# Operational Semantics Based Formal Symbolic Simulation

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**

This paper describes the development of progressively more powerful and abstract hardware simulators. A small computer hardware design and description language picoELLA is then introduced, followed by its formal semantics. Using a number of small examples, we will then show the how this formal semantics may be used within a proof system as a sophisticated simulation tool. Examples include some full adders, a general $N$ bit adder, and two parity checkers.

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

# 1 Introduction

This introduction describes the development of various kinds of hardware simulators. Following this, a small HDL called picoELLA, is introduced in section 2. Its formal semantics, and a brief account of this semantics' embedding in a proof system are described in section 3. Section 4 illustrates the use of the semantics in the capacity of a symbolic simulator, as described in the remainder of this introduction. Finally, integration with other design and verification methodologies will be discussed.

Modern hardware designs are complex. Conventional methods take many iterations to arrive at an acceptable implementation. Though exhaustive testing might allow designers to achieve a high degree of confidence in small circuits, exhaustive testing of modern designs is impossible both on time complexity and cost grounds.

Breadboarding is the process of constructing the design and then using this prototype to perform tests. This is only feasible for small circuits. With greater integration and density of components this method becomes prohibitively expensive. The first move away from using real hardware to test a design is to *simulate* the circuit. Greater complexity encourages the use of structured circuit descriptions, leading to hardware description

languages such as ELLA[1] [6] and VHDL [13]. It is possible to design simulators for such languages which model the behaviour of a circuit described in the language.

One of the problems with both breadboarding and value simulation is that for any substantial circuit the number of possible inputs (or *test vectors*) becomes very large. Circuits with internal state are even harder to verify in this manner. By introducing extra values in the value domain, such as don't know and don't care, the number of test vectors may be reduced substantially. If a particular input is irrelevant for a particular test, its value can be set to don't care, instead of having to simulate the test twice, with the value set to true and false respectively.

The MOSSYM simulator [5] is not limited to fixed values as input, but also allows symbolic variables and boolean formulae. That is, we may set an input to the symbolic variable $x$, say. Wherever $x$ appears we cannot assume anything about its value, so that the result of the operation may be a formula. Note that $x$ is not an extra value in the value domain, but ranges over *all* these values. Of course, this puts an extra burden on the simulator which now needs to be able to handle arbitrary formulae instead of simple values. It may also require algebraic capabilities to simplify intermediate formulae. In theory, we need to do only one simulation; the one with all the input values set to variables. The result would be an expression which would describe the circuit's behaviour. However, this expression may be as complex as the circuit description.

In MOSSYM we have an asymmetry: we are permitted abstraction over data but not over circuits. In other words, we may have symbolic variables ranging over data values, but we are not allowed circuits containing symbolic variables. Symbolic variables are abstract hardware. This idea is not as strange as it may seem; plug-in components are in effect abstract hardware, certainly as long as the circuit is under development. Why would it be useful to have this capability? It would seem that, since we are dealing with the design of a certain circuit, we would only want to simulate that circuit. Consider, however, that large circuits are designed in a modular fashion to allow a number of people to work on separate parts of a circuit at the same time. When a subcomponent is ready, it has to be simulated in a larger context, all of which may not be completed. The availability of an abstract implementation for unfinished parts of the design would enable the component to be simulated in its correct context. A suitable simulator would allow the evaluation of a circuit containing a mixture of concrete and abstract components. Of course, certain properties of the abstract components may be needed to arrive at an output, but these should be available from their specifications. We use the specification of the abstract components to simulate them as long they are not available. Modifying a conventional simulator to deal with these extensions would completely transform it. Mathematical proof systems, in contrast, already have the capability to deal with abstract values of any sort. (This assumes that we work within a suitably powerful logic, such as higher order logic.) If we use a HDL to describe circuits and as input to the simulator, the HDL needs to have a precise mathematical definition to be used in conjunction with a proof system. The approach we advocate here uses a formal definition of the behaviour of a HDL and uses it within a proof system to provide simulation capabilities.

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

# 2 picoELLA

picoELLA is derived from ELLA [6]. It contains the 'active ingredients' but lacks its syntactic sugar. picoELLA contains the following constructs:

Type definitions; these are either enumerated types, such as `TYPE Signal = Hi | Lo`, or tuple types such as `TYPE twobool = bool * bool`.

Local declarations. `LET x = e IN e'` defines a local name `x` for a wire (or signal), which may be used in `e'`. Circuit descriptions may be structured using these declarations. Multiple use of a name such as `x` corresponds to a fan-out of the signal. Recursive declarations allow the description of feedback, an example of which is shown at the end of this section.

Constants are built up using constructors such as `Hi`, or `?type` representing the undefined, or don't know value of type `type`, or tuples of constants.

Tuples and indexing are straightforward. $(e_1,e_2)[i]$ behaves the same as $e_i$, for example. Although, strictly speaking, indexing is not needed it facilitates decomposition of circuit descriptions, as we shall see in the examples.

The `IF` statement, or multiplexor, has the form `IF e MATCHES chooser THEN e1 ELSE e2`. A chooser is a pattern against which the output of circuit `e` is matched. If it matches, the output of the first branch is the result of the `IF`. If the output of `e` and the chooser do not match the `ELSE` part is chosen. A third possibility is that the output of `e` is insufficiently defined to decide between the two branches. In this case, the undefined value `?type` (of the correct type) is the result. For example, consider the NOT gate `IF ?Signal MATCHES Hi THEN Lo ELSE Hi`. If the undefined, or don't know signal `?Signal` turned out to be `Hi` then the output would be `Lo`. On the other hand, if it were `Lo`, the output would be `Hi`. The undefined output `?Signal` therefore reflects our intuition that we don't know what the output should be.

The delay construct introduces a discrete and linear time base into the language, which may be modelled using the natural numbers. The output from circuit `e` at time $t$ will be output by $\texttt{Delay(}c_t\texttt{,e)}$ 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 is explicit in its description. This contrasts with other languages, where it resides in a memory or store. picoELLA dispenses with this, and can use a simpler environment instead, as we shall see later. 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 dynamic semantics is therefore *environment* $\rightarrow$ *expression* $\rightarrow$ (*value* $\times$ *expression*).

