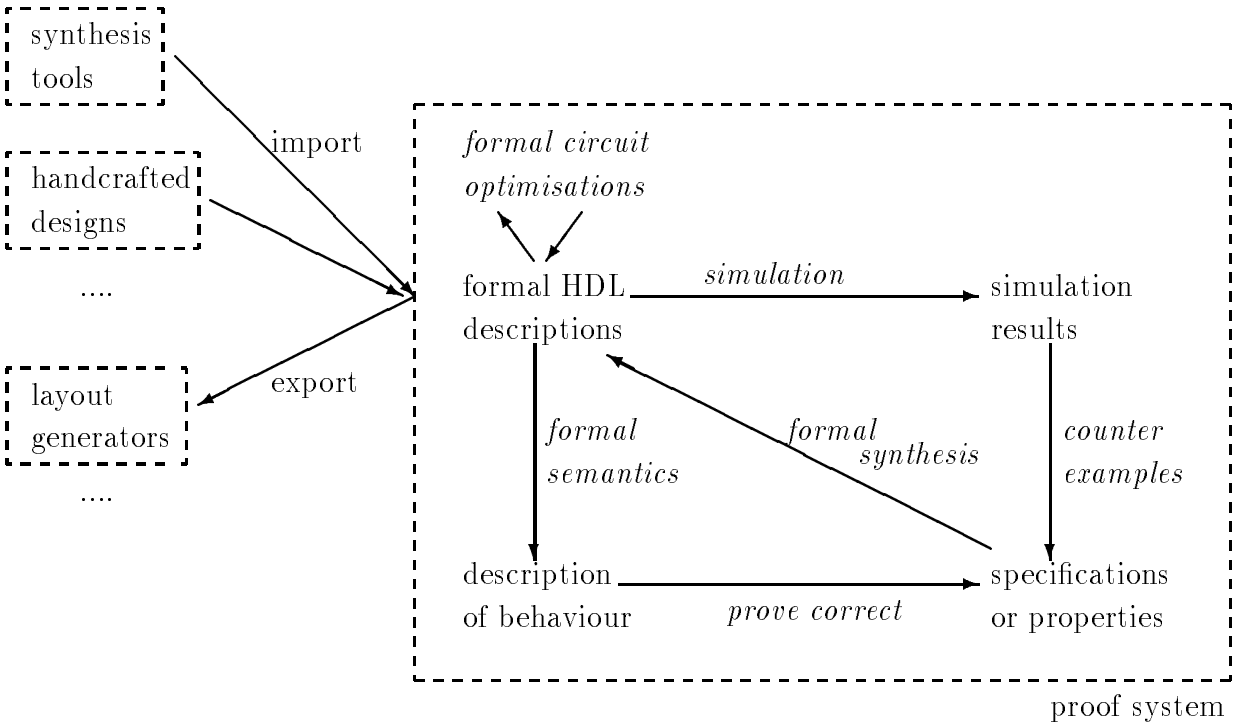
```
LET INIT ?bit
    REC loop = IF loop MATCHES lo THEN hi ELSE lo
IN loop
```

If the wire `loop` carries the signal `hi`, the output of the `NOT` gate will be `lo`. Thus the input changes to `lo`, followed by a change of output to `hi`, which was our starting point.

Similar oscillating behaviour is obtained for an initial input `lo`. The semantic model used for picoELLA results in the output `?bit`, which is the undefined, or 'don't know' value, reflecting accurately the intuition that the output is not stable. In fact, the semantics computes the least fixed point solution of the circuit in a finite number of steps. It is important to realise that these are results concerning *all* circuits, not particular instances. The totality result depends on the monotonicity of the dynamic semantics; if we have more information about the input of a circuit, we can say more about its output. Loosely, if the input becomes more defined, the output becomes more defined. The reduction function also preserves the 'shape' of the program (an adder does not become a multiplier after some time!) In other words, only the contents of delays changes over time. A detailed account of the embedding may be found in [12, 13].

