The HOL Verification of ELLA Designs[1]

Richard Boulton, Mike Gordon, John Herbert, John Van Tassel

University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
United Kingdom

Abstract:
HOL is a public domain system for generating proofs in higher order predicate calculus. It has been in experimental and commercial use in several countries for a number of years.

ELLA[2] is a hardware design language developed at the Royal Signals and Radar Establishment (RSRE) and marketed by Computer General Electronic Design. It supports simulation models at a variety of different abstraction levels.

A preliminary methodology for reasoning about ELLA designs using HOL is described. Our approach is to *semantically embed* a subset of the ELLA language in higher order logic, and then to make this embedding convenient to use with parsers and pretty-printers. There are a number of semantic issues that may affect the ease of verification. We discuss some of these briefly. We also give a simple example to illustrate the methodology.

---

[1] Presented at the International Workshop on Formal Methods in VLSI Design, Miami, January 1991.

1 Introduction

There is currently a gap between the methods used to specify hardware in simulation oriented CAD systems and the methods used for formal verification. In the former, a hardware description language (HDL) like ELLA [10] or VHDL [8] is used, in the latter special purpose formal systems like HOL [5], Boyer-Moore logic [2] or temporal logic [7] are employed.

The research described here, which has been in progress for about a year, aims to build a bridge between these two methods by providing theorem proving tools for reasoning about designs expressed in the ELLA HDL. It is hoped thereby to make formal specification and proof available within a conventional design process. Conversely, it is hoped to provide access to the engineering tools of ELLA (e.g. the simulator and layout generators) in a theorem proving environment.

The approach taken, which is illustrated by the example in Section 4, is to *semantically embed* a subset of ELLA in higher order logic, the logic of the HOL theorem proving system. This approach has already been used for programming languages [6]. Parsers and pretty-printers are provided that enable texts in ELLA syntax to be input and output from HOL. An ELLA text parses to the term of higher order logic that represents its semantics. Behavioural properties of this semantic representation can then be deduced, using the standard laws of logic. Since logical formulae may be unfamiliar to designers, those formulae that represent the behaviour of ELLA constructs can, optionally, be printed out in ELLA notation. This is illustrated in the example in Section 4.

We give below overviews of ELLA, higher order logic, and the HOL system. We hope that these will make this paper accessible to members of the formal methods community and hardware designers, where the former may have just a basic knowledge of hardware description languages, the latter, some familiarity with formal logic.

1.1 Examples of ELLA and the corresponding VHDL

In this section, we explain the ELLA constructs used in this paper. For readers familiar with VHDL we give here both the ELLA and VHDL specifications of the parity checker example used in Section 4.

VHDL[3] is an event-driven simulation language that has been accepted as an IEEE standard (Std 1076–1987). Since the language relies heavily on state transitions, it differs markedly in philosophy from ELLA, which is primarily a functional language.

The first design unit we describe models the unit delay element. In VHDL this is specified by:

[3] Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language.

```
ENTITY reg IS
    PORT (inp: bit; outp: OUT bit);
END reg;;

ARCHITECTURE data_flow OF reg IS
BEGIN
    outp <= TRANSPORT inp AFTER 1 NS;
END data_flow;
```

We have chosen nanoseconds as the unit of time.

In ELLA the unit delay element is specified by:

```
FN REG = (bit) -> bit: DELAY(lo, 1).
```

In this particular implementation the output is initially `lo`.

The type `bit` is built-in in VHDL, but in ELLA it has to be defined. We define it to have two possible values, `hi` and `lo`.

```
TYPE bit = NEW (hi | lo).
```

Our second example is the implementation specification of the parity checker device. A diagram of this device is given in Section 4, together with an explanation of its operation. In VHDL the specification is:

```
ENTITY parity_check is
    PORT (inp: bit; outp: INOUT bit);
END parity_check;

ARCHITECTURE gate_level OF parity_check IS
    COMPONENT one PORT (outp: OUT bit); END COMPONENT;
    FOR on1: one USE ENTITY one (data_flow);

    COMPONENT inverter
        PORT (inp: bit; outp: OUT bit);
    END COMPONENT;
    FOR inv: inverter USE ENTITY inverter (data_flow);

    COMPONENT mux
        PORT (selector,a,b: bit; selected: OUT bit);
    END COMPONENT;
    FOR mu1,mu2: mux USE ENTITY mux (data_flow);

    COMPONENT reg PORT (inp: bit; outp: OUT bit); END COMPONENT;
    FOR re1,re2: reg USE ENTITY reg (data_flow);

    SIGNAL l1,l2,l3,l4,l5: bit;
BEGIN
    on1 : one PORT MAP (l5);
    inv : inverter PORT MAP (l2,l1);
    mu1 : mux PORT MAP (inp,l1,l2,l3);
    re1 : reg PORT MAP (outp,l2);
    re2 : reg PORT MAP (l5,l4);
    mu2 : mux PORT MAP (l4,l3,l5,outp);
END gate_level;
```

In ELLA the parity checker is specified by the following text, which is an example of a *series*:

```
FN PARITY_IMP = (bit: in) -> bit:
 BEGIN
    MAKE INV: l1,
         MUX: l3 out,
         REG: l2 l4.
    JOIN (in, l1, l2) -> l3,
         hi           -> l4,
         (l4, l3, hi) -> out,
         out          -> l2,
         l2           -> l1.
 OUTPUT out
 END.
```

The various `COMPONENT` and `SIGNAL` declarations in the VHDL correspond approximately to the `MAKE` statement in the ELLA model while the component instantiations between `BEGIN` and `END` are roughly analogous to the `JOIN` construct in the ELLA specification. They differ in that in addition to declaring components and signals, the ELLA `MAKE` statement specifies to which component outputs the signals are connected. In the VHDL this task is performed in the `BEGIN/END` section.

The element `one` in the VHDL specification is replaced by a constant in the ELLA. The devices `INV` and `MUX` used in the ELLA specification are implemented using the ELLA `CASE` statement. The elaboration of an ELLA `CASE` statement most closely resembles that of a VHDL conditional signal assignment statement, but due to fundamental differences between the two languages, any real comparison is not meaningful. We therefore describe it by example, using the definition of `MUX` (a multiplexer).

```
FN MUX = (bit: cntl in1 in2) -> bit:
 CASE cntl
 OF hi: in1,
    lo: in2
 ESAC.
```

The `CASE` statement uses a control input (in this case `cntl`) to select from one of a number of other inputs. It does this by comparing the control input to patterns. In the example, the patterns are simply the constant values `hi` and `lo`, but in general they can be much more sophisticated. For example the patterns could be a set of constants. If the first pattern matches the control input the output is 'connected' to the corresponding ELLA unit. In our example, the unit for the first pattern (`hi`) is the input signal `in1`, but in general it could be almost any hardware description involving `in1`, `in2` and even the control input `cntl`.