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 ?Signal
    REC xor = DELAY (Hi, IF (input,xor) MATCHES (Hi,Lo)|(Lo,Hi)
                         THEN Hi ELSE Lo)
IN xor
```

We will return to this example in section 4.3.

# 3 A picoELLA Semantics and Its Embedding in Higher Order Logic

Few HDLs have a precise definition. In practice the simulator 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 a HDL, *i.e.* how a simulator should behave.[2] For example, a structural operational semantics [15] defines the behaviour of a construct in terms of its subexpressions. 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). Various subsets of ELLA [10, 1], Funnel [16] and VHDL subsets [18, 17] are some of the languages that have been given formal definitions.

Although a formal semantics is very useful as a mathematical reference manual, one can also use formal semantics as the basis for design tools. By embedding the semantics in a logical system, supported by a proof assistant such as LAMBDA[3] [8] it is possible to provide support for the formal development of circuit design [11].

The semantics for picoELLA [10] takes the form of a structural operational semantics. 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 semantics rules fall into two categories; those dealing with time, and those dealing with the evaluation of expressions within one time step. The ReduceSeqCons rule falls into the first class:

$$\frac{\Gamma' \vdash\ expr_t \Rightarrow\ o_t,\ expr_{t+1} \qquad tl,\ \Gamma \vdash\ expr_{t+1} \Rightarrow\ tl',\ expr_{t+N}}{i_t :: tl,\ \Gamma \vdash\ expr_t \Rightarrow\ o_t :: tl',\ expr_{t+N}}$$

Here $expr_t$ is the program at time $t$, $\Gamma$ the environment in which the program runs, and $i_t :: tl$ the input stream. $\Gamma'$ is $\Gamma$ 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 $\Gamma'$. 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 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 $\Gamma$, both subexpressions must be evaluated in the same environment. The rule for the delay shows the use of the embedded state:

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

[2] 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.

[3] LAMBDA and DIALOG are products of Abstract Hardware Limited.

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.

The semantics described above, has been embedded in the LAMBDA proof system. The LAMBDA proof system implements a polymorphic constructive higher order logic of partial terms.[4] An existence predicate `E` is provided to reason about partial terms. Equality `==` compares two denoting objects, weak equality (or equivalence) `===` is true if either both objects do not denote, or if both denote and are equal. The functional language ML is used as a command language. A large subset of ML is also 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 characterising the new ML data type definitions, such as existence of constructors, (in)equality rules and a structural induction principle. In the case of functions rules include a rewrite rule for every function clause. These new rules may be used to define derived rules and tactics. Tacticals can be used to combine tactics into rewrite strategies, or symbolic simulation commands. Examples are `OpSemTac` and `safeOpSemAllTac` in section 4.1.

A type *const* has been encoded using the ML definition system, representing picoELLA constants:

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

`Cons(i,t)` encodes the $\mathtt{i}^{th}$ 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. To illustrate structural induction we consider the `choosers` data type used to encode patterns in the `IF` statement.

```
datatype choosers = C of const
                  | B of choosers * choosers
                  | T of choosers * choosers;
```

`B(ch,ch')` represents the bar, or disjunctive chooser; it matches with a constant if at least one of `ch` and `ch'` does. `T(ch,ch')` is the tuple, or pairing chooser, matching if both subchoosers do. `C c` is the constant chooser. `C (Cons (0,type))` represents the *wild card* chooser `type`; it always matches. It is not allowed to match for the bottom value `?type`, as this would permit non-monotone circuit descriptions. LAMBDA returns the following choosers structural induction rule for this data type.

```
[3] E r1, E r, P#(r1), P#(r) |- P#(T (r1,r))
[2] E r3, E r2, P#(r3), P#(r2) |- P#(B (r3,r2))
[1] E r4 |- P#(C r4)
--------------------
E w |- P#(w)
```

There are three premisses, each containing some hypotheses. `E r4` is an *existence* hypothesis, asserting that `r4` denotes. `P#(r1)` states that the property `P` holds for `r1`. To prove a property `P` of all choosers `w` three subgoals must be proved: in case of the second premise, for example, it must be shown that `P` holds for `B(r3,r2)` provided it holds for `r3`

[4] Note that we use LAMBDA version 3.2. The more recent version 4.0 uses a different logic.

and r2, and r3 and r2 denote. 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 * choosers
              | 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. 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 encoding `LET x = a IN LET y = b IN (x,(x,y))` by `Let (a, Let (b, Tuple (Var 1, Tuple (Var 1, Var 0))))`. The de Bruijn encoding was sufficient for our purposes because the environment is used only as a stack. Work using the HOL system has usually represented names by strings; the value environment has the type `string -> const` [14, 17]. Finally, the dynamic semantics can be defined as a function Reduce. Its type is `Reduce: const list -> expr -> (const * expr)`, where `const list` represents the value environment.

```
fun Reduce l (Let (e,e')) =
           let val (c, f) = Reduce l e
               val (c', f') = Reduce (c::l) e'
           in  (c', Let (f,f'))
           end |
    Reduce l (Var n) = (elem l n, Var n) |
    Reduce l (If (e,e',e'',ch)) =
           let val (c,d) = Reduce l e
               val (c',d') = Reduce l e'
               val (c'',d'') = Reduce l e''
           in  ( case match ch c of
                   tt => c' |
                   ff => c'' |
                   uu => bottom c', If (d,d',d'',ch) )
           end | ...;
```

The LET statement reduces the defining expression, and pushes the value result on the stack l (*i.e.* stores it in the environment). Evaluating a name corresponds to a lookup in the environment $\Gamma$ in the dynamic semantics, and a lookup in the stack l in the embedding.

The IF construct evaluates all of its subexpressions. It then returns (tt) the result of the first branch if we have a definite match; or (ff) if we have a definite no-match, the

result of the second branch; or (uu) a bottom value of the appropriate type if we cannot decide between the two branches.

We have proved a number of properties of the embedded semantics using LAMBDA. For example, the reduction function is monotone, that is, if the input becomes more defined, the output becomes more defined. Also, the reduction function 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. Moreover, we have shown that even in the presence of delayless feedback loops the reduction function terminates in a finite number of steps. In fact, the semantics computes the least fixed point solution of the circuit. It is important to realise that these are results concerning *all* circuits, not particular instances. A more detailed account of the embedding may be found in [11, 9]. For the remainder of this paper, with the exception of subsection 4.3 which deals with feed-back, an embedding without the `LET REC` has been used. The operational semantics rules dealing with `IF` are slightly simpler in the embedding without the `LET REC`. Other rules are identical.

Using these definitions, and derived properties, the operational semantics rules described at the start of this section may be derived within the proof system.[5] These rules encode both the static and dynamic semantics.

The rule `ReduceSeqCons` corresponds to ReduceSeqCons previously. Rules 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 $\Gamma'$ in the paper rule. `circ_` is then evaluated 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. It is a pretty printed version of:

