Embedding Hardware Design and Description Languages in Proof Systems

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

December 12, 1989

Contents

1 Introduction

This section will give an overview of computer hardware description languages (CHDLs), proof systems (logical frameworks) and the way in which they may interact.

1.1 Computer hardware description languages

The spectrum of CHDLs ranges over a number of different levels of description, both structural and behavioural. Descriptions of a circuit can be given at the floor planning, sticks, register transfer or block diagram level; alternatively we can use a behavioural or procedural description.

Here we are mainly concerned with high level register transfer level and upwards CHDLs. The examples and motivations used in this document are mostly drawn from high level CHDLs; though much of this discussion has relevance to the lower levels.

The distinction between hardware description languages and hardware design languages is becoming blurred, the result is CHDDLs (computer hardware design and description languages). The aim is to allow the same language to be used throughout the design process, provided the language is sufficiently expressive to describe the both the specification and the low-level detail. Two such languages are ELLA [Pra86a, Pra86b, MPT85] and VHDL [Ins88, SLM$^+$85, Mey89].

Even with the help of a high level structured language substantial circuits are becoming harder to design and verify. Traditionally, after circuit design a prototype was built and thoroughly tested; as errors were found the design was modified and the process iterated. Nowadays it would take too long to produce a prototype and the cost of products would be prohibitively high. Thus simulations of the circuit are used. This involves a simulator for the CHDDL in which the circuit description has been written. One of the main problems with simulations is that, for a circuit of substantial size it becomes unfeasible to do exhaustive simulations.

For example, if we were to exhaustively simulate a 32 bit adder we would have to consider 2^{64} input combinations. Even if we checked one test vector per nano second it would take us more than half a year! In circuits that have some sort of internal state the order in which the input values are supplied will often matter, probably increasing the number of test vectors we have to consider.

There are, however, symbolic simulators, which allow algebraic manipulation of inputs which may reduce the number of test vectors.

1.2 Hardware verification

Formal verification of a design involves specifying the circuit and the properties it must satisfy in some formal system (see below) and then proving the circuit description has the desired properties. For example we show that the circuit satisfies its formal specification, or that a certain relation always holds between points of interest in the circuit.

Ideally we would like the formal verification of a design to go hand in hand with the actual design process so that they may support each other.

For example, in the proof of a simple counter (discussed below), the specification is used to obtain a simple finite state automaton which in turn suggests a hardware design mimicking the two phases present in the automaton. Conversely, the circuit design will often suggest a line of proof. In one design of Mike Gordon's computer (detailed below) a ROM is used to dictate sequence of micro instructions implementing the user visible instructions. The proof of the computer shows the effects of each micro instruction on the internal state and show that the particular sequences indicated by the ROM have the effect that the (user level) instructions are supposed to have.

Thus we would like to be able to change from CHDDL description to the proof system description. It should really be done automatically while preserving the intended structure of the circuit. Appendix B contains an outline of a simple *Behaviour* function, which maps from a circuit to a formula of a proof system characterising its behaviour. Even if the change over CHDDL to proof system is not automatic it would still be valuable, breaking the design and the proof into a number of smaller manageable steps. We will return to these considerations in section 3 (Pragmatic aspects).

I will now discuss some of the more well known verification efforts.

1.2.1 Sobel image processing chip

The proof of the Sobel image processing chip by Narendran [Nar88] was performed after the chip had been taken into production. Narendran claims that this is the first time a chip, not designed specifically with verification in mind, has been proved to implement its specification. The method used here is quite different from the other verification efforts listed below, mainly in its use of rewrite rules rather than logic reasoning.

The description of the chip was given in VHDL, which was manually translated to a formal description of the implementation. A formal specification was written from an informal specification. The correctness proof used the Kapur-Narendran method. The Kapur-Narendran method is a refutational approach: one assumes that the specification is not met and tries to derive a contradiction. The set of formulae representing the negated statement of correctness is skolemised to remove existential quantifiers. This new set is then translated into an equivalent set of polynomials over a boolean ring with operators XOR and AND. (The atoms of the formulae become the indeterminates of the polynomials.) Now, the original set of formulae is satisfyable if and only if the set of polynomials has a solution. This method has been implemented in the Rewrite Rule Laboratory (RRL).

1.2.2 FM8501

FM8501 is a sixteen bit micro processor designed by Hunt [Hun85]. He proved that a high level (block diagram) description correctly implemented the specification of the processor. From this high level implementation a lower level (register transfer level) implementation was derived, by expanding the definitions of the building blocks (e.g. adders) to their basic components (e.g. AND gates). The resulting circuit was optimised by identifying common subterms.

The specification of FM8501 was in the form of an instruction interpreter, with the visible registers as arguments. The interpreter was self recursive to model successive discrete time steps. The same method was used to describe the implementation. The correctness proof showed that one step of the specification interpreter corresponds to a number of steps of the "big-machine" (the implementation). This is, as we shall see, a common method. The Boyer-Moore theorem prover, the proof system used by Hunt, is a first order logic with induction. In first order logic this relation (of temporal abstraction) between two levels of description cannot be expressed. Thus Hunt uses oracles that predict the exact number of low level time units that correspond to one time step at the higher specification level. The big-machine was systematically expanded to produce a combinatorial logic implementation with feedback loops broken by delays (representing previous state information).

1.2.3 Verify

VERIFY is a hardware verification program written by Barrow [Bar84]. The system is written in Prolog, and it uses first order logic, to which a number of heuristics to deal with hardware verification have been added. It has been used to prove some fairly substantial circuits such as the D74 circuit involving some 18,000 transistors and an implementation of Gordon's computer (see below).

VERIFY proves correctness of hierarchies by recursively proving the the subcomponents correct. Identical component re-use proofs. Circuits are described as modules and each module is a considered to be a finite state machine, described by equations (rather than an exhaustive table):

Ports A list of input and output connections.

StateEqns State equations describe the internal state variables that a module may or may not have.

OutputEqns Output equations indicate the output variables' values.

Connected This part describes the connectivity of the subcomponents of a module.

StateMap This part allows one to identity state variables, for example.

The structural description is transformed by VERIFY to a behaviour description, which is proved to be equivalent to the specification. For this algebraic methods are used. VERIFY does not handle structural and behavioural homomorphisms in general, but did deal with the temporal abstraction (a behavioural homomorphism) of Gordon's microcomputer, for example.

Barrow includes two proofs in his paper. The first one is a straightforward proof of the D74 circuit. The second proof describes an implementation of Gordon's computer. The equivalence of a micro coded control and the high level specification (written in terms of the effect of instructions on the state) was to be shown. Rather than use "real" induction to deal with the equivalence of micro instruction and instruction levels Barrow unrols the definitions of the circuit repeatedly. Accumulating the effects of the micro instructions allows him to show that the sequences of these correctly implement the (user level) instruction. He cannot describe this process within his system, so that his proof is not as neat as the one described below.

1.2.4 Gordon's micro processor

Mike Gordon formally specified a simple micro computer which has been given various implementations [JBG86, Joy87, Bar84]. One of these has even been implemented in silicon. The computer is simple, yet it is interesting to verify because of its features, such as a bus, ROM, RAM and microcoded control. Implementations have been proved correct using LCF_LSM[1] [Gor] and and HOL, which is a descendant of LCF_LSM. The latter verification effort, which is described below, is essentially an updated version of the former.