4 Applications

Explicitly separating the structure and behaviour of circuits leads not only to a conceptually clearer picture, it also permits a number of different applications to be combined in one overall framework. Consider the following diagram:



Starting at 'formal HDL descriptions', we can derive properties about the formal semantics in the proof system. However, it is also possible to derive properties about individual circuits, which can then be used, for example, to show that they satisfy their specifications. In the remainder of this section we discuss three other applications: formal symbolic simulation, synthesis of circuits both interactively and using functions, and transformations of hardware.

4.1 Formal Symbolic Simulation

We can derive operational semantics rules such as `reduceSeqCons` and `reduceDelay` shown previously from the definition of the semantic function `reduce`. These rules may be used to check semantic derivation trees: given a circuit, input, and output, find a corresponding theorem. This is not terribly useful, because we have to supply the output from the start. However, recall that the LAMBDA proof system has meta-variables which may be flexible. Flexible meta-variables may be instantiated or specialised throughout a proof. Thus, rather than trying to prove, for example, the following theorem (an `AND` gate with input `(hi,lo)` outputs `lo`):

```
--------------------
⊢ (env_ ⊢ IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo
⇒ (lo,IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo) :  Type 1)
```

which has the input `(hi,lo)` and the output `(lo,IF (hi, lo) MATCHES (hi,hi) THEN hi ELSE lo)` hard-wired in, we derive instead a more general rule with a symbolic output (*out_*, *newcirc_*). We also introduce an abbreviation `AND` which is parametrised on the input circuit *circ_*.

```
⊢ out_ == (case match (hi,hi) out_1 of uu => bit | tt => hi | ff => lo)
⊢ (env_ ⊢ circ_ ⇒ (out_1,circ'_) :  TyTuple (Type 1,Type 1))
--------------------
⊢ (env_ ⊢ AND#(circ_) ⇒ (out_,AND#(circ'_)) :  Type 1)
> val reduceAND = popGoal();
val reduceAND = ?  :  rule
```

The semantic derivation for an `AND` gate with a general input has been collapsed into one derived rule `reduceAND`. It states that to evaluate an `AND` gate, one has to evaluate the circuit *circ_* which supplies its input (the first premise), and then has to check which branch of the `IF` statement is taken (the second premise). By deriving this sort of rule first for simple gates, then for components built from these gates, one can build 'skeleton rules', and computations can be speeded up greatly.

To return to our motivation for this rule: to compute the output of an `AND` gate with input `(hi,lo)` we apply the above rule schema to the goal

```
env_ ⊢ AND#((hi,lo)) ⇒ (out_,newcirc_) :  t_
```

where *out_*, *newcirc_*, and *t_* are flexible meta-variables (*i.e.* they may be instantiated with other terms). Indeed, applying `reduceAND` to the goal, and discharging the two premises will specialise *out_*, *newcirc_*, and *t_* to `lo`, `AND#((hi,lo))`, and `Type 1` respectively. While this is interesting, it would be much faster to use a real simulator for the HDL to simulate fixed value inputs. Consider, however, that for both `(lo,lo)` and `(lo,hi)` the output will be `lo`. In a conventional simulator we would need to simulate the `AND` gate twice, but using the proof system we can derive the output `lo` from the input `(lo,`x`)`, where x is a proof system variable. The fact that we can obtain an output without instantiating x with `lo` and `hi` is very important. It removes the need to *exhaustively simulate* the circuit: we can use the symbolic capabilities of the proof system to cut down the number of inputs or test vectors. Exhaustive simulation is not feasible for

large circuits because the number of possible input patterns grows exponentially with the number of inputs. Symbolic variables such as x are not to be confused with 'don't care' values, which are part of the value domain. Although in this particular case 'don't care' values could have been used to the same effect, the former may be arbitrary formulae, introducing relationships between inputs, *e.g.* `AND#(`x`,`$\neg x$`)`.

If we generalise this even further we can leave the input completely abstract (*in_* say) to derive a symbolic output:

```
(if in_ = (hi,hi) then hi else lo,AND#(in_))
```

(This holds only if the input is restricted to values without 'don't know' elements.) The computation has been pushed completely into the answer which is not possible with 'don't care' values. We could do this for the complete circuit, but the resulting term would probably be as complicated as the circuit description itself. We have progressed from simple values as inputs and outputs, via 'don't care' values [20], to symbolic values and formulae [4, 14].