```
E instream_, E env_, E circ1_, E t_
|- ReduceSeq env_ circ1_ instream_ == (outstream_,circ2_)
E (i1_ ::  env_), E circ_, E t_
|- typeOfExpr (map typeOfConst (i1_ ::  env_)) circ_ == (t_,true)
/\ Reduce (i1_ ::  env_) circ_ == (o1_,circ1_)
--------------------
E (i1_ ::  instream_), E env_, E circ_, E t_
|- ReduceSeq env_ circ_ (i1_ ::  instream_) == (o1_ ::outstream_,circ2_)
```

Henceforth we will only show pretty printed output. Unfortunately, no quotation/anti-quotation system is available, so that any input must still use the raw syntax. The rule for the multiplexor, `ReduceIf'` is similar:

[5] This is in contrast with work by van Tassel [17], which starts directly with the semantic rules. Using the HOL inductive relation package more general relational semantics may be encoded. In LAMBDA version 3.2 we are limited to functional semantics.

```
[6] |- E t_
[5] |- o3_ == (case match chooser_ out_ of
uu => bottom o1_ | tt => o1_ | ff => o2_)
[4] |- chooser_ :  t_
|- (env_ |- branch2_ => (o2_,branch2'_) :  t1_)
|- (env_ |- branch1_ => (o1_,branch1'_) :  t1_)
|- (env_ |- circ_ => (out_,circ'_) :  t_)
--------------------
|- (env_ |- IF circ_ MATCHES chooser_ THEN branch1_ ELSE branch2_ =>
(o3_,IF circ'_ MATCHES chooser_ THEN branch1'_ ELSE branch2'_) :  t1_)
```

There are some extra hypotheses (`[4]` and `[6]`) dealing with the static semantics: the choosers must be well-typed and have (denoting) type `t_`. Note that `branch1_` and `branch2_` must have the same type `t1_`, which is also the type of whole `IF`. The output of the `IF` is computed in premise five. The three cases (match, no match and don't know) are represented in the `case` statement by `tt`, `ff` and `uu` respectively. The `IF` is strict; both branches must always be evaluated. Four other rules dealing with the `IF` are particular instantiations of this rule, as we shall see later.

It is important to realise that `circ_`, `t_`, *etc.* are *meta-variables*. The `ReduceIf'` rule is really a rule schema, which may be instantiated in an infinite number of different ways. When it is applied to a particular `IF` statement such as `IF Hi MATCHES Hi THEN Lo ELSE Hi`, `circ_`, `chooser_`, `branch1_` and `branch2_` will be unified with `Hi`, `Hi`, `Lo`, and `Hi` respectively. This unification is reflected in every place where these variables occur in the rule. The unification works both ways, meta-variables in a rule are unified to the current goal so that the rule applies (as in the example below). But meta-variables in the goal may also be unified (specialised, 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.

# 4 Examples

In this section we will illustrate the possible uses of an embedded operational semantics. First we will simulate a simple AND gate in various ways to illustrate the basic principles. Following this, some one bit adders and a general $N$ bit adder, parametrised on the word size and one bit adder subcomponent, will be shown. Finally, two parity checker implementations will be discussed.

## 4.1 A Simple AND Gate

An AND gate may be described in picoELLA as

```
IF e MATCHES (Hi,Hi) THEN Hi ELSE Lo
```

or, using the syntax of the embedding:

```
If (e, Const Hi, Const Lo, T (C Hi, C Hi));
```

Here `e` is the input to the circuit. `Signal`, `Hi` and `Lo` have been defined as `Cons(0,1)`, `Cons(1,1)` and `Cons(2,1)` respectively. All of `Signal`, `Hi` and `Lo` have type `Type 1`. We will simulate an AND gate with `(Hi,Lo)` as input using the `ReduceIfFf` and `ReduceConst` rules. The rule `ReduceIfFf` is comparable to the rule `ReduceIf'` of the previous page, but always chooses the `ELSE` branch.

```
> apprl ReduceIfFf;

***** Level 2 *****
[6] |- E t_
[5] |- match (Hi, Hi) out_ == ff
[4] |- (Hi, Hi) :  t_
[3] |- (env_ |- Lo => (Lo, Lo) :  Type 1)
[2] |- (env_ |- Hi => (o1_, Hi) :  Type 1)
[1] |- (env_ |- (Hi, Lo) => (out_, (Hi, Lo)) :  t_)
--------------------
|- (env_ |- IF (Hi, Lo) MATCHES (Hi,Hi) THEN Hi ELSE Lo
=> (Lo,IF (Hi, Lo) MATCHES (Hi,Hi) THEN Hi ELSE Lo) :  Type 1)
```

We now have six subgoals to prove, the first of which deals with the input to the `IF`. The second and third subgoals compute the `THEN` and `ELSE` branches respectively. As stated earlier, both branches must be evaluated, because the result circuit is always used to describe the `IF` at the next time step. Note, however, that the value output `o1_` does not appear in the output. The fourth premise states that the chooser must be well-typed; in this case it has type `t_`. `t_` is an as yet uninstantiated meta-variable. As we shall see below, evaluating premise 1 forces `t_` to become a tuple type. We also have to prove that the type denotes in premise 6. Subgoal five expresses the constraint that we choose the `ELSE` part of the `IF`; the result `out_` of the input circuit must not match with the chooser.

We may now apply `ReduceTuple` to reduce the tuple in premise 1 to two subgoals. Following this we apply `ReduceConst` to premises one to four:

