Experience with embedding hardware description languages in HOL

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

University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England.

Abstract

The semantics of hardware description languages can be represented in higher order logic. This provides a formal definition that is suitable for machine processing. Experiments are in progress at Cambridge to see whether this method can be the basis of practical tools based on the HOL theorem-proving assistant. Three languages are being investigated: ELLA, SILAGE and VHDL. The approaches taken for these languages are compared and current progress on building semantically-based theorem-proving tools is discussed.

Keyword Codes: F.4.1; B.7.2; I.2.3
Keywords: Mathematical Logic; Integrated Circuits, Design Aids; Deduction and Theorem Proving.

1 Introduction

Hardware can be directly specified in the notation of mathematical logic [11, 14, 18], but this form is unacceptable to many designers and is also unsuitable for input to CAD tools such as simulators and circuit synthesizers. The work described here aims to support the use of conventional hardware description languages (HDLs) within a general theorem-proving environment. The approach taken is *semantic embedding* in higher order logic [13, 19], and the theorem-proving infrastructure is provided by the HOL system [12]. Three languages are being investigated by separate teams: ELLA (by Boulton, Harrison and Herbert), SILAGE (by A. Gordon) and VHDL (by Van Tassel). The HOL-ELLA project [1] is the largest effort and has been running longest. The other two projects have benefited from experience and tools arising from early experiments with ELLA. Note that none of the three languages considered has a formal semantic specification; constructing suitable specifications is a major part of each of the projects.

The three projects surveyed here all have different goals. The aim of the HOL-ELLA project was originally to explore the combination of conventional CAD tools (namely the ELLA simulator) with formal proof (namely HOL). The intention was to build a hybrid system combining HOL and ELLA. This hybrid would then be used to explore design methodologies that mixed conventional methods with formal methods. As the project evolved, the emphasis shifted to the embedding of ELLA in HOL; the current goal is to produce a self-contained prototype design environment entirely within HOL. This is being driven by two examples: a simple real time controller and a floating-point square-root chip. The designs for these will be expressed in ELLA and then be verified against specifications written directly in logic.

The aim of the HOL-Silage project, which is part of the ESPRIT CHEOPS project, a collaboration between Cambridge, IMEC in Leuven, and Philips in Eindhoven, is to formalize the semantics of Silage and to use the result to validate Silage-to-Silage transformations. Eventually it is hoped to implement a transformation manager that will permit users to prove new transformations prior to their being added to a library. Current work (in collaboration with Catia Angelo of IMEC) concentrates on using the Silage semantics to prove correct examples from practical applications of Silage at IMEC.

The aim of the HOL-VHDL project is to explore the possibility of providing a formal semantics for a subset of VHDL that formalizes the spirit of the existing informal semantics. It is eventually hoped to use the semantics for a variety of purposes, including the validation of algebraic laws for manipulating VHDL processes. However, the current initial phase of the work is concentrating on finding a tractable semantics.

The general technique of embedding a conventional notation, such as a hardware description language, in a mechanized formal system, such as HOL, offers several possible benefits:

- formal definition of the semantics of various notations;
- mechanized support for syntax and type checking;
- a framework for establishing metatheorems about the notation (such as consistency);
- support for formal proof about programs;
- derivation of proof rules for notation (such as equational transformations);
- verification of compilers.

The most important part of doing a formal embedding is the definition of semantics. The HDLs presented here do not have a fully defined formal semantics. The few HDLs which have been designed with semantics as the primary consideration (e.g., Funnel [33]) are not part of CAD environments, and are not yet used in practical applications.

In defining a formal semantics for the HDLs, it may only be possible to give a semantics for a subset of the notation. There is often a balance between obtaining a useful subset of the notation and having a tractable formal semantics. A wider subset might be covered by re-defining the intended meaning of some constructs so that the formal semantics remains tractable. This departure from the 'correct' semantics must be treated carefully (cf., Section 7.7).

HOL is a foundational system which means that one can *define* new constants in a way that does not affect the logical consistency of the system. This is important as it means the embedding of an HDL can be achieved using these definitions rather than by introducing arbitrary axioms to describe the semantics.

The organization of the paper is as follows. Higher order logic is briefly reviewed in Section 2. In Section 3, the similarities and differences between the three hardware description languages are discussed. Section 4 discusses the general principles of semantic embedding. Section 5 gives an overview of the projects. The circuit used to illustrate the semantic embeddings of the three languages is described in Section 6. Sections 7, 8 and 9 outline the embeddings of ELLA, SILAGE and VHDL, respectively. Finally, in Section 10 the lessons learned so far are discussed.

2 Higher order logic

Higher order logic is a generalization of first-order logic that allows variables to range over functions and predicates. There are many kinds of higher order logic, but they all impose some kind of type discipline on the use of functions and predicates (types are required to avoid inconsistency). Higher order logics differ in the sophistication of the type system they provide. One of the simplest type systems is due to Church [3] and is similar to the programming language type disciplines found in functional languages like ML and Miranda. This is the system that underlies the work described here, but knowledge of the details will not be needed. More elaborate type systems supporting 'dependent types' and 'subtypes' are sometimes used for hardware specification [15, 21] and provide, at a cost, greater expressive power. However, simple types are adequate for the needs of this paper (though it is probable that some notational improvement would be possible if more sophisticated types were available).

Standard predicate calculus notation will be used. In higher order logic there is a single syntactic class of *terms*. There is no need (as there is in first-order logic) for a separate class of *formulae*, because these can be identified with Boolean-valued terms. The constants T and F denote 'true' and 'false', respectively. Predicates will be identified with Boolean valued functions. If P is a predicate, then $P(x)$ means 'x has property P'. Often the brackets will be omitted and just $P\ x$ will be written. If t, t_1 and t_2 stand for Boolean terms, and $t[x]$ stands for some Boolean term containing the variable x, then: $\neg t$ means 'not t', $t_1 \wedge t_2$ means 't_1 and t_2', $t_1 \vee t_2$ means 't_1 or t_2', $t_1 \implies t_2$ means 't_1 implies t_2', $\forall x.\ t[x]$ means 'for all x it is the case that $t[x]$', $\exists x.\ t[x]$ means 'for some x it is the case that $t[x]$' and $\varepsilon x.\ t[x]$ (a notation due to Hilbert) denotes an arbitrarily chosen value a such that $t[a]$ (if no such value exists then an arbitrary value is chosen). The epsilon operator is generally very useful for denoting terms known to exist without introducing additional constant symbols. If t_1 and t_2 are terms of the same type, then $t_1 = t_2$ means 't_1 equals t_2'(i.e., t_1 and t_2 denote the same value), in the case that t_1 and t_2 are functions, this means that they produce the same result when applied to the same arguments (this is called *extensionality*). The special notation $\lambda x.\ t$ denotes the function, which is sometimes written informally as $x \mapsto t$, that maps an argument a to the result of substituting a for x in t. For example, $\lambda x.\ x + 1$ denotes the successor function (i.e.,

$n \mapsto n+1$). The notation: $(t \to t_1 \mid t_2)$ means the conditional 'if t then t_1 else t_2'.

3 Comparison of HDLs

The three languages ELLA, SILAGE and VHDL differ along several dimensions, including intended applications, underlying behavioural model and generality. This makes it hard to compare them. Furthermore, the work on embedding each language has been done by different people having different stylistic tastes in writing formal specifications, and has been driven by different project aims. This section compares the three hardware description languages, while Section 5 gives an overview of the projects.

3.1 Size and style of languages

ELLA [4, 29] is not especially big for an industrial-strength language, but is considerably larger than the toy languages often used in research into formal methods and computer language semantics. The language is mostly functional in style though there are some imperative features. These require some notion of state within the semantics, whereas for the purely functional subset it is possible to give a semantics in terms of types and functions.

SILAGE [7, 16, 17] is a small dataflow language designed for specifying digital signal processing (DSP) devices. There are several dialects of SILAGE and no agreed standard; the HOL-SILAGE project deals with the IMEC dialect. The language is declarative in style. A program is an unordered series of equations specifying the values of signals.

VHDL [20] is a large HDL. In syntax it is reminiscent of Ada, but the similarity ends there. There are two main classes of statements in the language: concurrent and sequential constructs. Architectural elements are made up of concurrent statements that contain sequential statements. The syntax is in the style of an imperative programming language. The VHDL tools use an event-driven simulator, or interpreter. An operational-style semantics is therefore particularly suitable for VHDL, whereas both ELLA and SILAGE lend themselves to denotational techniques.