The 'value' compared with the pattern can also be a complex expression involving the inputs, rather than just one of the inputs. This illustrates the functional nature of the ELLA language in that, for the most part, anywhere a signal name can be used, an expression can be used instead.

Each of the patterns are tested in turn until one is found which matches. If no match is found a default value is used for the output, or if, as is the case in the example, no default is specified, the result is undefined. The presence of such undefined values in ELLA makes the pattern matching more complex than we have described it here. This is discussed in more detail in Section 3.2.

1.2 The HOL System

HOL is an interactive proof development system for higher order logic. A user interacts with the system via ML[4]. This is a strongly typed higher order functional programming language, which supports the definition of recursive concrete datatypes. There are predefined datatypes in ML representing the terms and theorems of higher order logic. The only way to create theorem values is to apply a sequence of inference rules to a value representing an axiom, or to an already existing theorem (theorems are printed with a preceding turnstile symbol $\vdash$). This approach, which

[4] ML is an acronym for "metalanguage".

is due to Milner [4, 9], ensures that a theorem can only be created if it has a formal proof.

The formulation of higher order logic supported by the HOL system is a version of simple type theory [3] with a restricted form of polymorphism (i.e. type variables). Terms of this logic include function-denoting λ-abstractions; the term $\lambda x.\ t[x]$ denotes the function $a \mapsto t[a]$. The type discipline of higher order logic is similar to the type system of the metalanguage ML. The type *bool* is the type of the truth values T and F. If σ_1 and σ_2 are types then $\sigma_1 \rightarrow \sigma_2$ is the type of functions from σ_1 to σ_2 and $\sigma_1 \times \sigma_2$ is the Cartesian product of σ_1 and σ_2. The propositional connective $\neg$ (not) is a constant in higher order logic with type $bool \rightarrow bool$; $\wedge$ (and), $\vee$ (or) and $\Longrightarrow$ (implies) are constants with type $bool \times bool \rightarrow bool$. The quantifiers $\forall$ and $\exists$ are also constants; they have the polymorphic type $(\alpha \rightarrow bool) \rightarrow bool$ (where α is a type variable). The idea is that $\forall P$ is true if P maps every argument to T and $\exists P$ is true if P maps some argument to T. The notations $\forall x.\ t$ and $\exists x.\ t$ parse to $\forall(\lambda x.\ t)$ and $\exists(\lambda x.\ t)$, respectively. An important logical constant is ε, which has type $(\alpha \rightarrow bool) \rightarrow \alpha$. This embodies the Axiom of Choice: the term εP denotes some x such that $P(x)$ is true (it denotes an arbitrary value of type α if no such x exists). The notation $\varepsilon x.\ t$ parses to $\varepsilon(\lambda x.\ t)$.

The HOL system comes with the logical constants built in, together with various other constants, which support useful notations. For example, the conditional notation $(p \Rightarrow t_1 \mid t_2)$ stands for $\mathsf{Cond}\ p\ t_1\ t_2$, where Cond is defined by:

$$\mathsf{Cond}\ p\ t_1\ t_2 \;=\; \varepsilon x.\ (p \Longrightarrow (x = t_1)) \wedge (\neg p \Longrightarrow (x = t_2))$$

Users may define new types and constants within the logic using various definitional mechanisms. The HOL-ELLA system generates definitions of HOL types from ELLA type declarations and HOL constants from ELLA macro and function declarations. Examples are given later.

2 The HOL-ELLA System

ELLA is embedded in higher order logic using a parser and pretty-printers generated with tools from the HOL system library. ELLA text can be manually entered in a HOL session or can be read from a file. The text is parsed to an ML datastructure, which is then manipulated so as to make translation to higher order logic straightforward. If a semantic error is found during translation, or some part of ELLA not in the chosen subset is encountered, the offending piece of ELLA text is pretty-printed together with an appropriate message. It is also possible to pretty-print the HOL term so that it appears in a form similar to the ELLA text it represents. The structure of the HOL-ELLA system is illustrated in Figure 1.

The parser used is an ML program automatically generated from a specification of the ELLA syntax. The generator is a library in the HOL system, and performs much the same function as the `yacc/lex` combination in Unix[5]. It takes as input a