```
> applyTacn [1,2,3,4] (doRule ReduceConst);

***** Level 4 *****
|- E (TyTuple (t1_,t2_))
[6] |- match (Hi, Hi) (Hi,Lo) == ff
|- (Hi, Hi) :  TyTuple (t1_,t2_)
[4] |- Lo :  Type 1
[3] |- Hi :  Type 1
[2] |- Lo :  t2_
[1] |- Hi :  t1_
--------------------
|- (env_ |- IF (Hi, Lo) MATCHES (Hi,Hi) THEN Hi ELSE Lo
=> (Lo,IF (Hi, Lo) MATCHES (Hi,Hi) THEN Hi ELSE Lo) :  Type 1)
```

The tactical `applyTacn l t` applies tactic `t` to all premises in the list `l`. `doRule` converts

a rule into the tactic which applies the rule if it is applicable, and fails otherwise. `tryRule`, on the other hand, is the identity tactic if the rule fails to apply. `tryRules` and `doRules` are similar functions operating on lists of rules.

As mentioned earlier, the type of the chooser has been constrained to a less general type `TyTuple (t1_,t2_)`. Evaluating premises one and two will specialise it further to `TyTuple (Type 1,Type 1)`, as the type of `Hi` and `Lo` is `Type 1`. All the subgoals, except `[6]`, are now dealing with the static semantics, or typing of terms. It makes sense to deal with the static and dynamic semantics simultaneously because they are both structural semantics. Moreover, the dynamic semantics only evaluates well-typed expressions.

Using `applyTacn [1,2,3,4] (doRule ReduceHi elseR ReduceLo)` we discharge premises one to four. Using `ReduceMatchTac`, a tactic which rewrites expressions involving `match`, we prove premise six. We also discharge the static typing of the chooser.

```
> applyTac (doRules[ReduceT,ReduceC,ReduceHi]);

***** Level 7 *****
|- E (TyTuple (Type 1,Type 1))
--------------------
|- (env_ |- IF (Hi, Lo) MATCHES (Hi,Hi) THEN Hi ELSE Lo
=> (Lo,IF (Hi, Lo) MATCHES (Hi,Hi) THEN Hi ELSE Lo) :  Type 1)
> applyTac (doRules[ReduceTyTuple,ReduceType,ReduceSn,Reduce0]);

***** Level 8 *****
--------------------
|- (env_ |- IF (Hi, Lo) MATCHES (Hi,Hi) THEN Hi ELSE Lo
=> (Lo,IF (Hi, Lo) MATCHES (Hi,Hi) THEN Hi ELSE Lo) :  Type 1)
> val example1a = popGoal();
val example1a = ?  :  rule
```

Note that `doRules [r1,r2]` is a tactic which applies rule `r1` and then applies `r2` to all resulting subgoals. Thus `ReduceC` and `ReduceHi` are applied to both subgoals resulting from `ReduceT`. The theorem is saved as `example1a` so that we can apply this derivation in one step in the future.

While this is very instructive, it becomes tedious very quickly to this sort of proof by hand. Tactics may be used to great advantage in this sort of regular reasoning. The whole previous example could have been done using one general purpose tactic:

```
val OpSemTac = (repeatT (nonTrivT (tryRules OpSemRules))) thenT
               (tryT (theoremT ReduceMatchTac)) thenT
               (tryT (theoremT ReduceTypeTac));
applyTac OpSemTac;
```

This tactic repeatedly applies one or more of the standard operational semantics rules until none apply. It then applies `ReduceMatchTac` followed by `ReduceTypeTac`, to rewrite any typing subgoals. These last two tactics are applied to a subgoal only if they discharge it.

The circuit as it stands is not very useful, as it deals with only one particular input. Moreover, we had to supply the output from the simulation at the start! We will now

quickly redo the example, but using meta-variables as output. These will be flexible, so that they may be instantiated as we compute the output to a fixed answer. We will also use an abbreviation for the AND gate. Unlike the abbreviations for `Hi` *etc.*, it has an argument. Abbreviations are syntactic functions at the meta-level in the proof system. They are distinct from functions at the object level, such as `nadd` which we shall see later.

```
val Signal = Cons (0,1);
val Hi = Cons (1,1);
val Lo = Cons (2,1);
val AND#(e) = IF e MATCHES (Hi,Hi) THEN Hi ELSE Lo;
```

When a new goal is to be proven, all meta-variables are rigid; they cannot be (inadvertently) instantiated. In general this is what is required, because the result so proved is then more general. Every operational semantics rule has meta-variables such as `env_`, `circ_` and `t_`, which are unified with the corresponding expressions in the premise it is applied to. Consider the use of rule `ReduceIf` below, for example. In this case we want to specialise the meta-variables if required, so we make them flexible using the `flex` command. A pop-up menu shows the current subgoal, and one selects subterms by clicking on them with a mouse. The `flex` command is then automatically generated by LAMBDA, so that it may be included in proof scripts for later use. We will now unfold the abbreviation for AND, and apply `ReduceIf` which is a third rule for the `IF` statement.

```
> apprl ANDU;

***** Level 3 *****
|- (env_ |- IF circ_ MATCHES (Hi,Hi) THEN Hi ELSE Lo =>
(out_, IF h MATCHES (Hi,Hi) THEN Hi ELSE Lo :  t_)
--------------------
|- (env_ |- AND#(circ_) => (out_,AND#(h)) :  t_)
> apprl ReduceIf;

***** Level 4 *****
|- E t_1
|- T (C Hi,C Hi) :  t_1
|- (env_ |- Lo => (o2_,Lo) :  t_)
|- (env_ |- Hi => (o1_,Hi) :  t_)
|- (env_ |- circ_ => (out_,h) :  t_1)
--------------------
|- (env_ |- AND#(circ_) => (case match (Hi,Hi) out_ of
uu => bottom o1_ | tt => o1_ | ff => o2_,AND#(h)) :  t_)
```

`ReduceIf` defers the computation of the output of the `IF` by delivering a *symbolic answer*. In this case, however, we would like to have a concrete value answer rather than an expression describing what happens in the most general case. After undoing everything using `undoAll()` we prove the result we want by flexing type `t_` and circuit `newcirc_`, expanding all the abbreviations using `applyTac (doRules[ANDU,HiU,LoU,SignalU])`, finally followed by `applyTac OpSemTac`. `OpSemTac` uses the rule `ReduceIf'` rather than `ReduceIf`, so that the required answer is obtained.