3.2 Level of abstraction

ELLA takes a fairly high-level view of hardware, though designs can be described at various levels of abstraction. The ability to declare new datatypes makes this easy.

VHDL can be used to describe hardware from a high level to a low level. The ideal design cycle would see VHDL used to describe the high-level behaviour of a system and throughout the expansion of the design into an implementation.

SILAGE programs can be thought of as signal flow graphs, and are intended to be free of any architectural commitment.

3.3 Application areas of each language

ELLA is intended as a general-purpose hardware description language, and has the necessary infrastructure to support this. The language takes a structural view of hardware where designs are built up from components which are declared as functions or macros.

VHDL is, like ELLA, meant to be used as a general-purpose language. Each is intended to support design from initial specification to implementation.

SILAGE is intended for high-level descriptions of DSP circuits in areas such as image processing or cryptography. SILAGE is not meant for low-level description of circuits; such descriptions of SILAGE programs are intended to be synthesized automatically.

3.4 Differences in timing models

The ELLA language has a timing model based on an implicit, universal time base. Time is discrete and has a beginning. All variables in ELLA texts represent sequences over time. Special constructs such as the `DELAY` operator refer explicitly to time. The ELLA timing model can be represented in HOL using natural numbers, and signals as functions from these natural numbers to the type of the data. For components with state it is possible to specify initial values.

The IMEC dialect of SILAGE also makes use of a global clock, and all signals are produced in phase at the same rate. SILAGE expressions denote infinite arrays of sample values, indexed by time. Time is represented by the integers. Computation starts at time 0, and negative times are used only to initialize delays. Multirate and aperiodic signals can be expressed in other dialects but are not present in IMEC SILAGE, and hence not dealt with in HOL-SILAGE.

VHDL makes use of a non-decreasing global clock which begins at time 0. Because it is event-driven in nature, VHDL's clock does not advance in regular discrete time steps. Instead, it moves forward on demand to the nearest collection of events.

4 Semantic embedding

There are two approaches to embedding HDLs in logic:

(1) Represent the abstract syntax of HDL programs by terms, then define *within the logic* semantic functions that assign meanings to the programs.

(2) Only define semantic operators in the logic and arrange that the user-interface parse input from HDL syntax directly to semantic structures, and also print semantic representations in HDL syntax.

These two approaches will be referred to as *deep embedding* and *shallow embedding*, respectively. An embedding is deep or shallow depending on whether the syntax of the HDL is represented by a HOL type or ML type respectively. Each of these has advantages and disadvantages. The advantage of deep embedding is that it allows reasoning about classes of programs, since one can quantify over syntactic structures [24]. Setting up HOL types of abstract syntax and semantic functions can be a lot of work. The advantage of shallow embedding is that this work is avoided; the interface handles the mapping between HDL programs and their semantic representations. Since this mapping is outside the logic it is not subject to the rigour of mechanized formal specification and proof, and so the resulting system is less secure. Furthermore, it is not possible to directly state theorems about classes of programs, since program structures are not represented in the logic (e.g., variables ranging over programs are not available). However, it may be possible to formulate general results at the semantic level by proving properties of semantic operators (this is what is being done for SILAGE), but there may be results that can only be expressed by saying that all HDL programs of a certain form have a particular property.

Both the HOL-ELLA and the HOL-SILAGE projects are using shallow embedding: only the semantics is represented in the HOL logic. The HOL-VHDL project uses deep embedding, since it aims to model in the logic the VHDL simulation cycle, which is represented by an interpreter on program texts.

The approach to the semantics of ELLA and SILAGE is quite similar; they are both given 'denotationally', but the denotations of ELLA programs are functions, whereas the denotation of SILAGE programs are relations. However, in both cases the meaning of a program is couched in terms of infinite sequences of values, representing signals. The semantics of VHDL, on the other hand, is quite different. This is completely operational and consists in a formalization within higher order logic of an idealized VHDL simulator. Reasoning about VHDL is conducted by reasoning about runs of the simulator.

5 Overview of projects

There are different concerns when it comes to embedding different HDLs in HOL. The original aim of the HOL-ELLA project was to explore the combination of conventional CAD tools with a mechanical theorem prover. The emphasis has shifted towards the embedding of ELLA in HOL, but the original aims have had a significant influence on the choice of the subset of the ELLA language in use.

Since the project is industrially oriented, the subset chosen was quite substantial. A declarative (functional) subset was selected since this made the embedding in HOL more tractable. However, many of the more complex language structures left out of the subset (e.g., *sequences*) can be thought of as abbreviations of longer texts and might be transformed into the subset (see [29]).

Since the connection between conventional tools and theorem proving techniques was being investigated, the HOL style of definition was closely adhered to so that the applicability of known techniques to a real hardware description language could be investigated. This, together with the difficulties of formally supporting a substantial subset of a real hardware description language led to the use of a shallow embedding in which ELLA declarations become HOL definitions.

Shallow embedding has the disadvantage that only specific ELLA texts can be reasoned about formally because ELLA texts are not explicitly represented in the logic. However, it is still possible to prove generic properties of ELLA constructs below the declaration level. One advantage of transforming ELLA declarations into HOL definitions is that they can be freely mixed. In particular, behavioural specifications can either be given in ELLA and translated or can be written directly in HOL. Shallow embedding is simpler than deep for large languages, since it avoids building infrastructure for the syntax of the language within the logic.

The semantics in HOL was originally written to reflect very accurately the behaviour of the industrial compiler and simulator for ELLA. However, with the change of emphasis of the project the semantics has been simplified to make formal reasoning easier. The cost is a minor inconsistency between the semantics and the behaviour of the ELLA simulator. The ELLA designers have indicated that the new semantics accurately reflects their original intention.

The eventual goal of the HOL-Silage project is to build an interactive system to manage source-to-source meaning-preserving transformations of Silage programs. There being no existing formal semantics, a substantial subset of Silage has been defined using set theory and logic. The HOL-Silage subset contains all the IMEC dialect, except that the type system is much simplified, and many arithmetic, bitwise and vector operations are omitted. This policy was adopted because examples of program transformations at IMEC have exclusively dealt with control flow rather than with details of arithmetic. The definition has been mechanized as an ML program mapping Silage syntax into the HOL logic. Experiments are under way to see how the HOL system can be used alongside the IMEC simulator and synthesizers to support transformational design of Silage programs.

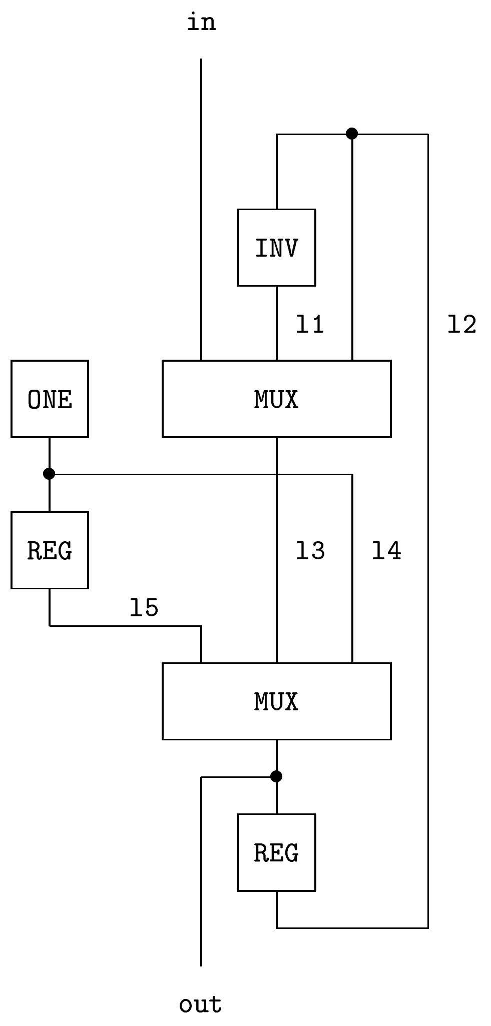
The goals of the VHDL project are twofold. The first one is to develop a tractable semantics for a subset of VHDL. The second is to make use of the semantics to reason about VHDL programs. Support will be offered for reasoning about properties of individual VHDL programs, as well as for proofs involving two VHDL texts purporting to be equivalent.