The computer was proved correct with respect to its specification in two ways. The first proof resembles Barrow's proof. The second proof is a more sophisticated version which VERIFY could not handle because it uses inherently higher order reasoning.

The first proof proceeded as follows. All abbreviations (of subcomponents etc.) were expanded. Then the lookups of the ROM were replaced by the actual values at the ROM locations. The proof then proceeded with the most important step of considering all the different execution paths of the processor. At the start of an instruction, at time $t1$ say, we have a certain set of input signals. From this we can derive the state of the computer at time $t1 + 1$ by "symbolically executing" (or simulating) the operation of the computer by expanding its definitions at the micro instruction level. After a number of time steps at the micro instruction level the instruction finishes at time $t2$ say. At both times $t1$ and $t2$ a signal *ready* is true, signifying that that the computer can start another instruction. Now the cumulative effect of the sequence of micro instructions is rewritten in terms of its effect on the state at time $t1$, which can then be compared to the specified effect at the instruction level. This procedure is repeated for all different instructions. Note that $t2$ may differ from instruction to instruction: not all instructions need take the same number of iterations at the micro instruction level. All the states at the times $t2$ have in common that the signal *ready* is true. Using this, the this result was proved equivalent to the specification by case analysis on the values of input signals and on the possible values of the opcode (i.e. different instructions).

The second proof is similar to the first but is not written directly in terms of the start and end times of the instructions. Instead, a temporal abstraction mapping is used. As mentioned above, this function is inherently higher order and could therefore not be used for the proofs of Gordon's computer in VERIFY. The temporal abstraction function relates the microcode time scale (more concrete) to the instruction set time scale (more abstract).

$$\begin{array}{ll} abs\ p\ 0\ t = & p\ t \wedge \forall t'.t' < t \wedge \neg p(t') \\ abs\ p\ (n+1)\ t = & p\ t \wedge \exists t'.t' < t \wedge abs\ p\ n\ t' \wedge \\ & \forall t''.(t' < t'' \wedge t'' < t) \rightarrow \neg p(t'') \end{array}$$

This function has variously been called *abs* in [JBG86], *istimeof* in [Mel87] and *nexttime* in [Coh87]. All instruction set time signals were now defined in terms of the microcode time scale signals using *abs*. The signal *ready* was used as the synchronisation condition:

$$\forall n. sig\_abs\ n\ t = signal(abs\ ready\ n\ t)$$

This means that the abstract (instruction level) signal at time n corresponds to the micro instruction time t such that at t *ready* is true for the nth time. The actual proof was

[1] LCF_LSM is a Cambridge variant for hardware verification of LCF, Milner's logic of computable functions [Pau85].

now concluded by the well-definedness of *abs* and the first proof. The issue of temporal abstraction is discussed in more detail in section 3.1.

1.2.5 A simple counter

A mechanised proof of a simple counter [CG86] is a small part of the verification effort of the VIPER micro processor. The VIPER effort as a whole will be discussed below. Although the counter is a very small circuit, it shows on a manageable scale, most of the techniques used for the verification of VIPER.

The counter is specified in decreasing levels of abstraction:

- The top level specification. It is a functional specification: the behaviour of the counter is specified by its effect on the state.
- The specification of the so-called host machine is the next level. It is a description of a finite state automaton.
- The high level design is the next level of specification. Here a structural description of the circuit is given in terms of the behaviour of its components and how their (functional) behaviours interact.
- The lowest level of description is the description of the circuit in terms of basic combinatorial gates (INV, AND etc.) and feedback loops broken by delays.

Correctness is proved by showing the equivalence of the first two levels, the second and third level and, the equivalence of the two lowest levels. By the transitivity of equivalence the circuit description is equivalent to the top level specification.

The proof of correctness of the host machine takes a form very similar to that of Gordon's computer in [JBG86]. The statement of correctness has the following underlying reasoning behind it: The state of the counter at time $t+1$ is derived from the top level specification of the counter applied to the state at time t. The state at $t+1$ must be equal to the state arrived at by computing it using the host machine. However, the host machine has its own time scale, in which it may take more than one time step to arrive at the result. The condition on which the two time scales synchronise is the "initial node" condition.[2] If the automaton is at its initial node it is in a position to start a new instruction. The correctness theorem thus states that:

> **if** at time t the host machine is at its initial node **then** there exists a time t' after time t at which the host machine is at its initial node for the first time after t (termination condition) **and** at time t' the state arrived at by the host machine by repeatedly applying its definition, agrees with the state resulting from applying the top level definition of the counter to the state at time t. (This is the partial correctness condition.)

This theorem was proved in three steps. Firstly the effect of each node in the state transition diagram of the host machine was described. Combining these effects to describe the effect of each of the possible paths through the state transition diagram gave four theorems. These were then used to show that each path was finite in length (to prove

[2] The initial node plays the same rôle as *ready* in the proof of Gordon's computer discussed above.

termination) and also to demonstrate that the effects on the state were the same as those of the top level specification.

The verification of levels two and three was a straight forward rewriting and case analysis of the definitions of the subcomponents of the high level (block) design. This part used standard HOL techniques.

Data abstraction was used to prove the equivalence of the logical specification and the high level design. The logical specification was written in a relational (e.g. AND (in1, in2, out)) rather than functional (e.g. AND (in1, in2) delivering out) manner. To overcome this difference, an intermediate relational level was introduced. This specification implicitly used the data abstraction function from 6 bit boolean vectors (used at the logical level) to numbers (used at the high level design level). This mapping $V6 : \mathit{bool\ list} \rightarrow num$ is built in to HOL. The equivalence of both the high level design and intermediate specification, and intermediate level and logical design specification was proved, again, using standard rewriting techniques. By transitivity, the logical specification and the top level specification of the counter have thus been proved equivalent.

This example has been given in quite some detail. It shows that even a small circuit, such as a counter, can exhibit most of the techniques and issues of verifying hardware. The proof of the correctness of VIPER will follow the same general proof outline as the counter. It uses decreasingly abstract levels of specification, and equivalence proof for these levels. It also uses structural, temporal and data abstraction.

1.2.6 VIPER micro processor

This must be the best known and most publicised example [CG86, Coh87, Cul87, Coh88]. The VIPER chip was designed at RSRE by Cullyer in response to growing concerns about the reliability and provability of commercial processors in safety critical applications. VIPER was designed with formal verification in mind.

As indicated above, the correctness proof proceeds via a series of increasingly concrete descriptions. Each specification is then proved to correctly implement the specification at the previous level. The four levels at which VIPER has been specified are: (i) The top level specification. It maps states to states. (ii) A finite state automaton, which is the most abstract implementation. (iii) The next level is a block diagram specification. This is a structural specification combining the behaviours of the blocks into the overall behaviour. (iv) The most concrete level consists of the circuit in terms of basic combinatorial gates. This gate level implementation was originally written in the CHDDL ELLA.

The equivalence of these four levels was proved manually by Cullyer. Avra Cohn has since shown the equivalence of the top two levels, and some results about the second and third level using HOL. The automated proof showed some minor errors in the manual proof, but these did not have seem to affected the actual circuit.

The proof of the first equivalence is very similar to that of the counter and is discussed in [Coh87]. The top level specification of VIPER is written as a function mapping states to states with visible components (such as registers) part of the state.The major state machine is the second level specification and takes the form of a finite state automaton. It has twenty four possible execution paths. As in the proof of the counter, we derive an expression describing the cumulated effect of a sequence of transitions on the start state. The disjunction of these expressions can be proved to satisfy the top level specification.