```
> applyTac OpSemTac;

***** Level 5 *****
|- out_ == (case match (Cons (1,1), Cons (1,1)) out_1 of
uu => bottom (Cons (1,1)) | tt => Cons (1,1) | ff => Cons (2,1))
|- (env_ |- circ_ => (out_1,h) :  TyTuple (Type 1,Type 1))
--------------------
|- (env_ |- AND#(circ_) => (out_,AND#(h)) :  Type 1)
> val ReduceAND = popGoal();
val ReduceAND = ?  :  rule
```

The abbreviations for `Hi` *etc.* have been expanded so that this derived rule `ReduceAND` may be used in general contexts without any extra work. This derived rule may be thought of as abbreviating the whole proof tree which was generated to prove this rule. Derived rules may be used very effectively in a hierarchical manner. Simulations may be speeded up by passing rules which reduce subcircuits such as AND gates or adders in one step. An alternative approach, is to write a tactic with the same effect. A tactic would actually *replay* or recreate the proof tree, which would be as slow as rerunning the proof. The application of a derived rule, in contrast, is as fast as a primitive rule.

All of the computations we have shown so far have been within a single clock tick or time step. The following example shows how a delayed AND gate may be simulated during two time steps. At every time step the value at the head of the input stream is put on top of the stack. `Var 0` indicates the first value on the stack or environment `env_`. `Delay (c,e)` is a unit delay of expression `e`. Thus the circuit `DELAY (Signal,AND#(Var 0))` is an AND gate which takes its input from the input stream, and whose output is delayed by one time step.

```
> apprl ReduceSeqCons;

***** Level 3 *****
|- ([(Signal,Lo)], env_ |- circ1_ => (outstream_,newcirc_))
|- ((Lo,Lo) ::  env_ |- DELAY (Signal,AND#(Var 0)) => (o1_,circ1_) :  t_)
--------------------
|- ([(Lo,Lo), (Signal,Lo)], env_ |-
DELAY (Signal,AND#(Var 0)) => (o1_ ::  outstream_,newcirc_))
> apprl ReduceDelay;

***** Level 4 *****
|- ([(Signal,Lo)], env_ |- DELAY (out_,circ'_) => (outstream_,newcirc_))
[2] |- Signal :  t_
[1] |- ((Lo,Lo) ::  env_ |- AND#(Var 0) => (out_,circ'_) :  t_)
--------------------
|- ([(Lo,Lo), (Signal,Lo)], env_ |-
DELAY (Signal,AND#(Var0)) => (Signal ::  outstream_,newcirc_))
```

Note that the output from circuit is known even though the output from the AND has not been computed yet. We reduce premise 1 using `ReduceAND`, and the second premise

using `ReduceSignal`.

```
> apprl ReduceSignal;

***** Level 7 *****
|- ([(Signal,Lo)], env_ |-
DELAY (Lo,AND#(Var 0)) => (outstream_,newcirc_))
--------------------
|- ([(Lo,Lo), (Signal,Lo)], env_ |-
DELAY (Signal,AND#(Var 0)) => (Signal ::  outstream_,newcirc_))
```

We have now completed time zero, and can compute the next time step. Note that the description of the delay now has state `Lo`, which was the output from the AND gate at the previous time step. The second time step may be dealt with in exactly the same manner, resulting in the following:

```
***** Level 10 *****
|- ([], env_ |- DELAY (Lo,AND#(Var 0)) => (outstream_,newcirc_))
--------------------
|- ([(Lo,Lo), (Signal,Lo)], env_ |-
DELAY (Signal,AND#(Var 0)) => (Signal ::  Lo ::  outstream_,newcirc_))
```

The final application of `ReduceSeqNil` closes the input stream. Note that only at this point do we know what the final circuit looks like, in case we want to continue this simulation.

```
> apprl ReduceSeqNil;

***** Level 11 *****
--------------------
|- ([(Lo,Lo), (Signal,Lo)], env_ |-
DELAY (Signal,AND#(Var 0)) => ([Signal,Lo],DELAY (Lo,AND#(Var 0))))
```

As in the previous example, we could have done all of this with the application of a single tactic `safeOpSemAllTac' [ReduceAND]`. A list of derived rules may be passed into the tactic. This means that an AND gate, for example, is reduced using one derived rule application, rather a series of primitive rules. This facilitates faster, hierarchical simulation because a circuit does not need to be flattened out into individual gates to be simulated. It is also easier to pinpoint errors in a circuit when it is simulated hierarchically because boundaries of subcircuits are clearer when the subcomponents have not been flattened out. One needs to open up a subcircuit only when it is found to be in error.

## 4.2 Adder circuits

One of the strengths of the embedding approach used here is that we can manipulate circuit expressions just like any other term in the proof system. This allows us to write functions operating on and delivering circuits. In this subsection we will describe two implementations of a full adder, followed by an $N$ bit adder generator. Formal circuit generators were introduced by Brock *et al.* in [3, 4].

We will first show two implementations of a full adder. ADD1 is composed of two half adders in the following manner:

```
val OR#(e)   = IF e MATCHES (Lo,Lo) THEN Lo ELSE Hi;
val XOR#(e)  = IF e MATCHES (Hi,Lo)|(Lo,Hi) THEN Hi ELSE Lo;
val HA#(e)   = LET e IN (XOR#(Var 0), AND#(Var 0));
val ADD1#(e) = LET e (* ((x,y),c) *) IN
               LET HA#((Var 0)[1]) IN
               LET HA#(((Var 0)[1], (Var 1)[2])) IN
                      ((Var 0)[1],                          (* sum *)
                       OR#(((Var 0)[2], (Var 1)[2])));      (* carry *)
```

The outermost LET is necessary, in case the input expression contains Vars. It also avoids duplication of the input circuit by using a fan-out. For example, without this LET, the second half adder in ADD1#(Var 0) would incorrectly access the first half adder as input. We easily derive ReduceHA and ReduceADD1 using safeOpSemAllTac. In this example we can see quite clearly how we use proof system capabilities to structure our circuits at the object level. AND *etc.* are meta-level syntactic functions.

ADD2 is built directly from three AND gates, two OR gates and two XOR gates.