VHDL is a radically different language from both ELLA and Silage. Its semantics, even at the informal level, is specified by the way different language constructs interact with the VHDL simulation engine. This lends itself to a more operational style of semantics than those used in the other two projects. This particular style is advantageous in that it will allow us to work with general classes of VHDL programs, and to derive useful laws about their operation. It also lends itself well to capturing the intuition behind the way VHDL users actually think about their programs. A side effect of developing the semantics of VHDL in this manner is the specification of what constitutes a VHDL simulator. This can then be used as a design document for language implementors.

At the current stage of development, only an individual iteration of the simulation loop has been formalized. Work is under way to incorporate this individual simulation cycle with a more complete view of the VHDL simulation engine. Nevertheless, it is possible to make use of the single iteration to give a flavour of the kind of reasoning involved in embedding VHDL in HOL.

6 An example circuit

The example described in this section is based on the example in [12]. It is not claimed to be typical of real design, but is merely intended to illustrate the HOL semantics for the three hardware description languages. The example is a bit-serial parity checker. Its design 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 *true* is input at `in`, this stored value is complemented. Registers are assumed to 'power up' in a state in which they are storing *false*. The second register (connected to `ONE`) initially outputs *false* and then outputs *true* 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 *false*) or the complement of this value (if the current input is *true*).

7 Embedding ELLA

In ELLA, designs are built up from components which are declared as functions or macros (essentially parameterized functions). These can be implicitly connected together by function application. Constructs are also present which allow the designer to make connections between components explicitly.

7.1 ELLA text for the parity checker

The following is a complete specification in ELLA of the parity checker. This gives a flavour of the language; the example is explained in more detail below.

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

```
FN INV = (bit: in) -> bit:
 CASE in
 OF lo: hi,
    hi: lo
ESAC.
```

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

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

```
FN PARITY_IMP = (bit: in) -> bit:
 BEGIN
    MAKE INV: l1,
         MUX: l3 out,
         REG: l2 l5.
    JOIN (in, l1, l2) -> l3,
         hi           -> l5,
         (l5, l3, hi) -> out,
         out          -> l2,
         l2           -> l1.
 OUTPUT out
 END.
```

The first line of the text is a type declaration. All types used in ELLA have to be declared explicitly; there are no primitive types. In the above example the basic type is declared as a two-valued enumerated type `bit`, which has the values `hi` and `lo`. Intuitively this corresponds to the two possible logic levels on a wire. Note however that an ELLA datatype is an abstract object and the nature of its mapping onto a wire *or wires* is not defined as part of the ELLA text — as a matter of policy ELLA abstracts away from such low-level implementation details.

The ELLA text then defines four different circuit blocks, respectively an inverter, a 2-to-1 multiplexer, a D-type latch or 'register', and finally a complete parity checker.

ELLA includes no logic gates or other circuit elements as primitive. The definitions of `INV` and `MUX` illustrate one of the main tools for constructing such gates, namely the `CASE` statement. This is interpreted like a `CASE` statement in an Algol-like programming language. For example, the `CASE` statement in the definition of `INV` says that if the input is `lo` then give the output `hi`, and if the output is `hi` give the output `lo`.

All elements built from such primitives are delay-free. The `REG` element introduces time delays using the `DELAY` primitive. The declaration of `REG` simply says that the output has value `lo` at time zero, and thereafter the input delayed by one time unit.

The most complex declaration is the last one, specifying how the entire circuit is constructed from the above-mentioned elements. The `MAKE` statement declares names for the instances of each sub-element; for example `l1` represents the only inverter, while `l3` and `out` are both multiplexers. The `JOIN` statement specifies, as its name suggests, how the parts are then connected.

In a `JOIN` statement the names of the elements have a dual role. When they appear on the left of an arrow, they represent the output of that element; conversely on the right of an arrow, they represent its input. Then the arrow simply specifies how the output of an element or elements is connected to the input of others. For example `out -> l2` means connect the output of the multiplexer `out` to the input of the register `l2`.

The component `ONE`, which appears in the description of the circuit in Section 6, is not used in the ELLA description. Instead the value `hi` is used directly as an input.

7.2 The semantics of ELLA in HOL

ELLA constructs are translated into HOL via a shallow embedding which reflects the intended meaning of constructs in the ELLA subset. The subset is quite substantial, but avoids parts of ELLA which do not have a simple functional semantics. Note that there is no formal semantics of ELLA against which to judge this embedding — the main point of reference is the ELLA simulator, but this is imperfect in various respects. Since ELLA is embedded in logic by means of conservative extension, the semantics is at least consistent in the sense of not leading to logical contradiction.

The terms of HOL produced by the translation are pure logical terms, but include various *semantic constants* which are constructed using HOL's definitional mechanisms. Although it is possible (and often necessary as part of a proof of correctness) to expand these constants into their primitive logical meanings, the constants are useful in maintaining a close syntactic link between an ELLA text and the result of translation into HOL.

This makes the translation process simpler and more modular, keeps the size of the resulting HOL more manageable, and most importantly allows an approximate inverse mapping back to ELLA, allowing the pretty-printing of HOL terms as ELLA text. This last consideration justifies the independent existence of semantic constants which simply correspond to things like equality, which already have a name in HOL.

Before launching into an examination of the semantics of the parity checker in the HOL-ELLA system, a few points are in order about the general scheme.

ELLA types are modelled using corresponding type definitions in higher order logic. These are not primitive in HOL, but are made available in terms of primitive types using a procedure written by Tom Melham [26].

These primitive types may be built up into composites, which may be thought of as representing collections of wires such as buses. These are modelled as lists or tuples in HOL, depending on precise structural details. These will not feature in our example, so will not be discussed further.

Time in ELLA is modelled by the natural numbers starting at zero. A type synonym `time` is provided for the type of natural numbers, but this is purely surface syntax, so all the HOL theory of natural numbers applies to `time` — for example it is possible to perform proofs by induction to show that a certain property holds for all time.

A signal in ELLA is simply represented as a function from time into the corresponding type of values. Such signals are the objects which are generally manipulated in a HOL proof of correctness: higher-order logic places no inconvenient restrictions on their use. In the example which follows, the reader may simply identify 'wire' with 'function from `time` to `bit`'.

7.3 Translating CASE statements

The definition of the `INV` block is translated into the following piece of HOL:

```
INV in = CASE in[OF[(CONST lo,SIGNAL hi); (CONST hi,SIGNAL lo)]](εx.T)
```

Although not especially readable, the correspondence with the ELLA text should be fairly clear. The semantic constant `CONST` essentially corresponds to equality, so `CONST lo` means *is equal to* `lo`. The semantic constant `SIGNAL` takes a constant and generates a signal which has that value for all time. The `CASE` and `OF` constants behave in the way one would expect: when applied to a signal `x`, the entire `CASE` statement means: look at the list of pairs given as an argument to `OF`, and try applying the first element of each pair to the signal `x` at the current time. For instance, if the `CONST lo` predicate returns true, it means that the signal has value `lo` at the current time. For the first predicate which succeeds, the output is the second element of the corresponding pair. If none of them succeed, the output is the term at the end (`εx.T`), the unknown value.

The unknown value is discussed in more in more detail in Section 7.7, but it is not important for this example because every signal has either value `hi` or `lo` at any time, so the default clause of the case statement is never reached. This can be proved in HOL using a theorem returned by the type definition package, and using this fact as well as expanding with the definitions of the semantic constants, the following theorem can be proved, which specifies in pure HOL the behaviour of the inverter:

```
∀in t. INV(in) t = (in t = lo) → hi | lo
```

This may be read as: the output of the inverter, given the input `in`, at time `t` is either `hi` or `lo` when the signal `in` at that time is `lo` or `hi`, respectively. Clearly this reflects the intended meaning of the inverter.

7.4 Translating delays

The semantics of delays is quite simple. The subset of ELLA handled by the HOL-ELLA system allows only unit delays, and these are implemented by a semantic constant `DELAY1` which has the following property:

```
∀in t. DELAY1(in)(t + 1) = in(t)
```

The above theorem states that the output of the delay element at time $t+1$ is equal to its input at time t. The behaviour at time zero is specified in a slightly involved way, but it is fairly easy to prove that the output of `REG` has value `lo` at time zero, as requested in the declaration.