The proof uses the *abs* function described earlier. An "initial node" condition indicates synchronisation of the two time scales, as for the proof of the counter.

The second equivalence proof is substantially more involved [Coh88]. It relates top level specification and a block diagram specification *not* the major state machine and block diagram. The original intention had been to take the second option. However, it turned out to be hard to relate the graphs of the two levels. As the goal was equivalence with the top level, anyway, the new equivalence proof did fine. In fact, equivalence is not proved, as it would not have been practical with the available tools. What has been produced is a "symbolic simulator" as Barrow did for Gordon's computer. The behaviour of the block level has been deduced for all possible instructions of the top level specification. So, to be able to conclude the equivalence proof, the accumulated effect of a simulation of each instruction, would have to be proved equivalent to the behaviour of the top level specification of that instruction. The main problems were the sheer size of this effort, and the difficulty in relating the descriptions at the two levels. To solve these difficulties a considerable infrastructure would be needed, for example, more sophisticated ways of managing large proofs.

Another verification of the Cambridge group is the correctness proof of the Cambridge fast ring chip [Her85, GH]. The implementation was written in Modula2, which was manually translated to HOL. No techniques were used in this proof that have not been discussed above.

1.2.7 An overview

In the examples above, we have seen a number of different formalisms. Rewriting rules, first order logic (with or without induction) and higher order logic. Other formalisms that have been used to verify hardware include various temporal logics [Mos83, FTMO85] and Prolog ([Bar84] and others). CIRCAL [Mil85a, Mil85b, Pez87, DM87] is Milne's formalism. Like CCS [Mil89, Par87] it combines devices using operators such as guarding, composition, non-determinism, termination, choice and abstraction.

Higher order logic is not only used in the Cambridge HOL system [Gor85, Gor87, BG87]. VERITAS is another proof system using the same higher order logic as HOL. It has been developed by Hanna and Daeche at Kent [HD85b, HD85a]. LAMBDA is Mike Fourman's system [FFM89]. It has a different higher order logic which uses the iota operator, whereas HOL uses the epsilon operator.

1.3 Combining CHDDLs and proof systems

My project aims to assess what aspects of CHDDLs and proof systems may be combined to provide the designer of hardware with one framework in which circuitry may be formally designed (or designed and then verified).

In [BP87] Borrione and Piloty outline a similar, but more limited, framework. They propose to translate programs written in a number of CHDDLs into one common proof system, in which properties may then be proved about these programs. There would be one underlying proof system and various translators for the different CHDDLs. They experimented with the translation of VHDL into the first order logic of the Boyer-Moore theorem prover and into the REVE theorem prover. Their work was carried out as part of the CONLAN project [PB85].

As I indicated above, design and verification can support each other so that the idea of combining the two is not like trying to merge two totally unrelated processes. As proof systems usually come with their own environments to manage proofs. I have decided to try and embed a CHDDL into a proof system rather than the other way round.

To be able to embed any CHDDL into a formal system it must have a formal basis. To my knowledge very little has been done in this field. CIRCAL has a formal semantics, and the above mentioned work of Borrione and Piloty goes into the right direction. Recently I have provided a formal operational semantics for a subset of ELLA called MICROELLA [Goo89a]. In [Goo89b] a formal operational and denotational semantics are outlined for PICOELLA, very small functional language, into which MICROELLA may be translated. Appendix A contains more information about this. Using these semantics some initial work has been done on embedding PICOELLA in LAMBDA (see appendix B).

The next section (theoretical aspects) deals with some of the more fundamental questions raised by the proposed embedding. Section 3 discusses the issues that concern the user of the system such as the support of large circuits and what sort of operations should be provided. Section 4 talks about this particular project; what CHDDL and proof system I propose to use and the reasons for doing so. The conclusion gives an overview of this proposal. The various appendices show some concrete information about the particular work that has been done in connection with this project. Finally a bibliography has been provided.

2 Theoretical aspects

The theoretical aspects embedding a CHDDL in a logical framework are described in this section.

To be able to formalise a CHDDL in any way we must have some definitive reference. We could have the choice of various definitions such as:

- The language reference manual of the CHDDL. It will probably written in "careful prose".
- The behaviour of a simulator provided for the CHDDL. However, if available simulators differ, which one would be the definitive simulator? Why would it be superior over the others?
- A formal semantics of the CHDDL. This could be an operational or denotational semantics. As indicated in the introduction, I know of no CHDDLs which have a formal semantics, except for CIRCAL and a subset of ELLA.

If we take the operational semantics as the definition of the CHDDL, we would expect that it agrees with the behaviour of the simulator and the language reference manual.

Assuming that we have some definition of the CHDDL there are various ways in which we can try to incorporate a CHDDL in a proof system:

- We encode the operational semantics of the CHDDL in the proof system. This requires that an abstract syntax for the CHDDL be programmed into the proof system. Using this abstract syntax we can then reason about CHDDL programs as proof system entities. The operational semantics would be written in the form of a set of theorems allowing one to reduce an abstract syntax expression, in a way which would reflect the original operational semantics.

- We provide a *Behaviour* function which maps a CHDDL description to a proof system formula, which reflects the behaviour of the circuit described by the CHDDL program. This *Behaviour* function could either take a CHDDL program from outside the proof system, or operate on an abstract syntax from within the proof system. In both cases a proof system formula (or set of formulae) would be produced. It may be possible to provide *Behaviour* functions emphasising different views of a circuit description.

When we have some embedding an important question would be: is it sound and complete? Briefly, soundness means that we cannot derive any false results. Completeness states that we can derive everything we can derive using the definition of the CHDDL. In the remainder of this section we assume that the definition of the CHDDL is an operational semantics.

2.1 Soundness

The soundness of an embedding indicates that whatever statement we can derive in the proof system is also true in the original definition. This is obviously crucial for a useful system. However, it must be remembered that it is correct *only* with respect to this definition. A CHDDL is only a model of some process in the real world. If this model is inaccurate or incomplete any results derived will reflect these inaccuracies.

Assuming the following definitions

$$\begin{gathered} Time = I\!N \\ Signal = Time \to Value \\ i, o : Signal \\ Behaviour : CHDDL \to Signal \times Signal \end{gathered}$$

the soundness theorem for a *Behaviour* function would take the form:

$$\forall p.\ \forall i, o.\ \text{“} \vdash^{PS} Behaviour(p)(i, o)\text{”} \ \Rightarrow\ \text{“}p(i) \Rightarrow_{OS} o\text{”} \tag{1}$$

Thus time is discrete and modelled by natural numbers. Signals are values varying over time. Behaviour is a relation between input and output signals. PS and OS stand for proof system and operational semantics respectively. Thus equation (1) states that if signals i and o are related by the behaviour extracted from the circuit description p in the proof system, then the operational semantics agrees. Note that it is a meta theorem, stated outside the proof system.[3] The antecedent of the implication is a proof system formula, the conclusion is not.

For the embedded operational semantics we would prove a statement that looks very similar, but probably has a more direct proof of correspondence, because the two operational semantics would be similar.[4]

$$\forall p.\ \forall i.\ \text{“} \vdash^{PS} AbsSyn(p)(i) \Rightarrow^{PS}_{OS} o\text{”} \ \Rightarrow\ \text{“}p(i) \Rightarrow_{OS} o\text{”} \tag{2}$$

[3] But see equation (3).