```
val ADD2#(e) = LET e IN (* ((x,y),c) *)
               LET AND#(((Var 0)[1][2], (Var 0)[2])) IN (* bc *)
               LET AND#(((Var 1)[1][1], (Var 1)[2])) IN (* ac *)
               LET AND#((Var 2)[1]) IN                   (* ab *)
               LET OR#((Var 2, OR#((Var 1,Var 0)))) IN
               LET XOR#((Var 4)[2], XOR#((Var 4)[1])) IN
                   (Var 0,Var 1);
```

Most of the complexity is due to the destruction and construction of tuple wires.

These two adders behave identically on fully defined inputs. However, ADD2 may be more defined than ADD1 on partially defined inputs, such as ((Hi,Signal), Hi). The former outputs (Signal, Hi) while the latter results in (Signal, Signal) for the (sum,carry) pair. For this input we cannot say anything about the sum, but we know that the carry must be Hi. In the case of ADD1 the pessimism is due to non-optimal use of the input; information is consumed piecewise by independent subcomponents. There is no one bit adder implementation whose outputs are more defined than those of ADD2 for partially defined values.

```
> applyTac (safeOpSemAllTac'[ReduceADD2]);

***** Level 4 *****
--------------------
|- ([((Hi,Signal),Hi), ((Signal,Hi),Hi), ((Lo,Signal),Lo),
((Hi,Hi),Signal)], env_ |- ADD2#(Var 0) =>
([(Signal,Hi), (Signal,Hi), (Signal,Lo), (Signal,Hi)],ADD2#(Var 0)))
```

We will now define a $N$ bit adder generating function which is parametrised on the full adder subcomponent.

```
(* onebitadder:  ((x,y),c) -> (s,c) *)
(* nadd:  (((x_{N+1},(..,x_0)),(y_{N+1},(..,y_0))),c_0) -> ((s_{N+1},(..,s_0)),c) *)
fun nadd onebitadder (S 0) x = onebitadder x |
    nadd onebitadder (S (S n)) x =
         LET x IN (* (((x_{N+1},(..,x_0)),(y_{N+1},(..,y_0))),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 *);
```

`nadd` 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. A $N+1$ 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 $((x_N, y_N), c_N)$. As with the `ADD1` circuit, virtually all of the complexity is due to the composition of intermediate wires. It is more complicated than in the 'paper version' of picoELLA due to the de Bruijn encoding of variables. The derived rule `ReduceNADDSSn`, dealing with $N+2$ word size, is quite involved. Premise one evaluates the input circuit; premise two the $N+1$ bit adder, and premise three the full adder. The remaining premises deal with the static semantics.

```
|- E t1_
|- o1_ :  t1_
|- E (Type m2)
|- E t1_3
|- o2_2 :  t1_3
[3] |- (CoTuple (o2_2,Cons (n3,m2)) ::  o1_ ::  env_ |-
add_ (((Var 1)[1][1][1], (Var 1)[1][2][1]), (Var 0)[2])
=> (CoTuple (Cons (n2,m1),Cons (n1,m)),
add_ (((Var 1)[1][1][1], (Var 1)[1][2][1]), (Var 0)[2])) :
TyTuple (Type m1,Type m))
[2] |- (o1_ ::  env_ |-
nadd add_ (S n) ((((Var 0)[1][1][2], (Var 0)[1][2][2]), (Var 0)[2]))
=> (CoTuple (o2_2,Cons (n3,m2)),
nadd add_ (S n) ((((Var 0)[1][1][2], (Var 0)[1][2][2]), (Var 0)[2]))) :
TyTuple (t1_3,Type m2))
[1] |- (env_ |- circ_ => (o1_,circ'_) :  t1_)
--------------------
|- (env_ |- nadd add_ (S (S n)) circ_ =>
(CoTuple (CoTuple (Cons (n2,m1),o2_2),Cons (n1,m)),
nadd add_ (S (S n)) circ'_) :  TyTuple (TyTuple (Type m1,t1_3),Type m))
```

We see that the output of the $N+1$ bit adder is a tuple `CoTuple (o2_2, Cons (n3,m2))`. Comparing this to the definition of `nadd` we see that `o2_2` represents the partial sum $(s_N, (.., s_0))$, and `Cons (n3,m2)` the carry $c_{N+1}$. Decoding the inputs of the final $N+$ 2nd bit adder `add_`, we see that its input carry `(Var 0)[2]` accesses the output carry

`Cons(n3,m2)` from the $N+1$ bit adder, as expected. The final result of the $N+2$ bit adder consists of (i) the concatenation of the sum bit of `add_` (`Cons(n2,m1)`) concatenated with the partial sum `o2_2`; and (ii) the carry bit `Cons(n1,m)` of `add_`. The derived rule `ReduceNADD1` just unfolds the `nadd` definition to evaluate the full adder. Note that the result circuit must be identical to the circuit we evaluate. This means that the adder is not allowed to have any state.