7.5 Translating arbitrary networks

The `MAKE` and `JOIN` statements in ELLA allow more or less arbitrary connections between functional blocks. This means that it is not possible in general to translate the ELLA into a HOL function — HOL functions are all total, and so cannot reflect unrealizable configurations like wiring the input and output of a (zero-delay) inverter together.

The HOL-ELLA system therefore uses the ε-operator. Intuitively, a network is represented as the choice term meaning *some function which has the appropriate property*, namely the property of satisfying a set of equations representing connections. Internal wires are represented as existentially quantified variables, as is conventional in hardware verification in pure HOL.

This means that no restrictions are placed on the translation process, nor any proof obligations involved in making the translation go through. However it leaves some work to do in HOL afterwards to show that the resulting block does represent a realizable function (and indeed, this may be impossible, but that would be indicative of a design error such as wiring outputs together).

There do exist various proof tools which provide automatic support for the process. Some of these tools are specific to the HOL-ELLA system, whereas others are common to hardware verification in pure HOL. A good example is *unwinding* (removal of existentially quantified 'internal wires'). In particular, most sensible circuits without loops can be reduced to an equivalent functional form without user proof effort.

The case of the parity checker is slightly more complex since it does feature a loop. However the output at time t is only fed back as an input at time $t+1$, so it is a relatively straightforward exercise to define a primitive recursive function in HOL and prove it is the *unique* function satisfying the constraints. This shows that the ε-term representing the parity checker is equal to this function.

The result is a pure HOL function representing the parity checker. This can then be proved in HOL against its specification using standard methods.

7.6 Proof methodology in the HOL-ELLA system

The general objective of the HOL-ELLA project is to investigate how formal verification can be merged with existing CAD tools. Although this is notionally tied to ELLA, the lessons learned and proof tools written have had wider application.

Since most hardware engineers are unfamiliar with the formal details of logical proof, it was felt to be worth exploring the possibility of working with an ELLA representation of the underlying logic. This is the rationale behind incorporating an ELLA pretty-printer into the system. However it seems unlikely that ELLA offers the expressive power to state many desirable properties of a specification. (On the other hand it should be possible to use the system to justify, with respect to our semantics, the correctness of appropriate ELLA transformations.)

In the case studies so far undertaken, the usual method is to eliminate the semantic constants and generally simplify to a logical term representing the implementation. This is then proved correct with respect to a specification (also a pure logical term) just as in normal hardware verification in pure HOL. The simplification is assisted by various proof procedures.

7.7 Lifting

ELLA features an unknown value, which is denoted by `?` in ELLA texts. This is encoded in the HOL translation using the ε-operator applied to a predicate which is always true. This seems to be appropriate in that it represents an unknown value of the appropriate signal type. However this does not entirely reflect the behaviour of the ELLA simulator in all contexts.

In a `CASE` statement, 'choosers' (that is, the set of values of a given type to be matched against) may be written as a type name. This matches any value of the relevant type, including the unknown value. However a chooser consisting of an explicit enumeration of all the values of a type does *not* match the unknown value. This means that in this context, the unknown value really behaves like an extra value in the relevant type.

This is not possible to implement in the straightforward interpretation of the unknown value explained above, because although an ε-term is an *unknown* value of a given type, it *is* known (to be more precise, is provable in HOL) that it is equal to one of the possible values of that type.

Because of this mismatch, the first version of the HOL-ELLA system used a more elaborate semantics featuring *lifting*. This means that for every type declared in ELLA, the type in the HOL translation consists of one special unknown value denoted by `UU` *and* all the values corresponding to a straight implementation of the ELLA type. Values other than the unknown element are denoted by `LIFT t` where `t` is a member of the enumerated type.

The presence of lifting makes proofs in HOL significantly more difficult, as well as obscuring the intuitive meaning of what would otherwise be quite a clear semantics. The problems lie deeper than simply having the operators `LIFT` and `UNLIFT` scattered through the proof, although the unpleasantness of that should not be understated.

To get a lifted output at a particular time, circuit elements normally need lifted inputs at that time (or in the case of a delay block, at the previous time). For example the following might be true of an inverter:

```
(in t = LIFT x) ⟹ (INV(in) t = LIFT(¬x))
```

Unfortunately, theorems like the above are very inconvenient to use, because they cannot be used as (unconditional) rewrite rules. It is tempting to use instead theorems of the form:

```
INV (LIFT o in) t = LIFT (¬(in t))
```

Here '`o`' represents function composition. This theorem is a consequence of the first theorem above, but the reverse is not true because the second theorem assumes that the output is lifted *at all times* rather than just at the current instant. For a circuit without loops, this might be adequate, because one would normally assume that inputs are lifted. However if the circuit contained loops (presumably with one or more units of delay incorporated) it would be impossible to prove using theorems of the second form that the output is lifted for all time. It is therefore necessary to use theorems of the first form. The proof then becomes possible — however it is still necessary to perform a nontrivial proof by induction that a circuit has a lifted output at all times.

7.8 Experience with the HOL-ELLA system

The current version of the HOL-ELLA system has abandoned the semantics with lifting, largely because of the greater complexity of the proofs. In fact, it seems from discussions with ELLA's designers that in any case, the semantics of the simulator should not be regarded as a model implementation, and that the new HOL-ELLA semantics may more accurately reflect their original intentions.

Work is still in progress on the new version (with a lifting-free semantics) of the HOL-ELLA system. The pretty-printer and some parts of the translator have not yet been updated to work with the changed definitions of the semantic constants.

Case studies so far undertaken are the parity checker discussed above, a proof of equivalence of two ELLA texts (parallel and recursive formulations of an n-bit adder), and the verification against its functional specification of an elaborate 6-bit counter. The last proof has been performed in both versions of the ELLA semantics, and consequently provides real experience to justify the claims above that the version without lifting makes proofs significantly easier.

Work is currently under way on more complex case studies: proving a floating-point square root chip correct with respect to a specification in terms of real numbers, and proving correctness of a simple microprocessor.

8 Embedding SILAGE

The goal of this section is to motivate and sketch the embedding of SILAGE in HOL so that it can be compared to the embeddings of VHDL and ELLA. All the embeddings confer logical meanings on specific programs, but what sets the HOL-SILAGE approach apart from the others is the emphasis on a formal account of the language definition itself. This section is partly based on material from the definition of the HOL-SILAGE subset [8] and its summary [9].

Section 8.1 introduces transformational design of SILAGE programs, the motivation for the HOL-SILAGE project. The language is introduced in Section 8.2 by way of the parity checker example. Section 8.3 shows how SILAGE definitions can be treated as HOL predicates, and then Section 8.4 sketches how this treatment enables transformation design to be rendered as theorem proving in HOL. Sections 8.5 and 8.6 sketch the formal definition and how it has been mechanized as an ML program respectively. Finally, Section 8.7

summarizes the achievements so far, and points to future goals.

8.1 Transformational design

The design process used in practice at IMEC is to develop a SILAGE program using a software simulator [30], and then to synthesize for a particular architecture — bit-serial or microcoded — with one of the CATHEDRAL synthesis systems [23].

Silicon compilers such as CATHEDRAL can cope mechanically with all the details of mapping a SILAGE program's high-level behavioural description to a low-level implementation, but they cannot currently find an implementation that is best in terms of clock-rate or area. To obtain high performance, designers need to transform their initial SILAGE programs into tuned programs, that express the same behaviour but have a better performance against a particular silicon compiler and architectural requirements. There are several reports of experience of transformational design using SILAGE [22, 32, 35]. At present the only way to check that the initial and tuned programs have the same behaviour is to use simulation.

The goal of the HOL-SILAGE project is to obtain an interactive system that under the guidance of the designer can mechanically prove that the initial and tuned programs have the same behaviour. The idea is to embed SILAGE programs as HOL terms, and represent transformational design as proof of equations in HOL.

A formal definition is essential if SILAGE programs are to be represented by HOL terms, but no formal definition of the full SILAGE language exists,[1] although there is an informal reference manual [17] and there are compiler manuals [27, 30]. Therefore the first task in the project has been to write down a formal definition of a subset of the IMEC dialect of SILAGE.