[4] See, however, section 3.2.

Note the function *AbsSyn* from the CHDDL program to the embedded abstract syntax in the proof system. The arrow $\Rightarrow_{OS}^{PS}$ is a formalisation in the proof system of $\Rightarrow_{OS}$. Thus this condition expresses that the formalisation of the operational semantics in the proof system gives the same results as the original operational semantics for all programs p and inputs i.

Assuming the soundness proof (2), we could prove that an embedding of the *Behaviour* function was correct:

$$\vdash^{PS} \forall p : AbstractSyntax.\ \forall i, o.\ Behaviour^{PS}(p)(i, o) \rightarrow^{PS} \ p(i) \Rightarrow_{OS}^{PS} o \tag{3}$$

The arrow $\rightarrow^{PS}$ is the proof system implication, and $Behaviour^{PS}$ is the formalisation in the proof system of *Behaviour*. Note that p is now a proof system entity and therefore $Behaviour^{PS} : AbstractSyntax \rightarrow Signal \times Signal$.

2.2 Completeness

Proving the completeness of the embedding of the CHDDL in the proof system amounts to showing that "all the results we can get using our definition of the CHDDL we can prove using our embedding." This is obviously quite important, as we would like not to loose any information by formalising our CHDDL. To prove completeness for the *Behaviour* function we have to show the following:

$$\forall p.\ \forall i.\ \text{“}p(i) \Rightarrow_{OS} o\text{”} \Rightarrow \ \text{“}\vdash^{PS} Behaviour(p)(i, o)\text{”} \tag{4}$$

The embedded operational semantics completeness result would take the following form:

$$\forall p.\ \forall i.\ \text{“}p(i) \Rightarrow_{OS} o\text{”} \Rightarrow \ \text{“}\vdash^{PS} AbsSyn(p)(i) \Rightarrow_{OS}^{PS} o\text{”} \tag{5}$$

Analogue to (3) we would have:

$$\vdash^{PS} \forall p.\ \forall i.\ \exists o.\ p(i) \Rightarrow_{OS}^{PS} o \rightarrow^{PS} \ \ Behaviour^{PS}(p)(i, o) \tag{6}$$

Other completeness results may use some notion of equality of programs. For example, if we define operational equality from the definition as follows:

$$p =_{OP} q \Leftrightarrow \ (\forall i.p(i) \Rightarrow_{OS} o \wedge q(i) \Rightarrow_{OS} o') \rightarrow o = o' \tag{7}$$

We can judge the embedded operational semantics and *Behaviour* function relative to this:

$$p =_{OP} q \Rightarrow \ \text{“}\vdash^{PS} \ \ Behaviour(p) = Behaviour(q)\text{”} \tag{8}$$

$$p =_{OP} q \Rightarrow \ \text{“}\vdash^{PS} (\forall i.p(i) \Rightarrow_{OS}^{PS} o \wedge q(i) \Rightarrow_{OS}^{PS} o') \rightarrow^{PS} \ o = o'\text{”} \tag{9}$$

$$\vdash^{PS} p =_{OP}^{PS} q \rightarrow^{PS} \ \ Behaviour^{PS}(p) = Behaviour^{PS}(q) \tag{10}$$

In equation 10 $=_{OS}^{PS}$ stands for

$$p =_{OP}^{PS} q \leftrightarrow^{PS} \ \ ((\forall i.p(i) \Rightarrow_{OS}^{PS} o \wedge q(i) \Rightarrow_{OS}^{PS} o') \rightarrow^{PS} \ o = o') \tag{11}$$

3 Pragmatic issues

Listed below are a number of facilities that could be provided to help a user of the design/proof system. Most of these are essentially open ended and may be expanded into any desired depth.

3.1 Design support

It would be very useful to have facilities to help the user with various abstractions used when designing large systems. Discussed below are four such abstractions following [Mel87].

Data abstraction A data abstraction step indicates the refinement of an abstract datatype into a more concrete, specialised, representation. This involves providing the high level and low level datatype and the abstraction function from the low level to the high level type. We may then automatically rewrite the high level implementation and prove that the properties holding for the high level implementation also hold for the low level datatype. This may, of course, not be quite so straight forward as there may be complications introduced by viewing the circuit at a lower level (such as set up times). Also, a refinement from integers to N bit vectors will introduce constraints on the range of the integers than may be represented. These would normally be added to the assumptions of the low level expressions. Alternatively, the high level specification may have been written with these limitations in mind. For example, consider the following specification in HOL from [CG86]:

$$COUNTER(count, loadin, func) = \ \ldots \text{ let } value = VAL6\ count \text{ in}$$
$$(funcnum = 2)\ \ =>\ \ ((value = 63)\ \ =>\ \ WORD6\ 0 \mid WORD6(value + 1)) \mid\ \ldots$$

This assumes that incrementing 63 will deliver 0, that is, 6 bits are available. Note that this specification talks about bit vectors, yet uses VAL6 and WORD6 to switch to natural numbers to specify addition.

Structural abstraction We break the design down into subunits which will probably be interconnected. We have to show that the total interaction of these units implements the original specification. The system can help here by providing ways of managing and organising goal stacks in various ways as to organise the proofs in independent chunks suggested by the subunits.

If we have an embedded abstract syntax in the proof system, as outlined in the previous section, then we could formalise structural abstraction as follows:

$$\vdash^{PS} \forall in, out : Signal.\ \exists c_1, c_2 : AbstractSyntax.$$
$$COMPOSE(c_1, c_2)(in, out) \rightarrow^{PS}\ \ SPEC(in, out)$$

$COMPOSE$ combines the behaviours of c_1 and c_2; a simple example would be $COMPOSE(c_1, c_2)(in, out) = \exists x.c_1(in, x) \wedge c_2(x, out)$.

Behavioural abstraction Here we incompletely specify the behaviour of a component; only the those properties relevant to the environment are specified and proved. This could be facilitated if there were ways of linking properties needed for a particular context to specifications and implementations. This leads us to idea of contexts and holes in descriptions, which we will discuss below.

Temporal abstraction We already have seen some examples of this namely the micro instruction versus instruction level in a proof of Gordon's computer [JBG86] and also the finite state automaton to block level (for VIPER in [Coh87, Coh88] and in the simple counter example [CG86]).

We could give a condition indication at which times the high and low level times synchronise (for example that the micro instruction counter is at location zero). We can then automatically construct a function which, given a time, delivers the time at which the two levels synchronise, provided we know the behaviour of the two levels at their respective time scales.[5] If we can show that this behaviour is the same (or at least satisfies the same properties in which we are interested) for both levels from one synchronisation time tick to the next then the two levels are the same (with respect to the properties of interest). We would thus have a sort of behavioural equivalence. It may of course be necessary to impose some extra conditions on the low level description (such as stability during set up time) which not necessarily be visible at the higher level (because they would be abstracted away).

Using the various abstractions would give a layered design in which design decisions would be documented by information the user would have had to supply to the proof system. For example, the decision to model integers by 32 bit values would be reflected in a data abstraction step $DataAbstraction(int, 32bitvalue, 32bittoint, constr)$ where $32bittoint$ would be the abstraction function, and $constr$ the constraints added to the low level datatype.