```
|- (env_ |- add_ circ_ => (out_,add_ circ'_) :  t_)
--------------------
|- (env_ |- nadd add_ 1 circ_ => (out_,nadd add_ 1 circ'_) :  t_)
```

A four bit adder has been simulated, with `ADD2` as the subcomponent. For example, binary $1010 + 1101 + 1 = 11000$, that is, a sum of 1000 and a high carry:

```
> applyTac (safeOpSemAllTac'[ReduceNADD4bit']);

***** Level 4 *****
--------------------
|- ([(((Hi,(Lo,(Hi,Lo))), (Hi,(Hi,(Lo,Hi)))), Hi)], env_ |-
nadd (fn e => ADD2#(e)) 4 (Var 0)
=> ([((Hi,(Lo,(Lo,Lo))), Hi)], nadd (fn e => ADD2#(e)) 4 (Var 0)))
```

Note that `ADD2` is a meta-level syntactic function, and must therefore be converted into an object level function, using `(fn e => ADD2#(e))`.

In this section we see most clearly the increased power of our methodology over symbolic simulation as it has been used by Bryant [5] for example. When we remarked that MOSSYM does not allow abstraction over circuits what we intended to convey was that it does not allow the simulation of an $N$ bit adder. Our approach allows more than this; we can even simulate an $N$ bit adder built using any one bit adder `onebitadder`. As long as we know that the subcircuit `onebitadder` behaves like a one bit adder, we can simulate any circuit in which it is used. We can simulate a circuit containing abstract hardware, as long as we know what the behaviour of the subcomponent is. Let us consider an ALU, containing an $N$ bit adder. The $N$ bit adder specification will usually be stated at a higher level of abstraction, using natural numbers. The specification for the sum could be $bitsof(natof\ x + natof\ y)\ mod\ 2^N$. *natof* is a data abstraction function, and *bitsof* its inverse. When we simulate the ALU, and arrive at the $N$ bit adder subcomponent, it makes sense to use the specification rather than the implementation. (This assumes we have shown that the implementation specifies the specification.) Rather than simulating the basic gates the adder is composed of, we compute the natural number expressions stating the values *sum* and *carry* have. This is not only faster, also conceptually clearer.

## 4.3 Two Parity Checkers

Boulton *et al.* illustrate their approach to the verification of ELLA designs with a parity checker [2]. It consists of two multiplexors, two delays and a NOT gate. `PCHECK2` below describes the same circuit as `PARITY_IMP` in the cited paper.

```
val NOT_g#(e) = IF e MATCHES Hi THEN Lo ELSE Hi;
val MUX#(e,b1,b2) = IF e MATCHES Hi THEN b1 ELSE b2;
val REG#(c,e) = DELAY (c,e);
val PCHECK2#(s1,s2,e) = LET e IN
                        LET INIT Signal REC
                        (* Use a LET to avoid duplication of register *)
                        LET REG# (s1,Var 0) IN
                            MUX# (REG# (s2, Hi),
                                  MUX# (Var 2, NOT_g#(Var 0), Var 0),
                                  Hi) IN
                            Var 0;
```

We use `NOT_g` because `NOT` is the truth value not operator in LAMBDA. It is worth noting that the state of the parity checker is explicit in the abbreviation. The reason for this is so that the abbreviation may be used in all possible states, and not just the initial state.

```
[5] |- ceq Signal (Cons (n1,1)) == false
|- Cons (n1,1) == (case match Hi (Cons (b,1)) of        (* Outer MUX *)
uu => bottom o1_ | tt => o1_ | ff => Hi)
|- o1_ == (case match Hi (Cons (n,1)) of                (* Inner MUX *)
uu => bottom o1_1 | tt => o1_1 | ff => Cons (a,1))
|- o1_1 == (case match Hi (Cons (a,1)) of               (* NOT *)
uu => bottom Lo | tt => Lo | ff => Hi)
|- (env_ |- circ_ => (Cons (n,1),h) :  Type 1)
--------------------
|- (env_ |- PCHECK2#(Cons (a,1),Cons (b,1),circ_) =>
(Cons (n1,1),PCHECK2#(Cons (n1,1),Hi,h)) :  Type 1)
```

The derived rule `ReducePCHECK2` contains some points of interest. First note that only the two multiplexors and the NOT gate are present as subgoals; both delays have disappeared. As described in [2], the rôle of the innermost register is to output `Lo` at time zero, and `Hi` ever after. This is evident from the conclusion of the rule below, where the state `s2` is always `Hi` after an evaluation. Also note that the output `Cons(n1,1)` is duplicated in the first register, so that it can be used in the next time step, using the feedback. At time zero, the values in the registers are both `Lo`. In fact, the value in the first delay at time zero is irrelevant:

```
|- ([Cons (y,1)], env_ |- PCHECK2#(Cons (x,1),Lo,Var 0) =>
([Hi],PCHECK2#(Hi,Hi,Var 0)))
```

This derivation uses an arbitrary input `Cons(y,1)` and state in the first delay `Cons(x,1)`. The only constraint on these *don't care* values is that they must have the right type. Note that their possible value includes the undefined or *don't know* value. This simulation shows that the state of the new circuit is fully defined no matter what the input at time zero is. In other words, the value of the input at time zero is ignored. This parity checker outputs `Hi` at time $t$ if there have been an even number of `His` in the input stream from time *one* to time $t$ inclusive.

An alternative parity checker is listed below.

```
val PCHECK1#(s,e) = LET e IN
                    LET INIT Signal REC
                    REG# (s, XOR# (Var 0, Var 1)) IN
                         Var 0;
```

The initial state must be `Hi`. `PCHECK1` outputs `Hi` at time $t+1$ if there have been an even number of `His` in the input stream from time *zero* to time $t$. The output at time zero is `Hi`.

```
> applyTac (safeOpSemAllTac'[ReducePCHECK1]);

***** Level 4 *****
--------------------
|- ([Hi,Lo,Hi,Hi,Lo,Lo], env_ |- PCHECK1#(Hi,Var 0) =>
([Hi,Lo,Lo,Hi,Lo,Lo],PCHECK1#(Lo,Var 0)))
```

Using conventional verification techniques we proved that the `PCHECK1` circuit does indeed count the number of `His` in the input stream.

```
(* Number of v's in the input stream from time 0 up to time t.  *)
fun noof v input 0 = 0 |
    noof v input (S t) = if input t = v then (noof v input t) + 1
                                        else (noof v input t);
fun even n = n mod 2 = 0;
fun absinv true = Hi | absinv false = Lo;
fun state x y = absinv (even (noof Hi x y));
```

`noof` counts the number of `v`s in the input stream, `even` returns true if there have been an even number of them, and `absinv` is the inverse data abstraction function, mapping booleans to constants. `state` combines these three functions into one, to make the result more readable.