Given a transformation rendered as a HOL theorem, simulation results from the initial and tuned programs should in principle be the same, so design time is saved because only one simulation is necessary. This claim is made only tentatively, pending practical experience, because it depends on factors beyond the scope of the project, such as whether existing IMEC tools, such as the simulator and synthesis system, correspond correctly to the formal definition. One hopes that future languages will be designed using formal descriptions from the start, so that compilers can be programmed in styles suitable for formal verification.

Other projects on transformational design have constructed systems programmed from scratch [5, 6]. The advantage of building a system for transformational design on top of an LCF-style theorem prover like HOL [10], in which programs are represented by their logical meaning and transformations can only be obtained by logical inference, is that then there is no fear that programming errors can lead to invalid transformations.

8.2 The Parity Checker in SILAGE

SILAGE is not intended to be used at the register transfer level in which the structural description of the parity checker is given, but the circuit makes a fine introduction to

[1]Personal communication with Paul Hilfinger, January 1992.

SILAGE constructs:

```
func PARITY_SIL(in : Bool)out : Bool =
  l1, l2, l3, l4, l5 : Bool
begin
  l1 = !l2;
  out@@1 = false;
  l2 = out@1;
  l3 =if in ⟶ l1 || l2 fi;
  l4@@1 = false;
  l4 = true;
  l5 = l4@1;
  out =if l5 ⟶ l3 || l4 fi;
end;
```

The function PARITY_SIL has one input and one output, signals in and out respectively. The type Bool represents single bits — written true and false. The five signals l1, ..., l5 are declared to be internal to the circuit. The body of the function is a set of definitions for the internal and output signals — the order in which they are listed does not matter.

A *signal* is an infinite stream of *samples*, indexed by time. Time is modelled by the integers — negative times are used only to initialize delays. There are two basic kinds of definition. An *initialization* such as out@@1 = false says that at time -1 the sample on signal out is to be false. An *equation* such as l1 = !l2 says that for all non-negative times, signal l1 equals the pointwise negation (operator !) of signal l2.

The conditional expression if $\mathtt{e}_C$ ⟶ $\mathtt{e}_T$ || $\mathtt{e}_F$ fi is the signal whose samples are drawn from signal $\mathtt{e}_T$ or $\mathtt{e}_F$, depending pointwise on the Boolean signal $\mathtt{e}_C$. The multiplexers from the parity circuit are expressed as conditionals.

Delays are expressed using the notation e@n, which is the signal denoted by expression e, but delayed by n units of time. The two registers in the parity circuit have been expressed as the SILAGE delay expressions out@1 and l4@1, and have been initialized by defining the samples on out and l4 at time -1 to be false.

There are two special kinds of expression in SILAGE, exemplified in the initialization out@@1 = false. A *selector*, such as out, is an expression that can appear on the left-hand side of an equation or initialization; variables are selectors, but constants or delays are not. A *manifest expression*, such as 1 or false, is one that is computable to a constant at compile-time; the time and sample expressions in an initialization must be manifest.

8.3 SILAGE definitions as HOL predicates

Time is identified with the integers, int, and we use the HOL variable t to stand for times. The type of signals with samples of type τ is given by a type abbreviation:

$$\mathit{Signal}(\tau) \overset{\text{def}}{=} \mathsf{int} \to \tau$$

The HOL variables A and B are used for signals. To interpret the parity checker in HOL, constants are needed to interpret SILAGE negation, conditional, and delay — SIGNOT, COND and infix <> respectively:

$$\begin{aligned}(\mathsf{SIGNOT}\ \mathtt{A})\,\mathtt{t} &\overset{\text{def}}{=} \neg(\mathtt{A}\,\mathtt{t})\\ (\mathsf{COND}\ \mathtt{A}_C\,\mathtt{A}_T\,\mathtt{A}_F)\,\mathtt{t} &\overset{\text{def}}{=} (\mathtt{A}_C\,\mathtt{t} \ \rightarrow\ \mathtt{A}_T\,\mathtt{t} \mid \mathtt{A}_F\,\mathtt{t})\\ (\mathtt{A{<>}n})\,\mathtt{t} &\overset{\text{def}}{=} \mathtt{A}(\mathtt{t}-\mathtt{n})\end{aligned}$$

Manifest expressions are turned into signals that have the same sample at all times:

$$\mathsf{VAL}\ \mathtt{x}\,\mathtt{t} \overset{\text{def}}{=} \mathtt{x}$$

A Silage equation asserts that two signals have the same samples for all non-negative times.

$$(\mathtt{A} \sim \mathtt{B})\,\mathtt{t} \overset{\text{def}}{=} \forall \mathtt{t} \geq 0.\ \mathtt{A}(\mathtt{t}) = \mathtt{B}(\mathtt{t})$$

Given these constants, the parity checker is interpreted as a HOL predicate `PARITY_SIL` WIth the following defining equation:

$$\begin{array}{l}\forall \mathtt{in}\,\mathtt{out}.\\ \quad \mathtt{PARITY\_SIL}(\mathtt{in},\mathtt{out}) \overset{\text{def}}{=}\\ \quad (\exists \mathtt{l1}\ \mathtt{l2}\ \mathtt{l3}\ \mathtt{l4}\ \mathtt{l5}.\\ \qquad \mathtt{l1} \sim (\mathtt{!l2}) \wedge\\ \qquad (\mathtt{out}(-1) = \mathtt{F}) \wedge\\ \qquad \mathtt{l2} \sim (\mathtt{out{<>}}1) \wedge\\ \qquad \mathtt{l3} \sim (\mathsf{SIGCOND}\ \mathtt{in}\,\mathtt{l1}\,\mathtt{l2}) \wedge\\ \qquad (\mathtt{l4}(-1) = \mathtt{F}) \wedge\\ \qquad \mathtt{l4} \sim \mathsf{VAL}(\mathtt{T}) \wedge\\ \qquad \mathtt{l5} \sim (\mathtt{l4{<>}}1) \wedge\\ \qquad \mathtt{out} \sim (\mathsf{SIGCOND}\ \mathtt{l5}\,\mathtt{l3}\,\mathtt{l4}))\end{array}$$

The original description [12] of the parity checker consists of a primitive recursive function specifying the desired behaviour, the circuit shown in Section 6, and a proof that the circuit has the behaviour specified by the function. Just as the ELLA implementation can be proved in HOL to meet the original specification (Section 7.5), the analogous property can be proved of the Silage implementation. The proof is routine, and makes use of equational laws about the delay operator, `<>`.

8.4 Transformation in HOL

Transformational design will be illustrated with a digital filter example suggested by Hans Samsom of IMEC.

```
func F₁(x : Nat)y : Nat =
begin
   y@@1 = 0;
   y@@2 = 0;
   y = x + (w1 × y@1) + (w2 × y@2)
end;
```

Each output sample on y is computed by adding to the input sample on x the previous two outputs, weighted by constants w1 and w2. Let us assume the multiplications can be cheaply computed by shifting and then the bottleneck in the circuit is the need to compute two additions in series during each cycle.

This circuit can be tuned by defining a new signal, z, to be the sum of the present and previous outputs. The two additions can then be computed in parallel:

```
func F₂(x : Nat)y : Nat =
  z : Nat;
begin
  z@@1 = 0;
  z = (w1 × y) + (w2 × y@1);
  y@@1 = 0;
  y@@2 = 0;
  y = x + z@1;
end;
```

It is straightforward to prove these two functions to be equivalent — the theorem

$$\vdash \mathsf{F}_1 = \mathsf{F}_2$$

has been proved in HOL, where F_1 and F_2 are constants standing for the interpretation in HOL of these two function definitions.

The point of this example is to show how a transformation can be formalized as a HOL theorem. The example shows simple verification of a given transformation — a future goal of the project is to support experimental discovery of a tuned program, by incremental transformation from an initial program.

8.5 The formal definition

The method of Structural Operational Semantics (SOS) [28, 31] is used to formalize the type-checking and translation processes for Silage. Existing descriptions of Silage [17, 27, 30] make the concrete syntax precise using BNF rules, but leave semantic issues such as type-checking or executability informal and partly vague.

The definition consists of three parts: BNF rules determine the *abstract syntax*; SOS *static semantics* rules determine the well-typed and well-formed expressions and definitions; *denotational semantics* rules map definitions into HOL terms. Existing Silage texts define the concrete syntax precisely, so it has not been redefined.