[5]Unix is a registered trademark of AT&T

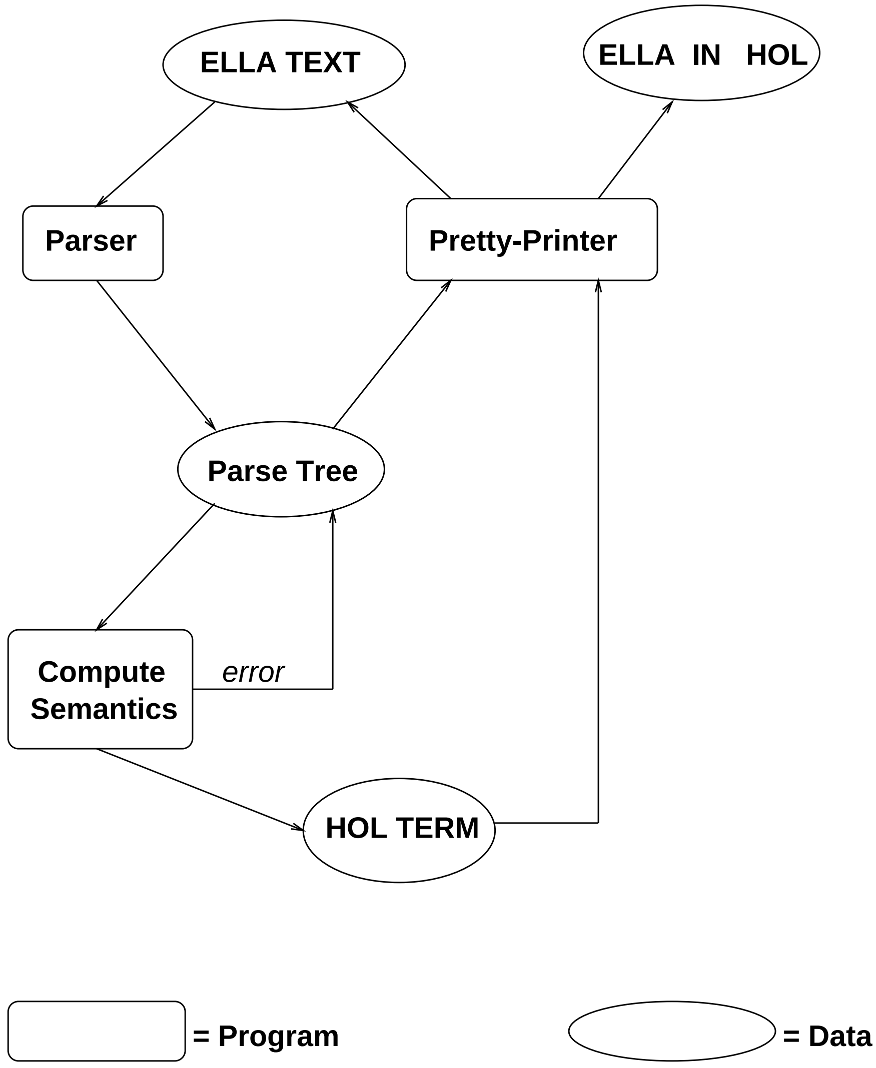


Figure 1: The HOL-ELLA System

specification which describes the context-free syntax of the language. The notation is a modified form of BNF resembling Prolog's definite-clause grammars. Parse-trees (or any other data object) may be created during the parse through action symbols embedded in the grammar.

The pretty-printer generator is inspired by the Pretty-Printing Meta-Language for the CENTAUR system [1]. As indicated previously, there are two pretty-printers in the system. One is for parse-trees and the other is for HOL terms. The parser and pretty-printer for parse-trees work for the whole of the ELLA language. The pretty-printer for HOL terms only deals with the semantically embedded subset of ELLA. It produces a mixture of higher order logic and ELLA. Semantic constants are printed as the ELLA construct they represent.

3 Embedding ELLA in HOL

The subset of ELLA we have embedded in HOL is quite large. We give here the semantic representations of only those features used in the example in Section 4. Our intention has been to keep the translation process as simple as possible, since it is outside the scope of formal verification. By 'simple', we mean that the HOL representation of a piece of ELLA is close to the parse-tree for that ELLA text.

We have taken steps to minimize the risk of error in the parser and pretty-printers. For example, the official ELLA grammar was used almost without modification to generate them. We could have performed simplification during the translation process, using an ML program. This would have greatly eased the work necessary for supporting proof within the logic. Instead we perform the simplification within the logic by formal proof, after the translation has taken place. We believe this to be the safest approach possible without formally verifying the translation programs. In addition, it is not clear to us how to specify the correctness of the translation.

When using the system, the ELLA text entered will (when the pretty-printer is active) be reproduced almost identically in the output of the system. The output is a list of theorems representing the results of definitions generated by the translator. These theorems can be simplified by logical inference so that the ELLA constructs disappear and are replaced by natural representations of the specified behaviour in logic. We hope that this approach will give the user confidence in the validity of the verification process all the way from the ELLA text, even though the translation from ELLA to HOL is not formally proved correct.

3.1 ELLA constants

Constants have two functions in ELLA. They are primarily used as *choosers* in the `CASE` statement. In this capacity, they behave as patterns. For example, the constant

```
hi | lo
```

matches a value if it is either `hi` or `lo`. The other use for constants is as initial values for devices which have state. In this capacity, ELLA constants are constants in the conventional sense.

The syntax of ELLA has a single syntactic category for constants, but only certain parts of it make sense as choosers. An overlapping, but different subset of the syntax of constants makes sense as initial values. To keep the translation from ELLA to HOL context-free, constants are translated in the same way wherever they appear. They are modelled as predicates. This fits in with their use as choosers. For use as initial values, Hilbert's ε-operator [5] is applied to the predicate. When the predicate specifies a unique value (as is required for initial values), the result is that unique value. In any other case, it is not possible to eliminate the ε-operator to yield a specific value, and so proofs depending on the identity of the constant will be inhibited.

We now look at the semantic representations in detail. The simplest form of ELLA constant is a single constant of some defined datatype. This is represented by a predicate which is true if and only if its argument is the constant value in question. For example, the representation of the ELLA constant `hi` in higher order logic is given by:

```
⟦hi⟧ = "CONST hi"
```

The notation ⟦...⟧ denotes the meaning of the ELLA text enclosed within the emphasised brackets ⟦ and ⟧. The meaning is a higher order logic *term*. HOL terms are enclosed in double quotation marks.

The semantic HOL constant `CONST` is defined by:

```
CONST (c:*) = (LIFT_apply1 (λx. x = c))
```

The notation `(c:*)` labels the argument of the function `CONST` as `c` and specifies its type to be `*` (a polymorphic type). The result of applying `CONST` is a *lifted* predicate. We explain lifting and justify its use in Section 3.2.

3.2 Lifting

ELLA features an unspecified value denoted by `?`. This value may be generated by arithmetic operations producing a result out of range, by `CASE` statements with choosers which are not exhaustive, or it may even be generated explicitly in the ELLA text (source code). In each case `?` represents one of the values of the appropriate type, but it is not known which.

An unknown value of some type can readily be represented in higher order logic either as a partial specification, or by use of Hilbert's ε-operator. However, there is one place in the semantics of ELLA where `?` behaves not as an unknown value of the type, but as an *additional* value of the type. Use of partial specification or Hilbert's ε-operator cannot represent such an additional value. We were therefore left with a choice between making a small modification to the semantics of ELLA or adding an extra value to every HOL type used to represent an ELLA type. We chose to do the latter, but further work is required to convince ourselves that the extra complexity introduced does not inhibit the verification of real examples.

We now give an example of the feature of ELLA which considers `?` to be an additional value. Suppose we have defined a type `bit` in ELLA:

```
TYPE bit = NEW (hi | lo)
```

This defines `bit` to be an enumerated type with two elements, `hi` and `lo`. In the semantics of ELLA the constant

```
hi | lo
```

matches `hi` or `lo`, but *not* the unspecified value. However, the constant

```
bit
```

matches any value of type `bit`, i.e. `hi` or `lo` *or* the unspecified value. Unless we add an extra element to the representation of the type `bit`, we cannot distinguish between these two constants in the semantics, which clearly have different meanings in ELLA.