```
|- forall t,l,e,input.  input t == Hi \/ input t == Lo ->>
Reduce l (PCHECK1#(state input t,e t)) ==
(state input t, PCHECK1#(state input (S t),e (S t)))
```

In other words, assuming the input is either `Hi` or `Lo` at every time step, the output at time $t$ consists of two parts. The first value is `Hi` is there have been an even number of `His` in the input stream. The second part states that the state of the new circuit is given by the `state` function at time $t+1$. As we discussed at the end of the previous subsection, we can use this specification instead of using the circuit in simulations.

Although it was not shown in the last two examples, the semantics computes the least fixed point of a `LET REC`. An iterative method is used, and the number of iterations may vary to reach the fixed point. In the case of delayed feedbacks, however, it takes at most one iteration. If the output is not undefined exactly one iteration is needed. In the derived rules for `PCHECK1` and `PCHECK2` the assumption was made that no undefined values were input to the circuit. (Premise `[5]` of rule `ReducePCHECK2` states this. The assumption is more explicit in the theorem above.) It follows from this assumption that only defined values are output and hence only one iteration is needed. The current tactics do not attempt to deal with recursion.

# 5 Conclusions

Simulation and verification are usually described as alternative, incompatible approaches. This paper shows that by suitably embedding the operational semantics of a HDL in an appropriate proof tool we are able to integrate simulation and verification within the same framework. We believe the approach taken here is applicable to any HDL. The choice of picoELLA and LAMBDA is not crucial to the discussion.

The strength of our approach is the ability to specify, implement, simulate and reason about a circuit within a single framework. At any stage in this process we may use a conventional HDL notation, the logic supported by the proof system, or a mix of the two. Although the specification will often be expressed using logic, an algorithmic specification, *i.e.* as a high level HDL program, may be useful. An algorithmic specification can also be used to give a more operational intuition by executing it. Logic specifications (and implementations) cannot be animated easily. The common relational hardware description style, which uses existential quantification for hidden wires, is an example. Moreover, structure and behaviour are not properly separated; the form of the behavioural description is used to indicate the intended structure of the circuit. Our approach strictly separates structure and behaviour [9]. Behaviour is given to a purely structural term through a formal embedded semantics, and properties of circuits are derived using this semantics. The ability to reason about and manipulate structural expressions *per se* is very useful. It facilitates interfacing with conventional design tools because they use the same notation. For example, circuits designed using a proof system may be exported directly to layout generators. Alternatively, hardware output by unverified hardware synthesis tools can be validated using the proof system. Hardware may also be synthesised formally using hardware generators such as `nadd` in section 4.2, first introduced by Brock and Hunt in [3]. Formal synthesis [12], and refinement based approaches [7] fit well into our framework. DIALOG is a graphical synthesis package integrated with the LAMBDA proof system. Using DIALOG, it would be possible to synthesise formally verified HDL descriptions without the need to explicitly use the underlying proof system. This could be seen as a HDL interface to the proof system; the user does not need to interact with the underlying proof system. We can also treat circuit optimisations formally. If two structural terms have equivalent behaviours, they may be substituted for one another in any context. A given circuit could be optimised by (possibly context dependent) rewriting, which is certainly possible in LAMBDA. Finally, we can use the embedded semantics to simulate the structural terms. Both data and circuit descriptions may be meta-variables, enabling powerful symbolic simulation. Partial implementations may be simulated by using the specifications of the missing components. In our opinion the main advantage of this approach is the possibility of using a conventional HDL in more formal setting. This bridges the gap between hardware designers and verification engineers.

Future work includes the optimisation of tactics. Tactics must be made to deal with recursion automatically if possible. It will also be helpful to make the use of the system more user-friendly by providing a menu-based X window interface using LAMBDA's built-in browser. Finally, picoELLA was designed to exhibit the ideas outlined here and in [9]. A larger, more readable, subset of ELLA must be used for practical applications.
A longer version of this paper is available as LFCS report number ECS-LFCS-92-231.

# References


[1] H Barringer, G Gough, and B Monahan. Operational semantics for hardware design languages. In P Prinetto and P Camurati, editors, *Advanced Research Workshop on Correct Hardware Design Methodologies*, pages 313–334. North Holland, June 1991.

[2] 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.

[3] B Brock and W Hunt, Jr. The formalization of a simple hardware description language. In L Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 778–792, November 1989. Elsevier Science Publishers.

[4] B Brock, W Hunt, Jr, and W Young. Introduction to a formally defined hardware description language. In V Stavridou, T Melham, and R Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 3–35. North Holland, June 1992.

[5] R Bryant and C 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.

[6] Computer General Electronic Design, The New Church, Henry St, Bath BA1 1JR, England. *The ELLA Language Reference Manual*, issue 4.0, 1990.

[7] M Fourman and E Mayger. Formally based system design – interactive hardware scheduling. In G Musgrave and U Lauther, editors, *Int'l Conf. on VLSI*, 1989.

[8] M Francis, S Finn, and E Mayger. *Reference Manual for the Lambda System.* Abstract Hardware Limited, version 3.2, November 1990.

[9] K G W Goossens. Embedding hardware design and description languages in proof systems. Forthcoming PhD thesis. University of Edinburgh.

[10] K G W Goossens. Semantics for picoELLA. Manuscript, June 1990.

[11] 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. North Holland, June 1991.

[12] F Hanna, M Longley, and N Daeche. Formal synthesis of digital systems. In L Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, pages 532–548, November 1989. Elsevier Science Publishers.

[13] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard VHDL Language Reference Manual*, IEEE std 1076-1987, 1988.

[14] T Melham. Using recursive types to reason about hardware in higher order logic. Technical Report 135, University of Cambridge Computer Laboratory, May 1988.

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

[16] V Stavridou, J Goguen, S Elker, and S 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. North Holland, June 1991.

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

[18] P Wilsey, T McBrayer, and D Sims. Towards a formal model of VLSI systems compatible with VHDL. In A Halaas and P Denyer, editors, *VLSI '91*, August 1991.