If the design process takes place in a layered way it may be possible to identify various parts of a CHDDL that are used mainly, or exclusively, in one particular part of the design process. If this is the case, then it would be worth while to refine design support to each of these levels. Transitions from one design level to another could be approached in a way similar to individual design abstractions such as those discussed above.

After we would have proved the correctness of a data abstraction step in the proof system we could make the same change in the CHDDL. Unfortunately this would probably not be automatic. Of course we can check this (manual) change by translating the new CHDDL program to a proof system formula which we can then prove using the initial data abstraction as a guide. Alternatively we could just try some abstraction or other change in the CHDDL first and then after translating it into the proof system try and prove it.

3.2 Reasoning support

We would expect to have a number of tactics to aid reasoning about programs translated from the CHDDL to the proof system. For example:

1. If we use the *Behaviour* function to extract information from the CHDDL program to the proof system we would expect the resulting formulae to be in a reasonably standard form. This allows tactics to be devised to delete internal wires for example, or to replace implementations by specifications (provided the implementation has been proved to imply the specification) and vice versa. For example if we have the following CHDDL definitions:

$$\text{FN AND (TWOBITS : in)} -> \text{TWOBITS: } \ldots$$
$$\text{FN ADD (TWOBITS : in)} -> \text{TWOBITS: } \ldots \text{ AND } \ldots$$

[5] Using *abs* on page 5 for example.

we would expect to get something like

AND_BEHAVIOUR#(in,out)=...
ADD_BEHAVIOUR#(in,out)=... AND_BEHAVIOUR#(...) ...
CIRCUIT_BEHAVIOUR#(in,out) = ADD_BEHAVIOUR#(in,out)

We could first prove that AND_BEHAVIOUR implies AND_SPECIFICATION and then replace all AND_BEHAVIOURS by AND_SPECIFICATIONs (which we assume are simpler to reason about). Then we could do the same for the adder, but with some of the complexity removed because of the simpler AND_SPECIFICATIONs. These proofs will, of course, only be done once (see library support below). Here too we would want to be able to manage our goal stacks to be able to do a subproof (lemma) without losing our view on the overall proof.

2. If we are dealing with an embedded operational semantics it would only be natural to supply supply tactics to reduce the the individual constructs. Examples could include:

   - reduceIfTac to reduce the top level If statement and similar tactics for other statements;
   - reduceTopConstruct to reduce the construct which happens to be on the top of the abstract syntax term;
   - reduceN n; to reduce the top n constructs;
   - reduceAll to reduce the current term to a normal form (ie a result value);
   - reduceAllTimes to reduceAll over all the time for which we have an input value.

   It may be possible to provide operational semantics at different levels of abstraction. For example, we may want to reflect the original CHDDL description as much as possible. On the other hand tracing the operational semantics of a net list model of the circuit may be useful to examine the low level behaviour of the description. This could aid the understanding of behaviour at boundaries of different (high level) functional units, as these would be flattened out. This would be achieved, for example, by translating ELLA to PICOELLA and then using PICOELLA's *Behaviour* function on the translated program.

3. The concept of a probe (or breakpoint) in a circuit is quite useful; we can look at the value of one particular wire in a circuit to provide information about the internals of a circuit, which are normally not visible.[6]

   Extending this leads to the idea of leaving the implementation of a particular function (or instance of a function) totally unspecified. For example, suppose that we have a module resolving accesses to a bus. We could simulate any resolving method by just setting the output of the hole in the surrounding circuit to the particular value we want, without actually having to write an implementation for the specific method we are simulating. Alternatively we can check that the environment of a hole can cope with various eventualities (for example, a jam on a bus, which

[6] An interesting article about what we can prove by looking at a circuit as a black box is [Bry85].

may be hard to generate using just the inputs of the whole circuit). A third way in which we can use holes is to try various standard implementations (but with different timing characteristics say) of a circuit and see if any one performs better or worse in that context. (This relates to having context dependent specifications, to see exactly what the context requires of a function. It would be useful to know that a slower (presumably cheaper) implementation of a subcircuit could do the job as well as a faster one in the particular context.)

It will be useful to have a library of interesting properties which may be used as outline above. It could include:

(a) the minimum set up time of a circuit

(b) the maximum hold time

(c) the minimum or maximum delay through a circuit

(d) response times

(e) the size of the circuit (measured in gates, wires or transistors)

(f) uses of bidirectionality of wires

(g) the access point of busses

3.3 Library support

When working with large circuits or designs we need to be able to re-use proofs (in possibly two different ways: once we have proved something we can just use that result as a theorem, and also if we have changed something we may want to try to re-run the old proof on the new circuit if the change is sufficiently small). Proof systems such as HOL and LAMBDA provide such facilities, but it may be worth while to investigate how for example the browser facility in LAMBDA may be used to its full potential.

Maybe more sophisticated ways of managing goal stacks could be provided. Successive versions of CHDDL fragments, their specifications and proofs should be identifiable, especially if abstractions such as outlined above have been used. In this case we could annotate different versions with their respective abstraction functions.

4 Embedding ELLA in Lambda

The scope of this project will be quite independent of the particular CHDDL or proof system. However, in order to be able to judge the ideas of the project in practice, some choice has to be made for these. Outlined below are the arguments for chosing ELLA as the CHDDL, and LAMBDA as the proof system, for this project.

4.1 Why ELLA?

I have chosen ELLA as the CHDDL to use after studying ELLA and VHDL in some depth. ELLA appeals because of its simplicity, both in terms of its concepts and the number of them. There are a fairly small number of base constructs into which the more sophisticated statements can be translated. Its seems to be a semantically quite a clean language. Royal Signals and Radar Establishment (RSRE) Malvern, where ELLA was designed, is essentially a programming language community. In [MPT85] it is mentioned that

> The designers believed in strongly typed ... functional ... orthogonal languages.

ELLA is exactly that. The operational semantics of MICROELLA, a subset of ELLA [Goo89a] did not present any significant difficulties.

VHDL, in contrast, has the look and feel of Ada; it is a large, verbose language. This is no surprise as the DoD, commissioning the development and definition of VHDL, required the use of Ada constructs wherever possible [BGL+85, SLM+85]. It is more powerful with its procedural parts, which allow access types, for example, than ELLA. Thus VHDL allows one to write procedural descriptions that do not directly (or obviously) correspond to a particular hardware implementation. In ELLA, however, every circuit description, no matter how abstract, corresponds to a model of hardware. (In this model the data values on the wires could be quite high level of course.) An interesting example of this can be found in the ELLA tutorial [Pra86b], exercise 10/2, where a circuit generating prime numbers is described.

Another point to note is that the designers of ELLA did not want the user to have to know about the ELLA simulator. Thus concepts like delays are not explained relative to the simulator.[7] VHDL takes the simulator mechanism into the language definition. Its (prose) language definition describes the effects of constructs in terms of the operations of the simulator:

> The effect of execution of a signal assignment is defined in terms of its effect on the projected output waveforms representing the current and future values of signals.
>
> *IEEE Standard VHDL Language Reference Manual. Section 8.3.1*

This shows that VHDL has a forward-looking view of time ("projected waveforms"), whereas ELLA has a backward-looking view:

> The effect of the ELLA method of handling of time information is to produce a language where the current values in the network are determined by the values that signals have held in the past.
>
> *The Design Rationale of ELLA, A Hardware Design and Description Language.* [MPT85]

I prefer the backward-looking view of time as it does not involve destructively updating previous predictions for future signal values, but always uses information that has already been computed (at a previous clock tick) and is thus fixed. We have a fixed history rather than a changing future to consider. [Goo89a] shows how the backward-looking method may be used semantically.