Rather than adding an extra element to every HOL type representing an ELLA type, we have chosen the more uniform approach of *lifting*. We represent a value of type `bit` in ELLA with a value of type `(bit)lifted` in HOL. A value of type `(bit)lifted` may be the type constructor function `LIFT` applied to some value of HOL type `bit`, or it may be the value `UU`, representing the unspecified value of ELLA type `bit`.

In addition to the constants `LIFT` and `UU`, our theory of lifting has other useful constants. The two most important of these constants are `UNLIFT` and `LIFT_apply1`. `UNLIFT` converts a value of type `(*)lifted` (where `*` can be any type) to a value of type `*` if and only if its argument is of the form `LIFT x` for some term `x`. The meaning of the term `UNLIFT UU` is unspecified. The definition of `UNLIFT` is:

```
UNLIFT (LIFT (x:*)) = x
```

The HOL constant `LIFT_apply1` can be used to lift a function. The type of `LIFT_apply1` is given by:

```
LIFT_apply1 : (*→**) → ((*)lifted → (**)lifted)
```

where `*` and `**` are type variables and the definition of `LIFT_apply1` is:

```
LIFT_apply1 (f:*→**) xl =
   ((xl = UU) ⇒ UU | (LIFT (f (UNLIFT xl))))
```

It can be seen that the first argument `f` is a function not involving lifted values. Applying `LIFT_apply1` to such an argument yields a new function which returns `UU` when given `UU` as argument, or else it applies `f` 'under' the lifting.

3.3 Type definitions

An ELLA type declaration such as

```
TYPE bit = NEW (hi | lo)
```

is passed to the HOL system's type definition package. The result is the introduction of a new HOL type `bit` and a theorem which characterizes the new type. In the example above, the theorem is:

```
⊢ ∀e0 e1. ∃! fn. (fn hi = e0) ∧ (fn lo = e1)
```

The system then proves four useful properties of the new type: structural induction, constructors are injective (one-to-one), elements are distinct, and cases. For the example these are respectively:

```
⊢ ∀P. P hi ∧ P lo ⟹ (∀b. P b)
⊢ T
⊢ ¬(hi = lo)
⊢ ∀b. (b = hi) ∨ (b = lo)
```

In this case the second theorem just states that true (`T`) is true. None of the constructors are functions, so there is nothing that can be said about their injectivity. For more complex types the corresponding theorem is not vacuous.

The system also generates a theorem for each constructor. Each theorem defines an abbreviation for the *lift* of the constructor (see Section 3.2). In the case when the constructor is just an element (as for all the constructors in the example) the definition is trivial.

```
⊢ LIFT_hi = LIFT hi
⊢ LIFT_lo = LIFT lo
```

3.4 The ELLA `CASE` statement

The ELLA `CASE` statement behaves as a generalised multiplexer and has the following semantics:

$$
\begin{array}{l}
[\![\texttt{CASE}\ \mathit{unit} \\
\ \texttt{OF}\ \mathit{choices}_1\ \texttt{ELSEOF}\ \mathit{choices}_2\ \ldots\ \texttt{ELSEOF}\ \mathit{choices}_n \\
\ \texttt{ELSE}\ \mathit{else} \\
\ \texttt{ESAC}]\!]\ \texttt{=} \\
\texttt{"CASE}\ [\![\mathit{unit}]\!]\ \texttt{[}[\![\mathit{choices}_1]\!]\texttt{;}\ \ldots\ \texttt{;}[\![\mathit{choices}_n]\!]\texttt{]}\ [\![\mathit{else}]\!]\texttt{"}
\end{array}
$$

where the HOL constant `CASE` is defined by:

```
(CASE (unit:time→*) [] (else:time→**) t = (else t)) ∧
(CASE unit (CONS f l) else t = (f unit (CASE unit l else t) t))
```

The first argument to `CASE` is an ELLA unit. It is the input to the `CASE` statement. The second argument is a list. There is one element in this list for each `OF` or `ELSEOF` in the ELLA text. The third argument is the default output unit. The HOL constant `CASE` is defined recursively on the second argument. Note that the input and output to the `CASE` statement are signals, i.e. functions of time.

The semantics of the `CASE` statement require that $choices_1$ are tested first for a match with the input, and only if there is no match should the other choices be considered. If no choices provide a match, the `ELSE` clause is used as the output. The definition of `CASE` expresses this semantics by making the tail of the list of choices a default to be used if the head does not provide a match, or in the case of the list being empty, the `ELSE` clause is immediately used as the default. Here is an example from Section 4:

```
CASE in
OF lo: hi,
   hi: lo
ESAC.
```

The HOL representation for this ELLA text is:

```
CASE
in
[OF[CONST lo,SIGNAL LIFT_hi;CONST hi,SIGNAL LIFT_lo]]
(εx'. T)
```

Here, the second argument to `CASE` is a list of one element. The value of that element and the value of the third argument are explained shortly. Applying the HOL term above to a variable `t` representing a time, and expanding once with the definition of `CASE` yields:

```
OF
[CONST lo,SIGNAL LIFT_hi;CONST hi,SIGNAL LIFT_lo]
in
(CASE in [] (εx'. T) t)
t
```

Expanding once more gives:

```
OF
[CONST lo,SIGNAL LIFT_hi;CONST hi,SIGNAL LIFT_lo]
in
((εx'. T) t)
t
```

For the first expansion, the second clause of the definition of `CASE` was used. For the second expansion the list argument was an empty list and so the first clause of the definition was used. It can now be seen that the value of the HOL term depends on the definition of the semantic constant `OF` which we now describe.