While current symbolic simulators can handle the above examples, the use of a proof system becomes essential when we deal with parametrised circuits, and circuit specifications. The `AND` gate, as defined above, takes its input circuit as a parameter. This is a trivial example of the use of *plug-in* components, or abstract hardware. To simulate a circuit which contains a hole or plug-in component, the input-output behaviour of the missing subcomponent needs to be supplied. The specification of the circuit, which should be available, can be used for this purpose. In fact, this strategy allows different people to work on one design in simultaneously; to simulate the whole design, parts of which may not be finished yet, specifications of the missing subcomponents are used. When a part has been completed it must be proved to satisfy its specification. After this has been done using the proof system, the part may be inserted into its context in the larger design. However, as the proof of correctness implies that the specification describes the behaviour of the implementation[9] it makes sense to continue to use the specification for simulation. In general, a specification will state the behaviour at a higher level of abstraction (for example, an adder is specified as adding two natural number, rather than two bit vectors). Thus, instead of computing an addition using the bit vectors in the implementation, the bit vectors are abstracted to natural numbers which are then added using natural number addition and the result converted back to a bit vector. In other words: the sum would be computed by the expression $bitsof((natof\ x + natof\ y)\ mod\ 2^N)$, where *natof* is the abstraction function, and *bitsof* its inverse. This technique should substantially speed up simulations. Although mixed-level simulators handle part of this functionality, they cannot guarantee that the high and low-level descriptions, interpreted as the specification and implementation respectively, have the same behaviour (other than through exhaustive simulation, which is not acceptable.) Moreover, the specifications used in the proof system need not be restricted to HDL programs: they could be any term in the logic with the appropriate result type, possibly non-algorithmic or non-executable.

[9] The relationship between implementation and specification is probably more subtle than this, but the argument remains the same.

4.2 Formal Circuit Synthesis

Two distinct types of synthesis are possible in the proof system: functions which generate correct hardware [3], and interactive synthesis methodologies.

Verified Hardware Generators

We can write functions which manipulate hardware descriptions: we have already encountered the dynamic semantics function `reduce`. Consider the following function which, given a natural number N, returns the description of an N bit adder.

```
fun nadd onebitadder (S 0) x = onebitadder x |
    nadd onebitadder (S (S n)) x =
         LET x IN   (* (((x_{N+1}, x̄), (y_{N+1}, ȳ)), c_0) *)
         LET nadd onebitadder (S n)
             (((Var 0)[1][1][2], (Var 0)[1][2][2]), (Var 0)[2]) IN
         LET onebitadder (((Var 1)[1][1][1], (Var 1)[1][2][1]),
                          (Var 0)[2]) IN
             (((Var 0)[1], (Var 1)[1]),     (* sum *)
              (Var 0)[2])                   (* carry *);
```

Unfortunately, due to the sparse nature of picoELLA, circuit descriptions are not always very readable. `nadd: (expr -> expr) -> natural -> expr -> expr` is a partial function: there is no such a thing as a zero bit adder. A one bit adder with input $((x_0, y_0), c_0)$ uses the full adder component *onebitadder*, on which `nadd` is also parametrised. Am $N+1$ bit adder with input $(((x_N, \overline{x}), (y_N, \overline{y})), c_0)$ uses an N bit adder with input $((\overline{x}, \overline{y}), c_0)$ connected to a full adder with input $((x_N, y_N), c_N)$.

The most important aspect of hardware generating functions is that they may be proven correct. That is, it is possible to prove a correctness statement for all word sizes:

$$\begin{array}{l} \forall c.\ \texttt{FULLADDER\_SPEC\#}(c) \rightarrow \\ \quad \forall N > 0.\ \forall x, y, c, s, c', e.\ (\vdash\ \texttt{nadd}\ \ c\ N\ ((x, y), c)\ \Rightarrow\ ((s, c'), e)) \rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{NBITADDER\_SPEC\#}(N, x, y, c, s, c') \end{array}$$

Thus, given a correct full adder c, all N bit adders generated by `nadd` using it, are guaranteed to be correct.