The definition will be illustrated by a discussion of the semantic treatment of initializations whose abstract syntax takes the form $\mathsf{e}_1@@\mathsf{e}_2 = \mathsf{e}_3$, where each e_i is an expression. Here is the static semantics rule:

$$\frac{\vdash \mathsf{e}_1 \;\Rightarrow\; \sigma, \tilde{a}_1\ (\mathit{Sel} \in \tilde{a}_1) \quad \vdash \mathsf{e}_2 \;\Rightarrow\; \mathtt{Nat}, \tilde{a}_2\ (\mathit{Man} \in \tilde{a}_2) \quad \vdash \mathsf{e}_3 \;\Rightarrow\; \sigma, \tilde{a}_3\ (\mathit{Man} \in \tilde{a}_3)}{\vdash \mathsf{e}_1@@\mathsf{e}_2 = \mathsf{e}_3\,\mathrm{ok}}$$

The predicate on the bottom says that the initialization is well-formed, and the rule says that this is so just when the three predicates on the top line hold, each of the form $\vdash \mathsf{e} \;\Rightarrow\; \sigma, \tilde{a}$, pronounced 'expression e has type σ and attribute list $\tilde{a}$'. An attribute a

records information about an expression; for instance, the attributes *Man* and *Sel* indicate whether an expression is manifest or a selector respectively. The rule above requires that expressions e_1 and e_3 have the same type, σ, that e_2 is numeric, that e_1 is a selector, and that e_3 and e_2 are manifest expressions. The SILAGE texts attempt to use BNF rules to assert type and well-formedness conditions; the SOS rules used here are more lucid and precise.

Here is the denotational semantics:

$$\boxed{[\![\mathsf{e}_1@@\mathsf{e}_2 = \mathsf{e}_3]\!] \overset{\text{def}}{=} [\![\mathsf{e}_1]\!](-[\![\mathsf{e}_2]\!]^{\mathcal{M}}) = [\![\mathsf{e}_3]\!]^{\mathcal{M}}}$$

If ϕ is a phrase of abstract syntax, $[\![\phi]\!]$ is its translation into the HOL logic. Any well-formed expression, such as e_1, can be translated to a signal, written $[\![\mathsf{e}_1]\!]$. A manifest expression, such as e_2, can also be interpreted as a scalar in HOL, written $[\![\mathsf{e}_2]\!]^{\mathcal{M}}$. The predicate requires that the signal denoted by e_1 have the sample denoted by e_3 at negative time e_2.

One can prove properties of the definition itself — a benefit of formalizing the translation process itself. For instance:

Theorem. *If expression* e *has type* σ*, its translation* $[\![\mathsf{e}]\!]$ *is a* HOL *term of type* $\mathit{Signal}[\![\sigma]\!]$*, where* $[\![\sigma]\!]$ *is the translation of type* σ *as a* HOL *type.*

Such a metatheorem about the language definition can be proved by hand, which is feasible given the simplicity of SILAGE, but tedious.

8.6 Mechanization of the definition

The formal definition discussed in the previous section exists as a mathematical definition on paper. It is good for the definition to be independent of any machine implementation, for then it is suitable for scrutiny by interested parties, such as compiler writers or users, to determine whether it matches tools already in existence, or even to serve as a precise reference standard.

The prototype HOL-SILAGE system is a shallow embedding of the formal definition in HOL. SILAGE syntax is embedded as ML types, and the static and denotational semantics rules are embedded as ML functions. The top-level of the system allows SILAGE function definitions to be interpreted mechanically as HOL constant definitions.

A deep embedding of SILAGE, in which the static and denotational semantics are represented as HOL operations, would enable metatheorems such as the one at the end of the previous section to be proved mechanically. Such an embedding has not been attempted because shallow embedding is sufficient to support transformational design, and is simpler than a deep embedding.

The mechanization is partly within ML and partly within the logic, and hence is not immediately amenable to mechanical proof. Even so, the existence of an independent definition helps in three ways to increase our confidence in the HOL-SILAGE system.

First, the likelihood of error has been reduced by structuring the program in exactly the same way as the definition. For instance, there is a polymorphic ML type α `Exp`, such that a value of the type represents the abstract syntax tree (AST) of a SILAGE expression, whose nodes are labelled with values of type α. As a matter of notation, if e is a SILAGE expression, let $\underline{\mathsf{e}}$, of type `void Exp`, be the AST corresponding to e tagged with no information.[2] Corresponding exactly to the static and denotational semantics are the following two ML functions:

$$\boxed{\begin{array}{c}\texttt{checkExp} : \texttt{void Exp} \rightarrow (\texttt{Type} \times \texttt{Attrib list})\,\texttt{Exp} \\ \texttt{sigExp} : (\texttt{Type} \times \texttt{Attrib list})\,\texttt{Exp} \rightarrow \texttt{term}\end{array}}$$

where `Type`, `Attrib` and `term` are the ML types of SILAGE types, SILAGE attributes and HOL terms respectively.

Second, the formal definition can be used to document precisely the behaviour of the mechanization. For instance, the two functions above can be specified as follows:

- If expression e is well-typed, evaluation of checkExp($\underline{\texttt{e}}$) returns an AST labelled with the type and attribute information derivable for e.
- If expression e is not well-typed, evaluation of checkExp($\underline{\texttt{e}}$) fails by raising an exception.
- If expression e is well-typed, evaluation of sigExp(checkExp($\underline{\texttt{e}}$)) returns the HOL term $[\![\texttt{e}]\!]$.

Third, properties proved by hand about the definition carry over to the mechanization. For instance, the theorem mentioned at the end of Section 8.5 assures us that the term $[\![\texttt{e}]\!]$, constructed in the HOL system by `sigExp`, will always be well-typed.

8.7 Achievements and goals

The Cambridge part of the HOL-SILAGE project has produced a formal definition of a substantial subset of the IMEC one, and mechanized it within the HOL system. Several examples drawn from a previous study of SILAGE transformations [22] have been proved correct in HOL. The formal definition is a basis for work in progress at IMEC, by Catia Angelo and Hans Samsom, on cataloguing the SILAGE transformations that arise in practice, and animating transformations using a windowing system. The goal is a system based on HOL for conducting transformations, but needing little knowledge of the underlying theorem prover, so as to be suitable for users whose primary interest is circuit design.

9 Embedding VHDL

This section starts with an overview of the semantics developed for VHDL. This is work in progress, and as such is not capable of the advanced reasoning displayed in the overviews of the HOL-ELLA and HOL-SILAGE systems just presented.

The following definitions will be used in discussing VHDL and the way it is embedded in HOL:

Event: A change in the value of a signal.

[2]The ML type `void` contains just one value and therefore no information.

Transaction: The value that a signal should take on at a particular time in the future. A transaction may (or may not) be converted into an event at that time.

Point of computation: The point at which a particular collection of transactions is processed.

Process: A VHDL concurrent statement equipped with a set of signal names, or sensitivity list, guarding its activation.

Simulation cycle: The evaluation of all the processes in a program at a particular point of computation.

The simulation loop of VHDL can be broken down into two distinct phases after the initialization step has been completed. These are the top-level simulation loop and a simulation cycle. The simulation loop is responsible for moving the simulation forward to the next collection of transactions to process, while a simulation cycle is performed each time around the loop to schedule transactions for the future. Only the flavour of the semantics is discussed here. The interested reader is directed to [34] for a more complete exposition.

The semantics has been written as a collection of transition relations, in the style of [31], which describe the simulation algorithm and the interaction of various VHDL statements with that algorithm. The emphasis has been on a single simulation cycle, and work is now under way to describe the top-level simulation loop, of which a simulation cycle is a constituent part. The following box contains 'pseudo-code' describing the top-level simulation loop:

```
while transactions remain to be processed
    go to nearest point of computation with transactions to process
    update the state from the current transactions
    determine which updates represent events
    perform a simulation cycle based on new state and events
end while
```

The first step is to move forward to the nearest interesting point of computation (i.e., one where there are transactions to process), and set the current time to be the physical time unit associated with that point of computation. The state of the signal values is then updated by those that they are supposed to take on during the present point of computation. Next the signals for which this update represents a change in value, or event, are determined. A simulation cycle is then performed based on the new state of the world. The cycle is then repeated until there are no more transactions to process.

Note that whilst the phrase 'point of computation' can often be understood to mean a time slice, a point of computation is more than that within the context of VHDL's particular simulation model. One could, for instance, say that the points of computation P_1 through P_n represent the time units 1 through n. Alternatively, they could represent 1 through n δ-delays between two major time units.