A choice of the form:

```
chooser₁: unit₁, ... ,choosern: unitn
```

where each *chooser* is used as a pattern, and each *unit* as an output value, is represented in HOL by:

```
⟦chooser₁: unit₁, ... ,choosern: unitn⟧ =
"OF [⟦chooser₁⟧,⟦unit₁⟧; ... ;⟦choosern⟧,⟦unitn⟧]"
```

This has already been illustrated by our example. The HOL constant `OF` is defined by:

```
(OF [] (in:time→*) (else:**) t = else) ∧
(OF (CONS pair rest) in else t =
    ((UNLIFT ((FST pair) (in t))) ⇒
        ((SND pair) t) | (OF rest in else t)))
```

`OF` is defined recursively in a similar way to `CASE`. If all the choosers have been tried without success, the default is used. The default is the remainder of the `CASE` statement (see above). If the list of chooser/unit pairs is not exhausted, the first pair is tested. The chooser is applied to the input value. If the result is not defined, then the result of the `CASE` statement will not be defined. If the result is true, the corresponding unit becomes the result of the statement, otherwise the next chooser is tested by a recursive call. So, expanding the example once more, but this time with the definition of `OF`, we get:

```
(UNLIFT ((FST (CONST lo,SIGNAL LIFT_hi)) (in t))) ⇒
((SND (CONST lo,SIGNAL LIFT_hi)) t) |
(OF [CONST hi,SIGNAL LIFT_lo] in ((εx'. T) t) t)
```

Using the definitions of `FST` and `SND` this simplifies to:

```
(UNLIFT (CONST lo (in t))) ⇒
(SIGNAL LIFT_hi t) |
(OF [CONST hi,SIGNAL LIFT_lo] in ((εx'. T) t) t)
```

We assume that the choosers of each choice are distinct, as required by the ELLA language, though our translator does not currently check this. It does not however seem an unreasonable assumption, since any ELLA design to be verified could have previously been passed through the compiler of the ELLA system. In the event that the choosers are not distinct, our semantics uses the unit associated with the first chooser (in textual order) to match.

As in our example, the `ELSE` clause may be omitted. In such an event, the HOL term:

```
"εx'. T"
```

is used as the third argument to `CASE`. This term represents an unspecified value of the output type (see Section 3.2).

3.5 ELLA series

An ELLA series has the form:

```
BEGIN step₁. ... stepₙ. OUTPUT unit END
```

We represent this by considering each step to be a Boolean-valued expression. We also introduce a variable `OUTPUT` which we equate to the value of the output unit. This equation and the representation of the steps are conjoined to yield a Boolean value. Having existentially quantified over the variables local to the steps, we select the value of `OUTPUT` that makes the expression true. We do this using Hilbert's ε-operator. Provided the series is well-formed, the value selected is unique. In HOL, the semantic representation is:

```
⟦BEGIN step₁. ... stepₙ. OUTPUT unit END⟧ =
"εOUTPUT. ∃local variables.
    SERIES [⟦step₁⟧; ... ;⟦stepₙ⟧] OUTPUT ⟦unit⟧"
```

where `SERIES` is defined recursively by:

```
(SERIES [] output (unit:*) = (output = unit)) ∧
(SERIES (CONS x l) output unit = (x ∧ (SERIES l output unit)))
```

For each internal 'line' of the series, we have two local variables. One represents the value on the line. The other is used as a link between `MAKE` and `JOIN` statements (see below). For typing reasons, it is not possible to use the value on the line as a link.

There are two kinds of step used in the example in Section 4, and there are others not documented here. The two we do consider are `MAKE` and `JOIN`.

A `MAKE` statement creates and names instances of functions. The semantics of `MAKE` are given by:

```
⟦MAKE fnname: name⟧ =
"MAKE [[MAKEITEM fnname name_fn]]"
```

The argument of `MAKE` is a list of lists. This allows for the abbreviations possible in ELLA. For example, in Section 4,

```
MAKE INV: l1,
     MUX: l3 out,
     REG: l2 l4.
```

is represented by:

```
MAKE [[MAKEITEM INV l1_fn];
      [MAKEITEM MUX l3_fn;MAKEITEM MUX out_fn];
      [MAKEITEM REG l2_fn;MAKEITEM REG l4_fn]]
```

Each application of MAKEITEM yields a Boolean value, and MAKE simply conjoins these together. The definition of MAKEITEM equates its two arguments, the second of which is a variable. The name of this variable is derived from *name* by adding '_fn'. It provides the link described above.

```
MAKEITEM (dev:*) fun = (fun = dev)
```

The JOIN statement 'connects' an input unit to a function. It is represented in a similar way to MAKE.

```
⟦JOIN unit -> name⟧ =
"JOIN [JOINITEM ⟦unit⟧ name name_fn ty_cast]"
```

JOINITEM is defined by:

```
JOINITEM in (out:***) fun (g:*→**) = ((fun (g in)) = out)
```

and JOIN conjoins the elements of the list it takes as argument. The fourth argument to JOINITEM is a type casting function which converts a tuple-valued signal into a tuple of signals. This is required to maintain the generality of the translation process. Type casting in the opposite direction is also required in places. This shows up in the example in Section 4 as an occurrence of the semantic constant MULTIPLE.

As an example, consider the following very simple series:

```
BEGIN
   MAKE INV: out.
   JOIN in -> out.
OUTPUT out
END
```

in is an input to the series, and INV is a function, as defined in Section 4. The HOL representation of the series is:

```
εOUTPUT.
 ∃out out_fn.
  SERIES
  [MAKE[[MAKEITEM INV out_fn]];
   JOIN[JOINITEM in out out_fn(λx'. εf1'. ∀t'. f1' t' = x' t')]]
  OUTPUT
  out
```

In this case, the type casting function reduces to the identity function `λx'. x'`. So, expanding with the definitions of MAKEITEM and JOINITEM, we get:

```
εOUTPUT.
 ∃out out_fn.
  SERIES
  [MAKE[[out_fn = INV]];
   JOIN[(out_fn ((λx'. x') in)) = out]]
  OUTPUT
  out
```

Applying the identity function λ`x'. x'` to its argument (`in`), and expanding with the definitions of `MAKE`, `JOIN` and `SERIES`, yields:

```
εOUTPUT.
 ∃out out_fn.
  (out_fn = INV) ∧
  (out_fn in = out) ∧
  (OUTPUT = out)
```

The first two equations can be combined and `out_fn` eliminated:

```
εOUTPUT.
 ∃out.
  (INV in = out) ∧
  (OUTPUT = out)
```

We can now eliminate the local variable `out` in a similar way:

```
εOUTPUT. OUTPUT = INV in
```

Finally, using the theorem ⊢ ∀`y. (`ε`x. x = y) = y`, we get:

```
INV in
```

So, the value of the series is simply the result of applying `INV` to the input.

3.6 Delays

ELLA has various forms of delay, but we only permit unit transport delay in our subset. So as not to change the syntax of ELLA, our system expects a value for the length of the delay. However this value must be `1`. The semantics of the ELLA `DELAY` construct are given by:

```
⟦DELAY(const, 1)⟧ = "DELAY1 ⟦const⟧"
```

where `DELAY1` is defined by:

```
DELAY1 (c:*→(bool)lifted) in (t:time) =
   ((t=0) ⇒ (εv. (c v) = (LIFT T)) | in(t-1))
```

The initial value *const* is an ELLA constant, i.e. a lifted predicate (see Section 3.1). Provided the predicate is true for exactly one value, the use of Hilbert's ε-operator yields that value. This value is used as the output of the delay at time `0`. For all later times, the output is the input at the previous time. (Time is represented by natural numbers).