Overall, the simpler model of hardware and time that ELLA uses lends itself more to formalisation.

Also, a collaboration with Praxis Systems Plc would be a possibility, providing support for the more technical issues. Praxis Systems Plc acquired the licence to market and develop ELLA in 1985.

[7] Rather, they are explained relative to a minimal simulator model, indicating only the discrete time basis.

4.2 Why Lambda?

I have chosen LAMBDA as the proof system to embed ELLA in. LAMBDA has user friendly features such as a browser and the ability to select parts of expressions using a mouse. These are important in allowing a system to be built, in which a user need not know all intricacies of an implementation. For example, to use an embedded operational semantics, the user can just use a menu and/or mouse to select high level objectives, such as "reduce this part of the expression". There is no need to know what the individual rules or tactics are called that actually do this. This is useful for users that may want to use just part of the capabilities of the system.

Another important factor is the ability to use ML with and within the LAMBDA system. As outlined in appendix B, ML may be used to program a *Behaviour* function. This function is not part of the proof system, but as LAMBDA is written in ML, they can both reside in the same ML system. This obviously eases any interfacing between the proof system and utilities that may be written. It is possible to declare and reason about ML data types and functions (though more restricted that Standard ML) within LAMBDA itself. This alone allows formal meta reasoning within LAMBDA about the process of embedding a *Behaviour* function, for example. The sort of meta reasoning we would be interested in is explained on page 11.

Mike Fourman, the principal designer of LAMBDA, is at Edinburgh University. This will make it easier to resolve problems that are of a more technical nature.

4.3 The project

The project will start with a scaled down version of the goals discussed in sections 2 and 3. I will continue the work on embedding PICOELLA in LAMBDA. Firstly I will provide a PICOELLA to LAMBDA formulae *Behaviour* function (see section 2 and appendix B). The resulting formulae will be large, and I will investigate the need for facilities to manipulate and reson about them, such as tactics.

Next, the operational semantics for PICOELLA will be embedded. No work has yet been done for this. Here tactics will definitely be programmed to help reasoning, and the browser in LAMBDA will be used to make everything as accessible as possible.

The two different embeddings above, correspond to different ways of designing hardware. The *Behaviour* function would correspond to a top down (or meet in the middle) design approach. The starting point of a design would be the statement $\vdash^{PS} \exists c.\ P(Behaviour(c))$.[8] It expresses that the circuit we are to design meets the properties P. Successive refinement steps would take the form indicated on page 12.

The operational semantics would allow the user to prove equivalence of implementations in several ways:

- Suppose a low level implementation of a circuit must be shown to be equivalent to a high level specification (written in a functional way). This may be proved by showing the commutativity of the diagram

$$\begin{array}{ccc} \text{High level circuit} & \Rightarrow_{OS} & Value \\ \uparrow_{ABS} & & \uparrow_{ABS} \\ \text{Low level circuit} & \Rightarrow_{OS} & Value' \end{array}$$

[8]This could perhaps be generalised to $\Gamma \vdash^{PS} \exists c.\ P(Behaviour(c))$, where Γ contains the resources with which, or the constraints within which the design must be designed.

The arrows $\uparrow_{ABS}$ indicate that the low level data values may be abstracted to the high level using an abstraction function. The diagram thus expresses that the abstracted output of evaluating the low level program with certain low level input values, agrees with running the high level circuit on the abstracted input values. This is not so much a (data) abstraction step, as a post hoc verification. It may be useful, if a designer's intuition indicates a certain type of refinement, which may be non-trivial to express using one of the "standard abstractions".

- Alternatively, it would allow proofs, requiring some sort of symbolic evaluation, to be completed more uniformly. In the case of a temporal abstraction, this may be very useful. In the proofs of Gordon's computer definitions were repeatedly expanded to show that a sequence of micro instruction correspond to a particular instruction. Providing we have a CHDDL fragment describing the particular implementation level, an operational semantics could achieve the same effect in a more structured way. To make this sort of application useful, the use of free variables in the abstract syntax terms should be allowed.[9] This would result in a symbolic answer, in case no information about these free variables is known. Alternatively, being able to use properties of the free variables, may still result in a "real" result value.

  The first of these options would take the form:

$$\cfrac{\cfrac{\overline{E' \vdash \ Component\ i \Rightarrow^{PS}_{OS}\ o'}}{E \vdash \ldots \Rightarrow^{PS}_{OS} \ldots}}{E \vdash \ Circuit\ i \Rightarrow^{PS}_{OS}\ o}$$

  In this case both o and o' would be symbolic values (formulae) rather than actual result values. The expression o would depend on the possible answers of the subcomponents. In general o and o' will be a conjunction of these possible answers. For example, if *Component* happened to be an `if` statement the top rule would be of the form:

$$\begin{array}{c}\hline E \vdash if(\ldots)\ i \Rightarrow^{PS}_{OS}\ \begin{array}{l}(match(\ldots) = tt \rightarrow o = v) \wedge \\ (match(\ldots) = ff \rightarrow o = v')\end{array}\end{array}$$

  However, if we have some information about the free variables, then we may make use of this:

$$\cfrac{\cfrac{\cfrac{\text{P(Component)}}{E' \vdash \ Component\ i \Rightarrow^{PS}_{OS}\ o'}}{E \vdash \ldots \Rightarrow^{PS}_{OS} \ldots}}{E \vdash \ Circuit\ i \Rightarrow^{PS}_{OS}\ o}$$

  Here, using $P(Component)$, we may be able to produce a proper result value for o' (and o).

One may also be able to use the operational semantics to get an operational, low level, understanding of the circuit. The output at certain boundary values may be useful, and could be obtained quickly using the operational semantics.

[9] Remember that we are talking about an embedded operational semantics operating on an abstract syntax in the proof system.

A problem of picoELLA is that it is a very small language. It consists of expressions only, and because of this, multiple instantiations of a function become multiple occurrences of the same expression. (The correct parameters will have been substituted for each instantiation.) Thus the level of description of picoELLA is too low to be able to take advantage of the higher level structure. See the example on page 13 for an example. So, even though the provision of a *Behaviour* function and an operational semantics will be useful, ultimately we would want to move to a less Spartan language, in which we can fully express the whole design process. However, the effort spent on picoELLA will not be in vain, as it will enable us to concentrate on the more theoretical aspects of the issues raised by the embeddings. So far, the theoretical problems have not been addressed in any detail.

Following this, we can optionally consider the problem of translating ELLA into picoELLA. The completion of this part, would, in theory, finish the project. However, the translated programs will most likely to be too low level (especially for non trivial descriptions), and also bear little resemblance to the original descriptions. This would impair the high level reasoning. The translation could be formalised and, if desired, implemented. Although it is not one of the priorities of the project, it would be interesting to see what problems would arise if we would try to translate other CHDDLs to picoELLA.

Doing the project in miniature will provide a valuable intermediate estimate of the effort required to complete a similar process for the whole of ELLA (or at least a representative subset). If needed, the effort to be spent on various parts of the project can be increased or reduced. Thus the rest of the project will have the same outline as the picoELLA effort. It may be that a not so much larger superset of picoELLA can remedy its shortcomings. In this case the emphasis of the remainder of the project may be focussed on this language.[10] We will certainly be dealing with some subset of ELLA. The justifications for this have been given previously. Thus, the evaluation of the new language will focus on the aspects picoELLA missed.