The concept of δ-delay is the way in which VHDL deals with 0-delay signal assignments. During each iteration through the simulation loop a static state of the world is used and transactions are scheduled to be performed at some future time point. This scheduling applies to any transaction, whether it is to occur at a delay offset zero units from now, or at some larger one. The delay around the simulation loop in zero time is called a δ-delay. An individual δ does *not*, therefore, represent a quantifiable unit of time. Rather it is simply a simulator artifact.

An individual simulation cycle can be summarized as:

```
determine which processes are active based on current events
run the active processes in parallel
join all activated process' scheduled transactions into a collective whole
```

The cycle starts with the activation of those processes that are sensitive to any of the current events. These active processes are then run in parallel. During their execution, each process schedules transactions to be processed in the future by the evaluation of sequential statements. When all the processes have terminated, the separate futures are gathered into an amalgamated view of future behaviour.

The semantics of small subset of VHDL called Femto-VHDL is embedded in HOL using a new tool developed by Tom Melham [26]. It includes concurrent process statements, simple sequential statements and general Boolean expressions. The syntax is described in the logic as a recursive type [25]. Process statements are the outermost level, and contain sequential statements which may make use of Boolean expressions. Given this rather terse summary of syntax, our running example would take on the following form in Femto-VHDL. It should be noted that the design has been somewhat simplified. The register and inverter in series between `out` and `l1` have been removed to be replaced by a single unit-delay inverter, doing away with signal `l2`. If this were not done, the device would have caused VHDL simulators to loop in δ-time after 1 nanosecond had elapsed.

```
process (inp,l1,outp)
begin
    if inp then l3 <= transport l1 after 0 ns;
    else l3 <= transport outp after 0 ns;
    end if;
end process;

process (l5,l3,l4)
begin
    if l5 then outp <= transport l3 after 0 ns;
    else outp <= transport l4 after 0 ns;
    end if;
end process;

process (outp) begin l1 <= transport (not outp) after 1 ns; end process;

process (l4) begin l5 <= transport l4 after 1 ns; end process;
```

Each of the `process` statements should be viewed as running concurrently with the others if it is activated by an event on its sensitivity list (e.g., `(inp,l1,out)` in the first process).

The signals `in` and `out` have been renamed to `inp` and `outp` to avoid clashes with VHDL reserved words.

As mentioned earlier, the responsibility of the simulation cycle is to schedule transactions to be processed in the future. This scheduling can be demonstrated by 'running' the example in a symbolic fashion through the simulation cycle algorithm (using some special purpose theorem proving tools). A starting state for the simulation cycle is given; this must include the current time, the values of the signals, the signals for which there are events and the currently scheduled transactions to the system. Then mechanized proof can be used to deduce in HOL what transactions result from the execution of a simulation cycle. So, if the environment is represented as a triple in HOL:

```
        env =
          "(now,{(`l1`,l1),(`l3`,l3),(`l4`,T),(`l5`,T),(`inp`,inp),
               (`outp`,outp)},{`inp`})"
```

where the first element is a variable representing the current time, the second is a set of signal value pairs representing the current state of the signals (all the signals except `` `l4` `` and `` `l5` `` have variables as values), and the third is a set containing the names of the signals for which there is currently an event. We may execute the example in HOL given some initial transactions `trans`, and ask what new transactions `trans'` this environment would cause to be scheduled.

The result is the following statement in HOL after simplification with facts about the semantics:

```
              trans' = (inp → (POST `l3` l1 trans now)
                            | (POST `l3` outp trans now))
                    ZIP trans
```

`POST` is a function which given a signal name, the value that signal is supposed to take on, some initial transactions and the time at which the assignment is supposed to take place will pre-emptively schedule the signal in question to take on the specified value at the specified time by adding it to the running transactions. Pre-emption means that any transactions previously scheduled on the signal at times after the one in question are removed when the current transaction is added.

`ZIP` is used to collect the transactions scheduled by each active process into a collective view of future activity. It does this by performing a simple set union at any given point of computation on the sets of signal-value pairs that it finds there. In our example, we see that only one process is actually activated, as all the other processes returned `trans` unchanged.

The result is as expected, namely an event on `` `inp` `` will cause `` `l3` `` to take on the value `l1` (the inverted value of `` `outp` `` 1 nanosecond ago) if the value `inp` is a change from false to true. Otherwise it takes on the value `outp`. Note that the toggling of `` `inp` `` does not immediately cause a value to be output on `` `outp` ``. One would have to go through another iteration of the simulation loop in δ-time to see that result. This can be demonstrated by changing our starting environment `env` to be:

```
        env =
          "(now,{(‘l1‘,l1),(‘l3‘,l3),(‘l4‘,T),(‘l5‘,T),(‘inp‘,inp),
               (‘outp‘,outp)},{‘l3‘})"
```

meaning that the toggling of `‘inp‘` represented a change from false to true.

‘Running’ the example yields the result (again after simplification):

```
                  trans’ = (POST ‘outp‘ l3 trans now) ZIP trans
```

Again, this is as expected. The signal `‘outp‘` takes on the value `l3` one δ-step from now.

It is unfortunate that iterating through the simulation loop must currently be done by hand. This will change in the near future, and symbolic execution of Femto-VHDL programs in HOL will allow the comparison of two texts purporting to be the same behaviour given properties of equivalence currently under development.

10 Conclusions

Semantics for subsets of three widely differing hardware description languages have been described. Each formal semantics is represented in higher order logic and supported mechanically by the HOL theorem-proving system. The three studies suggest that higher order logic is sufficiently expressive to conveniently represent the semantics of a wide variety of hardware description languages, and that the HOL system is flexible enough to support different project aims. The three studies cover both operational and denotational styles of semantics.

The following table is a rough comparison of the intended applications of the three projects. Each embedding is marked as deep or shallow, that is, whether the syntax of the language is embedded as a HOL or ML type respectively. The columns list three applications. The primary applications for each project are marked with a $\bullet$ symbol, and secondary applications are marked with a $\circ$ symbol. The $*$ indicates that proving language properties is an aim of the HOL-SILAGE project, but the proofs are not intended to be done mechanically in HOL.

	Language Properties	Program Verification	Program Transformation
ELLA (shallow)		●	○
SILAGE (shallow)	*	○	●
VHDL (deep)	●	●	○

In all three systems, a mechanism to map between the concrete syntax of an HDL program and its logical representation is important, but any program to do so is necessarily a non-secure piece of software in the sense that it cannot be directly verified by theorem proving in HOL. In the HOL-ELLA and HOL-VHDL systems, the risk of errors in the parser or pretty-printer is minimized by generating them automatically using tools in the HOL system. The input for the parser generator is a specification of the context-free syntax of the language and a set of action symbols for building the parse tree. The pretty-printer generator has a pretty-printing meta-language in which the layout can be described. The pretty-printer provides a means of viewing a HOL term representing an HDL text in the concrete syntax of the HDL. This is useful, as it gives feedback to the user on whether the parsing and semantic translation have produced something that the user expects. In the HOL-ELLA system, semantic constants correspond approximately to non-terminals in the syntax of ELLA, and hence it is easy to pretty-print HOL terms representing ELLA text as the ELLA text. At present, the HOL-SILAGE system has a parser programmed in ML and structured exactly according to the formal definition of SILAGE, but it has no pretty-printer. For the examples dealt with so far, this has been no great loss because there is such a simple correspondence between a SILAGE program and its logical interpretation. Work is in progress at IMEC to link the HOL system with the Cornell Synthesizer Generator. The latter would provide parsing and pretty-printing, as well as managing the application of SILAGE transformations.

There are some basic issues about semantics which affect the 'correctness' of the formal embedding. Having defined a semantics for a language, one must consider whether the semantics reflects the intention of the language designers. This is difficult to judge if no semantics were defined previously. Where another definition of semantics exists one may need to verify a correspondence between the semantics. With conventional notations, there is also the problem that a user may assume a certain meaning from the syntax irrespective of the formal semantics. One reason why the formal semantics of SILAGE was written up as a mathematical document, independent of the machine implementation, was so it could be shown to SILAGE experts to question whether the *post hoc* semantics reflected what had been implemented in existing tools.