3.7 Signals

There are places in ELLA where a fixed value can be used instead of the signal (function of time) that would normally be expected. We represent such occurrences by an application of the HOL semantic constant `SIGNAL` to the fixed value. `SIGNAL` is defined by:

```
SIGNAL (v:*) (t:time) = v
```

4 An example

The example described in this section is based on the example in [5]. It is not claimed to be typical of real design, but is merely intended to illustrate our approach to reasoning about ELLA designs with HOL.

The example will be presented by showing the input to the HOL-ELLA system supplied by a user, together with the system's response. The fragment of the session that comprises the specification and verification of the parity checker will be split into a number of subsessions, each displayed in a box. The language used for giving commands to the HOL system is called ML; this is an interactive functional programming language. The ML prompt is `#`; lines beginning with this are thus ML expressions or declarations. Lines not beginning with `#` are system output.

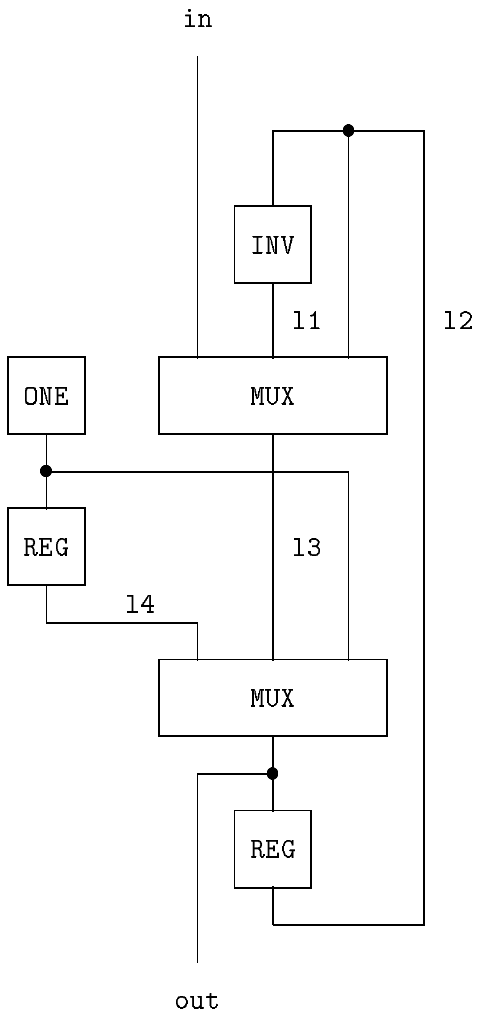
Work in the HOL system is organized into *theories*. Each theory introduces some definitions of types, constants or both. For the example described below, we create a theory called `ELLA-Parity` using the ML function `new_theory`. Commands to HOL are terminated by double semi-colons.

```
#new_theory `ELLA-Parity`;;
```

The parity checker is a device with an input `in` and an output `out`. The values on these lines will be modelled with an ELLA type `bit`. To define this type in the HOL-ELLA system, the appropriate ELLA text is enclosed between `BEGIN_ELLA` and `END_ELLA`. The system translates the ELLA text into a HOL type declaration, performs the declaration and then returns a list of canonical theorems (see Section 3.3).

```
#BEGIN_ELLA
#   TYPE bit = NEW (hi | lo).
#END_ELLA;;
[⊢ ∀e0 e1. ∃! fn. (fn hi = e0) ∧ (fn lo = e1);
 ⊢ ∀P. P hi ∧ P lo ⟹ (∀b. P b);
 ⊢ T;
 ⊢ ¬(hi = lo);
 ⊢ ∀b. (b = hi) ∨ (b = lo);
 ⊢ LIFT_hi = LIFT hi;
 ⊢ LIFT_lo = LIFT lo]
```

The design to be specified in ELLA is shown in the schematic diagram below.



This works by storing the parity of the sequence input so far in the lower of the two registers. Each time `hi` is input at `in`, this stored value is complemented. Registers are assumed to 'power up' in a state in which they are storing `lo`. The second register (connected to `ONE`) initially outputs `lo` and then outputs `hi` forever. Its role is just to ensure that the device works during the first cycle by connecting the output `out` to the device `ONE` via the lower multiplexer. For all subsequent cycles `out` is connected to `l3` and so either carries the stored parity value (if the current input is `lo`) or the complement of this value (if the current input is `hi`).

The devices making up this schematic will be modelled as ELLA functions with the indicated names. Here is the definition of `INV`:

```
#show_ella := true;;
true : bool

#BEGIN_ELLA
#   FN INV = (bit: in) -> bit:
#    CASE in
#    OF lo: hi,
#       hi: lo
#    ESAC.
#END_ELLA;;
[⊢ ∀in. INV in = CASE in
                 OF lo: hi,
                    hi: lo
                 ESAC]
```

The system response is a list of HOL theorems. In this case, there is only one theorem, a definition of a constant `INV` representing the semantics of the inverter. Before making the definition we set the pretty-printer to a state in which semantic constants are printed as ELLA text. We do this by setting the ML flag `show_ella`. The process of pretty-printing the semantic constants is straightforward. This illustrates that the translation from ELLA to HOL is merely syntactic. The semantics are expressed by the definitions of the HOL constants used. We can see the pure HOL term containing the semantic constants by flipping the state of the pretty-printer once more, and using the HOL-ELLA system function `def_of` to obtain the definition of `INV`.

```
#show_ella := false;;
false : bool

#def_of `INV`;;
⊢ ∀in.
   INV in =
   CASE
   in
   [OF[CONST lo,SIGNAL LIFT_hi;CONST hi,SIGNAL LIFT_lo]]
   (εx'. T)
```

Next the multiplexer `MUX` is defined with the pretty-printer set to `true`.

```
#show_ella := true;;
true : bool

#BEGIN_ELLA
#   FN MUX = (bit: cntl in1 in2) -> bit:
#    CASE cntl
#    OF hi: in1,
#       lo: in2
#    ESAC.
#END_ELLA;;
[⊢ ∀cntl in1 in2. MUX (cntl,in1,in2) = CASE cntl
                                       OF hi: in1,
                                          lo: in2
                                       ESAC]
```