4.4 Project overview

In overview we get the following:

1. picoELLA in Lambda:
    - Program the *Behaviour* function to generate formulae describing the behaviour of picoELLA descriptions.
    - Maybe provide some tactics for reasoning about these formulae. The automated generation of of these formulae will ensure a certain uniformity, aiding the implementation of circuit independent tactics.
    - Embed the operational semantics for picoELLA in Lambda.
    - Provide tactics to reason about this semantics. The remarks above, about the *Behaviour* tactics, also apply here.
2. Consider the theoretical aspects of the project in more detail.
3. **Optional**

[10]In this light a translation from ELLA or microELLA to picoELLA may be useful.

- Formalise the ELLA to picoELLA translation. (See the remarks about the fixity of ELLA for this part of the project in the previous subsection.)
- Implement this formalisation in the proof system,
- Provide tactics to reason about generated formulae.

4. Re-asses the remainder of the project. Increase or decrease the rôle of subprojects according to their expected demands.

5. Embed ELLA in LAMBDA

   - Write a type system and operational semantics for ELLA. Maybe a denotational semantics can be provided.
   - Produce a *Behaviour* function from ELLA programs to LAMBDA formulae. Implement this function and prove the appropriate correctness results. Provide tactics to aid reasoning about the generated formulae.
   - Embed the ELLA operational semantics in LAMBDA, again proving any correctness results we desire. Implement tactics to try to raise the functionality of the operational semantics embedding to that of a (symbolic) simulator (though not necessarily so fast!)

5 Conclusion

We have seen the growing importance of the formal verification of hardware. Combining CHDDLs and proof systems may provide a step into the right direction by allowing formalisation of the design process and the ability to prove properties about the designs and implementations with one framework.

I have indicated two ways in which we could embed a CHDDL in a proof system. Firstly we can provide a function *Behaviour* which extracts information from the circuit description and produces a proof system formula (or set of formulas). There may be scope for providing various functions emphasising differents views of the circuit (e.g. data flow versus timing behaviour).

The second method involves embedding an operational semantics for the CHDDL in the proof system. This could be done at various levels, for example at a netlist level where all function instances are flattened out into one network. Alternatively, we could try and keep an as high level view as possible, that is, reflecting the original CHDDL text as much as possible. Both of these methods can be supported by various tactics, support from a browser etc.

The combination of ELLA and LAMBDA would, I think, work very well. The initial effort of doing the project on a smaller scale (embedding picoELLA) will undoubtedly help in planning the major effort of embedding ELLA. Problem areas will most likely be roughly the same, and project time scales can be re-assessed if necessary.

I think that this project will not only serve as a case study in how to embed a CHDDL in a proof system, but will also provide information on how hardware designers go about designing and implementing circuitry. We can then see in which ways this process may be improved. This project will also have to define the semantics of ELLA, which will be a valuable effort in its own right. Feedback on how we could, or should, use the semantics of a CHDDL will be produced by the embedded operational semantics. As this embedding can function as a symbolic simulator we could compare this to the way in which standard CHDDL simulators are used.

A The picoELLA language

PICOELLA is a very small functional language. It was designed to reflect the most basic features of the CHDDL ELLA [Pra86a]. PICOELLA's statements include the delay, let, if and fix. The if statement is a simpler form of the ELLA case construct. The fix statement is not present in ELLA but was needed to allow feedback loops unbroken by delays[11]. The delays introduce a discrete time basis. It is possible to translate MICROELLA [Goo89a] into PICOELLA via an intermediate language (INTERM). The first translation step removes state variables and re-orders makes. The second part is more difficult; it expands recursively instantiations of functions by their function bodies and supplies the correct parameters. Delayless feedbacks are translated into fix statements.

The syntax of PICOELLA is listed below; the abstract datatype used to represent it in ML can be found in the next appendix.

$$
\begin{array}{rcl}
program & ::= & delays\ INPUT\ (\llbracket name, \rrbracket *\ name)\ IN\ se. \\
delays & ::= & \llbracket LET\ name =\ DELAY\ (int, values)\ IN\ \rrbracket * \\
se & ::= & values \parallel name \parallel \\
 & & name(se) \parallel se[int] \parallel (\llbracket se, \rrbracket *\ se) \parallel \\
 & & LET\ name = se\ IN\ se \parallel \\
 & & FIX\ name = se\ IN\ se \parallel \\
 & & IF\ se\ MATCHES\ choosers\ THEN\ se\ ELSE\ se \\
choosers & ::= & chooser\ \llbracket \mid chooser \rrbracket * \\
chooser & ::= & BOOL \parallel TRUE \parallel FALSE \parallel (\llbracket chooser, \rrbracket *\ chooser) \\
values & ::= & ? \parallel TRUE \parallel FALSE \parallel (\llbracket values, \rrbracket *\ values)
\end{array}
$$

PICOELLA has been implemented in ML and the system includes a parser, type checker, interpreter and compiler (which compiles PICOELLA into ML). A sample session of the system is shown below. The circuit used as an example is four bit adder. Note that the implementation of PICOELLA shown below allows macros to be used to increase the readability and conciseness of the source file. Macros are expanded on parsing the programs.

```
[1] kgg@luing Proposal picoELLA
val it = true : bool

               picoELLA interpreter and compiler system
                   with Macros and pretty printer.
                    (C) Kees Goossens 26/10/1989

Type help() for more information
```