The projects described here illustrate some of the benefits of designing a language with a formal semantics. Quite subtle 'inconsistencies' in the informal semantics of a language (as defined by the compiler/simulator) can have very significant effects on the formal semantics. An example of this is the behaviour of the ELLA `CASE` statement with respect to the unknown value (Section 7.7). A formal semantics can provide a precise specification from which the implementors of compilers and simulators for the language can work.

Finally, it should not be forgotten that hardware can be directly described in logic without using any HDL and this representation is likely to be the easiest one to reason about. An important research topic, dual to the approach taken here, is to develop CAD tools such as simulators and synthesizers that act directly on logic representations and thereby avoid the need for the kind of effort described in this paper [2, 18, 21]. In the meantime it is essential for HDLs to be modelled in logic if theorem-proving is to be applicable to hardware design.

Acknowledgements

Richard Boulton, John Harrison and John Herbert were supported by a SERC/DTI grant *HOL Verification of ELLA Designs* (IED project 1129). Andrew Gordon was supported by the CHEOPS project: *Higher Order Logic supported design for Complex Data Processing Systems*, (Esprit Basic Research Action 3215 managed by IMEC: Inter-university Micro-Electronics Centre, Belgium). John Van Tassel was supported by grant number AFOSR-91-0246 from the United States Air Force Office of Scientific Research entitled

Verification of VHDL Designs. All three projects make use of theorem proving tools written by Dr Tom Melham.

References

[1] R. Boulton, M. Gordon, J. Herbert, and J. Van Tassel, 'The HOL Verification of ELLA Designs', in *Proceedings of the International Workshop on Formal Methods in VLSI Design*, Miami, January 1991.

[2] A. J. Camilleri, 'Simulating Hardware Specifications within a Theorem-Proving Framework', *International Journal of Computer Aided VLSI Design*, 2 (3), pp. 315–337, 1990.

[3] A. Church, 'A Formulation of the Simple Theory of Types', *The Journal of Symbolic Logic*, Vol. 5 (1940), pp. 56–68.

[4] Computer General Electronic Design, 'The ELLA Language Reference Manual, Issue 4.0', 5 Greenways Business Park, Bellinger Close, Chippenham, Wiltshire, SN15 1BN, England, 1989.

[5] J. Darlington, P. Harrison, H. Khoshnevisan, L. McLoughlin, N. Perry, H. Pull, M. Reeve, K. Sephton, L. While, S. Wright. 'A Functional Programming Environment Supporting Execution, Partial Evaluation and Transformation', in *PARLE 89 Parallel Architectures and Languages Europe*, volume 365(1) of *Lecture Notes in Computer Science*, Springer-Verlag, 1989.

[6] G. Durrieu, K. Kessaci, and M. Lemaître. 'Transe: An Experimental Transformation Assistant Software for Digital Circuit Design', in R. Sharp and J. Staunstrup, editors, *Workshop on Designing Correct Circuits*, Lyngby, January 1992.

[7] D. Genin, P. Hilfinger, J. Rabaey, C. Scheers, and H. de Man. 'DSP Specification using the Silage language', in *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 1057–1060, April 1990.

[8] A. D. Gordon, 'A Mechanised Definition of Silage in HOL', Internal Report, ESPRIT BRA 3215, Computer Laboratory, University of Cambridge, 1992.

[9] A. D. Gordon, 'The Formal Definition of a Synchronous Hardware-Description Language in Higher Order Logic', Submitted for publication, January 1992.

[10] M. J. Gordon, A. J. Milner, and C. P. Wadsworth, 'Edinburgh LCF: A Mechanised Logic of Computation', volume 78 of *Lecture Notes in Computer Science*, Springer-Verlag, 1979.

[11] M. J. C. Gordon, 'Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware', in *Formal Aspects of VLSI Design*, edited by G. J. Milne and P. A. Subrahmanyam, North Holland, 1986.

[12] M. J. C. Gordon, 'HOL: A Proof Generating System for Higher-Order Logic', in *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P. A. Subrahmanyam, Kluwer, 1988.

[13] M. J. C. Gordon, '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.

[14] F. K. Hanna and N. Daeche, 'Specification and Verification using Higher-Order Logic: A Case Study', in *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, edited by G. J. Milne and P. A. Subrahmanyam, North-Holland, 1986, pp. 179–213.

[15] F. K. Hanna and N. Daeche, 'Dependent Types and Formal Synthesis', in *Mechanized Reasoning and Hardware Design*, edited by C. A. R. Hoare and M. J. C. Gordon, Prentice-Hall International Series in Computer Science, Prentice-Hall, 1992.

[16] P. N. Hilfinger, 'Silage, a High-Level Language and Silicon Compiler for Digital Signal Processing', in *IEEE Custom Integrated Circuits Conference CICC–85*, pp. 213–216, Portland, Oregon, May 1985.

[17] P. N. Hilfinger, 'Silage Reference Manual', Computer Science Division, University of California, Berkeley, December 1987.

[18] W. A. Hunt, Jr., 'The Mechanical Verification of a Microprocessor Design', in *From HDL Descriptions to Guaranteed Correct Circuit Designs*, edited by D. Borrione, North Holland, 1987.

[19] W. A. Hunt, Jr. and B. C. Brock, 'A Formal HDL and its Use in the FM9001 Verification', in *Mechanized Reasoning and Hardware Design*, edited by C. A. R. Hoare and M. J. C. Gordon, Prentice-Hall International Series in Computer Science, Prentice-Hall, 1992.

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

[21] M. Leeser, 'Using Nuprl for the Verification and Synthesis of Hardware', in *Mechanized Reasoning and Hardware Design*, edited by C. A. R. Hoare and M. J. C. Gordon, Prentice-Hall International Series in Computer Science, Prentice-Hall, 1992.

[22] P. Lippens, 'Defining Control Flow from an Applicative Specification', Internal Report, Philips Research Laboratories, Eindhoven, December 22, 1988.

[23] H. de Man, J. Rabaey, P. Six, and L. Claesen, 'Cathedral–II: A Silicon Compiler for Digital Signal Processing', *IEEE Design and Test*, 3(6):73–85, December 1986.

[24] T. F. Melham, 'Using Recursive Types to Reason about Hardware in Higher Order Logic', in *The Fusion of Hardware Design and Verification: Proceedings of the IFIP WG 10.2 Working Conference, Glasgow, July 1988*, edited by G. J. Milne, North-Holland, 1988, pp. 27–50.

[25] T. F. Melham, ‘Automating Recursive Type Definitions in Higher-Order Logic’, in *Current Trends in Hardware Verification and Automated Deduction*, edited by G. Birtwistle and P. A. Subrahmanyam, Springer-Verlag, 1988.

[26] T. F. Melham, ‘A Package for Inductive Relation Definitions in HOL’, in *Proceedings of 1991 International Workshop on the HOL Theorem Proving System and its Applications*, IEEE Computer Society Press, 1991.

[27] Mentor Graphics Corporation, Wilsonville, Oregon, ‘DSParchitect DFL User’s and Reference Manual’, December 1991.

[28] R. Milner, M. Tofte, and R. Harper, ‘The Definition of Standard ML’, MIT Press, Cambridge, Mass., 1990.

[29] J. D. Morison and M. G. Hill, ‘A Formal Definition of the Static Semantics of ELLA’s Core’, Report No 91024 Royal Signals and Radar Establishment, August 1991.

[30] L. Nachtergaele, ‘User Manual for the S2C Silage to C Compiler’, IMEC, Leuven, March 22, 1990.

[31] G. Plotkin, ‘A Structural Approach to Operational Semantics’, Technical Report DAIMI FN-19, Computer Science Dept., Aarhus Univ., September 1981.

[32] J. G. Samsom, L. J. M. Claesen, and H. J. de Man, ‘Correctness Preserving Transformations on the Hough Algorithm’, in *CompEuro 92, The Hague*, May 4–8, 1992.

[33] V. Stavridou, S. M. Eker, and S. Aloneftis, ‘FUNNEL: A CHDL with Formal Semantics’, preprint 1991.

[34] J. P. Van Tassel, ‘A Formalisation of the VHDL Simulation Cycle’, Technical Report 249, University of Cambridge Computer Laboratory, March 1992.

[35] I. Verbauwhede, ‘VLSI Design Methodologies for Application-Specific Cryptographic and Algebraic Systems’, PhD Thesis, Catholic University, Leuven, July 1991.