Without special pretty-printing, the definition appears in HOL notation:

```
#show_ella := false;;
false : bool

#def_of `MUX`;;
⊢ ∀cntl in1 in2.
   MUX(cntl,in1,in2) =
   CASE cntl[OF[CONST hi,in1;CONST lo,in2]](εx'. T)
```

The third component used in the design is a register.

```
#show_ella := true;;
true : bool

#BEGIN_ELLA
#   FN REG = (bit) -> bit: DELAY(lo, 1).
#END_ELLA;;
[⊢ ∀in1'. REG in1' = DELAY (lo, 1) in1'] : thm list
```

```
#show_ella := false;;
false : bool

#def_of `REG`;;
⊢ ∀in1'. REG in1' = DELAY1(CONST lo)in1'
```

The variable `in1'`, representing the input signal, is generated by the HOL-ELLA system. The name of the variable is primed so that it cannot clash with a name used in the ELLA text (ELLA names cannot be primed).

The remaining component used in the design is `ONE`, which outputs a constant `hi`. Functions with no input are not allowed in ELLA. There are two standard methods of modelling such devices: with a dummy input, or as a constant. The second approach is used here; the value `hi` is supplied directly in the `JOIN` statement below.

```
#show_ella := true;;
true : bool

#BEGIN_ELLA
#   FN PARITY_IMP = (bit: in) -> bit:
#    BEGIN
#       MAKE INV: l1,
#            MUX: l3 out,
#            REG: l2 l4.
#       JOIN (in, l1, l2) -> l3,
#            hi           -> l4,
#            (l4, l3, hi) -> out,
#            out          -> l2,
#            l2           -> l1.
#    OUTPUT out
#    END.
#END_ELLA;;
[⊢ ∀in. PARITY_IMP in = BEGIN
                           MAKE INV: l1,
                                MUX: l3,
                                MUX: out,
                                REG: l2,
                                REG: l4.
                           JOIN (in, l1, l2) -> l3,
                                hi -> l4,
                                (l4, l3, hi) -> out,
                                out -> l2,
                                l2 -> l1.
                        OUTPUT out
                        END]
```

Observe that the pretty-printing of the ELLA series expands abbreviations in the `MAKE` statement. This is not necessary but prevents misleading output when only some of the semantic constants are eliminated. The pure HOL representation of `PARITY_IMP` can be seen below.

```
#show_ella := false;;
false : bool

#def_of `PARITY_IMP`;;
⊢ ∀in.
   PARITY_IMP in =
   (εOUTPUT.
     ∃l1 l3 out l2 l4 l1_fn l3_fn out_fn l2_fn l4_fn.
      SERIES
      [MAKE
       [[MAKEITEM INV l1_fn];
        [MAKEITEM MUX l3_fn;MAKEITEM MUX out_fn];
        [MAKEITEM REG l2_fn;MAKEITEM REG l4_fn]];
       JOIN
       [JOINITEM
        (MULTIPLE(λt'. (in t',l1 t',l2 t')))
        l3
        l3_fn
        (λx'. ε(f1',f2',f3'). ∀t'. f1' t',f2' t',f3' t' = x' t');
        JOINITEM
        (SIGNAL LIFT_hi)
        l4
        l4_fn
        (λx'. εf1'. ∀t'. f1' t' = x' t');
        JOINITEM
        (MULTIPLE(λt'. (l4 t',l3 t',SIGNAL LIFT_hi t')))
        out
        out_fn
        (λx'. ε(f1',f2',f3'). ∀t'. f1' t',f2' t',f3' t' = x' t');
        JOINITEM out l2 l2_fn(λx'. εf1'. ∀t'. f1' t' = x' t');
        JOINITEM l2 l1 l1_fn(λx'. εf1'. ∀t'. f1' t' = x' t')]]
      OUTPUT
      out)
```

This can immediately be simplified using the tool `ELLA_fnbody_CONV`.

```
#let PARITY_IMP_SIMP =
# CONV_RULE (ELLA_DEF_CONV (ELLA_fnbody_CONV ALL_CONV))
#  (def_of `PARITY_IMP`);;
PARITY_IMP_SIMP =
⊢ ∀in.
   PARITY_IMP in =
   (εOUTPUT.
     ∃l1 l3 out l2 l4.
      (MUX(in,l1,l2) = l3) ∧
      (REG(λt'. LIFT_hi) = l4) ∧
      (MUX(l4,l3,(λt'. LIFT_hi)) = out) ∧
      (REG out = l2) ∧
      (INV l2 = l1) ∧
      (OUTPUT = out))
```

The definition of the design is now complete. The next step is to specify the desired behaviour. This uses a HOL function `NOT` for complementing values of type `bit`.

```
#new_definition (`NOT`, "NOT b = (b=hi) ⇒ lo | hi");;
⊢ ∀b. NOT b = ((b = hi) ⇒ lo | hi)
```

The function `PARITY` is defined by primitive recursion so that `PARITY n s` is `hi` if and only if there are an even number of `hi`'s in the sequence: `s 0, s 1, ... , s n`. `SUC` is the successor function.

```
#new_prim_rec_definition
# (`PARITY`,
#  "(PARITY s 0 = hi) ∧
#   (PARITY s (SUC n) =
#     (s(SUC n)=hi) ⇒ NOT(PARITY s n) | PARITY s n)");;
⊢ (∀s. PARITY s 0 = hi) ∧
  (∀s n. PARITY s(SUC n) =
         ((s(SUC n) = hi) ⇒ NOT(PARITY s n) | PARITY s n))
```

From these definitions one can then use the theorem proving tools in HOL to deduce:

```
⊢ ∀in. PARITY_IMP(LIFT o in) = LIFT o (PARITY in)
```

The circle 'o' denotes composition of functions, e.g., `((f o g) x)` is equivalent to `f(g(x))`. Informally, the theorem above states that, provided the input signal to the parity checker is specified for all time, the output will also be specified for all time and its value will be determined by the function `PARITY`. The proof of this correctness theorem follows the proof of the example in [5]. Extra work is required to eliminate the semantic constants.

5 Conclusions and Future work

We have described the first stage of a project to link the HOL formal logic, and its associated formal proof system, with the ELLA hardware description language and its associated CAD system. ELLA is semantically embedded in higher order logic through representing the semantics of an ELLA text by a HOL term. A parser and pretty-printer have been built which allow an ELLA text to be input and output from HOL. The HOL term represents the behaviour of the associated ELLA text, and can be manipulated to verify the correctness of a design with respect to a requirements specification in HOL.