Interactive Synthesis

A different approach to hardware synthesis is to provide a methodology for interactively constructing correct hardware. A number of valid design rules are given, and a circuit is built using these rules only. In [18] hardware is described in term of higher-order logic, but their approach can also be used with HDL descriptions. Fourman *et al.* [10] also synthesise hardware interactively, but use flexible meta-variables to represent circuits which are still to be refined, and are therefore not limited to a fixed set of design rules. This work uses the LAMBDA system and can also be adapted to use HDL descriptions as the underlying representation for circuits.

Operational semantics rules such as `reduceDelay` of Section 3 can be used to synthesise circuits. Initially the circuit is a flexible meta-variable: the circuit is completely unconstrained. Applying operational semantics rules does not only restrict outputs, it also specialises the circuit. For example, applying the `reduceTuple` to the circuit *circ_* forces it to become a tuple (*circ_1*, *circ_2*), with a tuple (*out_1*, *out_2*) as output. Successively applying operational semantics rules simultaneously constructs the circuit and its output. To make this approach useful, sufficiently high-level building blocks must be provided. To construct a circuit in a top-down fashion (*i.e.* successively refine subcomponents) rules are applied in a goal-directed, or backward manner. Conversely, forward rule application corresponds to bottom-up synthesis [13, Chapter 5]. The results of both top-down and bottom-up synthesis must be verified after they have been designed, in contrast to [18], and to lesser extent [10], where a design and its proof of correctness are constructed simultaneously.

4.3 Correct Hardware Optimisations

Hardware designs, whether originating from synthesis functions or hand designs, can often be optimised by applying transformations. Rather than trying to produce an efficient design from the outset, circuit optimisations can be used effectively to massage an existing design to produce a smaller layout, faster chip, *etc.* Optimisations can be treated formally in our framework: transformations can be rule-based [5], or functions can be written to detect certain patterns and replace them by others [13]. Both forms can be verified by showing that the behaviour of a circuit is either unchanged by the optimisation, or more favourable in some sense; *e.g.* lower latency or higher throughput. If a transformation changes an aspect of a circuit description which is not addressed in the semantics (*e.g.* layout area) the two descriptions will be behaviourally equivalent, even though one version of the circuit will be preferable to the other. Other aspects, such as latency, will be derivable using the semantics. In this case an optimisation will change the circuit's behaviour but in an acceptable fashion. It is possible to base an entire design style upon transformations, as described in [5].

5 Conclusions

Hardware description languages are widely used to aid the design process. A mathematical basis for HDLs allows formal methods, implemented in proof systems, to be applied to hardware design and description. Our approach clarifies important issues concerning behaviour and structure which have not been addressed properly in formal hardware verification. The definition of a formal semantics for a subset of the hardware description language ELLA, and its encoding in the LAMBDA proof assistant show how various methodologies can be integrated. The ability to prove properties of the simulator mechanism for the HDL, symbolic and mixed-level simulation, various types of synthesis, and transformations of hardware have all been treated formally. A number of practical issues have to be addressed; although ELLA may be translated into the picoELLA language, it is too restrictive for more than toy examples, and verification of individual circuits tended to be very slow. However, we believe that our aim, to find a suitable integrated methodology for hardware design and verification using HDLs, has been successful.

References

[1] H Barringer, G Gough, T Longshaw, B Monahan, M Peim, and A Williams. Semantics and verification for boolean kernel ELLA using IO automata. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 65–90. ESPRIT CHARME, North Holland, June 1991.

[2] Howard Barringer, Graham Gough, Brian Monahan, and Alan 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.

[3] Bishop C Brock and Warren A Hunt, Jr. The formalization of a simple hardware description language. In Luc Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 778–792, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers.

[4] Randal E Bryant and Carl-John Seger. Formal verification of digital circuits using symbolic ternary system models. Technical Report CMU-CS-90-131, School of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213, May 1990.