The parser and pretty-printer support an ELLA syntactic view of the behaviour. However, all reasoning about the behaviour is done in HOL using standard proof techniques. Tools are provided to turn the mechanically generated HOL term representing the behaviour into a term which is of a simpler, standard form, e.g., see `ELLA_fnbody_CONV` in Section 4.

We have shown a number of ELLA constructs and the corresponding HOL definitions which represent the semantics of these constructs. The translation has been kept as simple as possible to provide a straightforward relationship between the ELLA syntax and its semantics in HOL. Particular care must be taken with the ELLA unspecified value `?`, and the treatment of this using lifting has been presented. In the parity circuit example, a complete ELLA design is input into the HOL system. This example illustrates how the parser and pretty-printer support an ELLA syntactic view of behaviour, and allow one to move easily from the ELLA text to reasoning about the behaviour in HOL.

The first stage in integrating the formal system and conventional CAD system has been the provision of the interfacing tools and the completion of small examples. When using the composite system there is a choice regarding where to use the HOL and ELLA systems in the design process, and where to describe behaviour in ELLA and where to use HOL. The functionality of each system complements the other: ELLA provides a simulator and conventional CAD tools, HOL provides machine checked formal verification. When doing the design, the user may explore different options using the simulator to animate behaviour. Having converged on a design and specification, the HOL system can be used to formally verify that the design does meet its specification.

On the question of where to describe behaviour in ELLA and where to use HOL, we have assumed in this paper that the correctness requirements for a design are specified directly in HOL rather than ELLA. However, it is quite likely that some intermediate levels of specification might be in ELLA. Using HOL at the highest specification levels has advantages, such as providing support for a non-operational style of specification, and allowing mappings between specifications to be described and verified. It may be that engineers will not share these preferences, which reflect our background and interest in formal proof. However, we feel that it is better to base our work initially on the strengths of ELLA and HOL, and to find out experimentally where the languages and systems best complement each other.

The next stage of the project will exercise the prototype HOL-ELLA system. We hope to discover what improvements are needed, and also develop a methodology for design and verification using the system. This work will involve several design and verification case studies; these will include a standard I/O chip and a simple processor.

There are many questions to be addressed when doing the case studies:

- How much of the proof can be conducted while maintaining an ELLA syntactic view of the behaviour?

- Should the designer move quickly from the ELLA texts to the corresponding HOL and do all proofs with a HOL view of behaviour?

- Could one use mostly automatic verification tools at the lower levels of design [11], and expect that only the higher-level or system aspects be verified in HOL by "experts"?

- How much does specification help/guide the design?

- What design styles best accommodate formal methods?
- What verification tasks can be automated or made easier?

In trying to find answers to these questions we hope to develop a methodology for using the HOL-ELLA system, and find out what extensions and changes are needed to our prototype tools. A single methodology cannot cater for all designs, and the first case studies can only provide examples of a small variety of circuits. Nevertheless, the case studies will provide the initial evaluation of the ideas of the project, provide the basic methodology for using the system, and indicate the future directions of the work.

The HOL and ELLA systems are complementary systems in the hardware design process. The integration of the two systems may encourage users of each system to appreciate, learn and use the other system. The interaction of formal methods and conventional design techniques is not well understood and it is hoped that the linking of HOL and ELLA may help to establish some better understanding of this area.

6 Acknowledgements

Our thanks to Kees Goossens of University of Edinburgh LFCS for advice on the choice of the ELLA subset and for informative discussions about the ELLA unspecified value and `CASE` statement. Our thanks also to Tom Melham of Cambridge for providing proof support for integers and n-bit words.

We are grateful to Praxis plc for setting up and managing the SERC/DTI project "HOL Verification of ELLA Designs" (IED2/1/1129) and for supplying us with the ELLA system. We thank Dave Deans of Praxis, the project manager.

Richard Boulton was fully supported by IED2/1/1129, and is now supported by an SERC studentship. John Herbert is partly supported by IED2/1/1129 and partly supported by IED3/1/1036. John Van Tassel was supported by Esprit Basic Research Action 3215.

References

[1] Borras, P. *PPML Reference Manual & Compiler Implementation.* Sema Group, June 6, 1989. Appears in *The CENTAUR Reference Manual.* Contact: CENTAUR Distribution, INRIA Sophia-Antipolis, 2004 Route des Lucioles, 06565 Valbonne CEDEX, FRANCE (centaur@mirsa.inria.fr).

[2] Boyer, R.S. and Moore, J.S. *A Computational Logic.* Academic Press, 1979.

[3] Church, A. *A Formulation of the Simple Theory of Types. Journal of Symbolic Logic*, **5**, 1940.

[4] Gordon, M., Milner, R., Morris, L., Newey, M. and Wadsworth, C. *A Metalanguage for Interactive Proof in LCF.* Proceedings of the ACM Conference on the *Principles of Programming Languages*, Tuscon, 1978. (Also issued as Report CSR-16-77, Department of Computer Science, University of Edinburgh, 1977).

[5] Gordon, M.J.C. *HOL: A Proof Generating System for Higher-Order Logic.* University of Cambridge, Computer Laboratory, Tech. Report No. 103, 1987; Revised version in G. Birtwistle and P.A. Subrahmanyam (eds), *VLSI Specification, Verification and Synthesis*, Kluwer, 1987.

[6] Gordon, M.J.C. *Mechanizing Programming Logics in Higher Order Logic*, in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam, Springer-Verlag, 1989.

[7] Halpern, J., Manna, Z. and Moszkowski, B. *A Hardware Semantics based on Temporal Intervals.* In the proceedings of the *10-th International Colloquium on Automata, Languages and Programming"*, Barcelona, Spain, 1983.

[8] Institute of Electrical and Electronics Engineers. *IEEE Standard VHDL Language Reference Manual.* IEEE Press, New York, 1988.

[9] Milner, R. *A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences*, **17**, 1978.

[10] Morison, J.D., Peeling, N.E. and Thorp, T.L. *ELLA: Hardware Description or Specification?*, Proceedings IEEE International Conference, CAD-84, Santa Clara, November, 1984.

[11] Pygott, C.H. *NODEN-HDL: An Engineering Approach to Hardware Verification.* In *The Fusion of Hardware Design & Verification*, edited by G.J. Milne, Elsevier Science Publishers, 1988.