[5] Holger Busch. Transformational design in a theorem prover. In V Stavridou, T F Melham, and R T Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 175–196. IFIP TC10/WG 10.2, North Holland, June 1992.

[6] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, June 1989.

[7] N D de Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag Math.*, 34:381–392, 1972.

[8] James R Duley and Donald L Dietmeyer. A digital system design language (DDL). *IEEE Transactions on Computers*, C-17(9):850–861, September 1968.

[9] R W Floyd. Assigning meanings to programs. *Proceedings of American Mathematical Society, Symposia in Applied Mathematics*, 19:19–32, 1967.

[10] Michael P Fourman and Eleanor M Mayger. Formally based system design – interactive hardware scheduling. In G Musgrave and U Lauther, editors, *International Conference on VLSI*, Munich, 1989.

[11] Mick Francis, Simon Finn, and Ellie Mayger. *Reference Manual for the Lambda System.* Abstract Hardware Limited, version 3.2, November 1990.

[12] K G W Goossens. Embedding a CHDDL in a proof system. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 359–374. ESPRIT CHARME, North Holland, June 1991. Also as LFCS Report ECS-LFCS-91-155.

[13] 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.

[14] K G W Goossens. Operational semantics based formal symbolic simulation. In Luc Claesen and Michael Gordon, editors, *Higher Order Logic Theorem Proving and Its Applications*, pages 487–506, Leuven, Belgium, September 1992. North Holland. A longer version is available as LFCS Report ECS-LFCS-92-231.

[15] K G W Goossens. Structure and behaviour in hardware verification. In *Higher Order*

Logic Theorem Proving and Its Applications, Vancouver, Canada, August 1993.

[16] Andrew D Gordon. The formal definition of a synchronous hardware description language in higher order logic. In *International Conference on Computer Design*, October 1992.

[17] Mike Gordon. Why higher-order logic is a good formalisation for specifying and verifying hardware. In G Milne and P A Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177, Amsterdam, 1985. North Holland.

[18] F K Hanna, M Longley, and N Daeche. Formal synthesis of digital systems. In Luc Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 532–548, Amsterdam, November 1989. IMEC-IFIP International Workshop, Elsevier Science Publishers.

[19] Robert Harper, Robin Milner, and Mads Tofte. The definition of standard ML version 3. LFCS Report Series ECS-LFCS-89-81, LFCS, Department of Computer Science, University of Edinburgh, May 1989.

[20] John P Hayes. Digital simulation with multiple logic values. *IEEE Transactions on Computer-Aided Design*, CAD-5(2):274–283, April 1986.

[21] M G Hill. The dynamic semantics of kernel ELLA. Memorandum 4630, Defence Research Agency, Malvern, UK, August 1992.

[22] C A R Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[23] IEEE computer, December 1974. Special Edition on Hardware Description Languages.

[24] The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY10017 USA. *IEEE Standard VHDL Language Reference Manual*, IEEE std 1076-1987 edition, 1988.

[25] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[26] Gordon Plotkin. A structural approach to operational semantics. Technical Report FN-19, Computer Science Department, Aarhus University (DAIMI), 1981.

[27] 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.

[28] V Stavridou, J A Goguen, S M Elker, and S N Aloneftis. FUNNEL: A CHDL with formal semantics. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 115–137. ESPRIT CHARME, North Holland, June 1991.

[29] Joseph E Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[30] John P van Tassel. A formalisation of the VHDL simulation cycle. Technical Report 249, University of Cambridge Computer Laboratory, March 1992.

[31] Philip A Wilsey, Timothy J McBrayer, and David Sims. Towards a formal model of VLSI systems compatible with VHDL. In A Halaas and P B Denyer, editors, *VLSI '91*, pages 6a.2.1–6a.2.12, Edinburgh, Scotland, August 1991. IFIP TC 10/WG 10.5.

[32] Michael Yoeli. *Formal Verification of Hardware Design*. IEEE Computer Society Press, 1990. IEEE Computer Society Press Tutorial.