[11]ELLA does not handly delayless feed back loops satisfactorily in the current release (issue 3.0), but it is expected that it will do so in the next version.

```
val it = () : unit
- reset_file "/home/kgg/Simulators/Pgms/4b_add3.pe";
val it = () : unit
- parse (true,true); (* Listing & line numbers *)
1 { Four bit adder: }
2
3  MACRO not ( x ) = IF x MATCHES true THEN false ELSE true IN
4
5  MACRO and2 ( x ) =
6          IF x MATCHES ( true, true ) THEN true ELSE false IN
7  MACRO and3 ( x ) =
8          IF x MATCHES ( true, true, true ) THEN true ELSE false IN
9  MACRO and4 ( x ) =
10         IF x MATCHES ( true, true, true, true ) THEN true ELSE false IN
11
12  MACRO or2 ( x ) =
13         IF x MATCHES ( false, false ) THEN false ELSE true IN
14  MACRO or3 ( x ) =
15         IF x MATCHES ( false, false, false ) THEN false ELSE true IN
16  MACRO or4 ( x ) =
17         IF x MATCHES ( false, false, false, false ) THEN false ELSE true IN
18
19  MACRO parity3 ( x ) =
20         IF x MATCHES ( true, false, false ) | ( false, true, false ) |
21                      ( false, false, true ) | ( true, true, true )
22                      THEN true ELSE false IN
23
24  MACRO onebitadd ( a, b, c ) =
25         LET a = a IN LET b = b IN LET c = c IN { Avoid duplication: }
26         LET and_bc = #and2 (b,c) IN
27         LET and_ca = #and2 (c,a) IN
28         LET and_ab = #and2 (a,b) IN
29         ( #parity3 ( a, b, c ), #or3 ( and_bc, and_ca, and_ab ) )
30         { out, carry }
31  IN
32
33  MACRO fourbitadd ( in1, in2, carry ) =
34         LET in1 = in1 IN LET in2 = in2 IN { Avoid duplication: }
35         LET add4 = #onebitadd ( in1[4], in2[4], carry )   IN
36         LET add3 = #onebitadd ( in1[3], in2[3], add4[2] ) IN
37         LET add2 = #onebitadd ( in1[2], in2[2], add3[2] ) IN
38         LET add1 = #onebitadd ( in1[1], in2[1], add2[2] ) IN
39
40         ( add1 [2], ( add1 [1], add2 [1], add3 [1], add4 [1] ) )
41           {carry}   {add, msb=1..lsb=4}
42  IN
43
44  INPUT ( in1, in2, carry ) IN #fourbitadd ( in1, in2, carry ).
val it = () : unit
- max_depth 10;
val it = () : unit
- pretty_print_pgm();
INPUT ( in1, in2, carry ) IN
LET in1 = in1 IN
LET in2 = in2 IN
LET add4 = LET a = in1[4] IN
           LET b = in2[4] IN
           LET c = carry IN
```

```
           LET and_bc = IF ( b, c )
                        MATCHES ( t, t ) THEN
                        t ELSE
                        f IN
           LET and_ca = IF ...
                        MATCHES ( t, t ) THEN
                        ... ELSE
                        ... IN
           LET and_ab = ... IN
           ... IN
LET add3 = LET a = in1[3] IN
           LET b = in2[3] IN
           LET c = add4[2] IN
           LET and_bc = IF ...
                        MATCHES ( t, t ) THEN
                        ... ELSE
                        ... IN
           LET and_ca = ... IN
           ... IN
LET add2 = LET a = in1[2] IN
           LET b = in2[2] IN
           LET c = add3[2] IN
           LET and_bc = ... IN
           ... IN
LET add1 = LET a = in1[1] IN
           LET b = in2[1] IN
           LET c = ... IN
           ... IN
( add1[2], ( add1[1], add2[1], add3[1], add4[1] ) ).
val it = () : unit
- (* Some abbreviations: *)
- val t = Basic True and f = Basic False and u = Basic Undef;
val t = Basic True : value
val f = Basic False : value
val u = Basic Undef : value
- val inputs =                        (* X + Y + Carry = Carry + Z *)
-   [ List [List [t,f,t,f], List [f,t,f,t], f], (* 10+5+0  = 0+15  *)
-     List [List [t,f,t,f], List [f,f,t,f], f], (* 10+2+0  = 0+12  *)
-     List [List [t,t,t,f], List [f,f,t,t], f], (* 14+3+0  = 16+1  *)
-     List [List [t,t,t,t], List [t,t,t,t], f], (* 15+15+0 = 16+14 *)
-     List [List [t,t,t,t], List [t,t,t,t], t]];(* 15+15+1 = 16+15 *)
val inputs = [List [List [#,#,#,#],List [#,#,#,#],Basic False],List [List [#,#,#,#]
- g inputs;
Time        Input        Output
-------------------------------
0    ( ( t, f, t, f ), ( f, t, f, t ), f )    ( f, ( t, t, t, t ) )
1    ( ( t, f, t, f ), ( f, f, t, f ), f )    ( f, ( t, t, f, f ) )
2    ( ( t, t, t, f ), ( f, f, t, t ), f )    ( t, ( f, f, f, t ) )
3    ( ( t, t, t, t ), ( t, t, t, t ), f )    ( t, ( t, t, t, f ) )
4    ( ( t, t, t, t ), ( t, t, t, t ), t )    ( t, ( t, t, t, t ) )
val it = () : unit
- compile ( hd inputs );
[opening /tmp/...]
val initialise_delays = fn : unit -> unit
val crun = fn : value list -> unit
[closing /tmp/...]
val it = () : unit
- crun inputs;       (* Faster than "g inputs" but can't see this... *)
```

```
Time        Input        Output
------------------------------
0    ( ( t, f, t, f ), ( f, t, f, t ), f )    ( f, ( t, t, t, t ) )
1    ( ( t, f, t, f ), ( f, f, t, f ), f )    ( f, ( t, t, f, f ) )
2    ( ( t, t, t, f ), ( f, f, t, t ), f )    ( t, ( f, f, f, t ) )
3    ( ( t, t, t, t ), ( t, t, t, t ), f )    ( t, ( t, t, t, f ) )
4    ( ( t, t, t, t ), ( t, t, t, t ), t )    ( t, ( t, t, t, t ) )
val it = () : unit
- run [ hd inputs ];
The program

INPUT ( in1, in2, carry ) IN LET in1 = in1 IN LET in2 = in2 IN LET add4 = LET a

Has type ( ( Bool, Bool, Bool, Bool ), ( Bool, Bool, Bool, Bool ), Bool ) ->
           ( Bool, ( Bool, Bool, Bool, Bool ) )

Running...

*** time 0 ***

Input        ( ( t, f, t, f ), ( f, t, f, t ), f )
Output (traced)
[INPUT] input=( ( t, f, t, f ), ( f, t, f, t ), f ) IN LET in1 = in1 IN LET in2
[VAR]   Venv(in1) = ( t, f, t, f )
     .
     .  ...Lots of output deleted...
     .
[IF]    IF ( a, b, c ) MATCHES ( t, f, f ) | ( f, t, f ) | ( f, f, t ) | ( t, t,
[TUPLE] ( a, b, c )
[VAR]   Venv(c) = f
[VAR]   Venv(b) = f
[VAR]   Venv(a) = t
[VALUE] t
[VALUE] f
[TUPLE] ( add1[2], ( add1[1], add2[1], add3[1], add4[1] ) )
[TUPLE] ( add1[1], add2[1], add3[1], add4[1] )
[INDEX] add4[1]
[VAR]   Venv(add4) = ( t, f )
[INDEX] add3[1]
[VAR]   Venv(add3) = ( t, f )
[INDEX] add2[1]
[VAR]   Venv(add2) = ( t, f )
[INDEX] add1[1]
[VAR]   Venv(add1) = ( t, f )
[INDEX] add1[2]
[VAR]   Venv(add1) = ( t, f )
( f, ( t, t, t, t ) )

val it = () : unit
- ^D
52.2u 4.5s 21:16 4% 0+308k 1+4io 2pf+0w
[2] kgg@luing Proposal    wc /tmp/...           -- the compiled circuit
     312    1860       8351 total
[3] kgg@luing Proposal
```

B Behaviour function example

The abstract syntax of PICOELLA programs is represented at the ML level as the following datatype:

```
datatype expr = Delay of ( string * int * value * expr ) |
                Input of ( string list * expr ) |
                Let of ( string * expr * expr ) |
                Fix of ( string * expr * expr ) |
                If of ( expr * expr * expr * chooser ) |
                Name of string |
                Call of ( string * expr ) |
                Value of value |
                Index of ( int * expr ) |
                Tuple of ( int * expr list ) |
```

We can construct a *Behaviour* function, written in ML [HMT89], mapping this datatype to a string which can then be used by LAMBDA's `parseDecList` function delivering a new parser environment and a number of new rules. Shown below is a *very* rough outline of such a function.

```
fun Behaviour ( Delay (d,n,v,e) ) =
    "exists d_in, d_out.  DELAY_BEH# (n,v,d_in,d_out) /\ " ^
    ( Behaviour e )
|   Behaviour ( Input (s,e) ) =
    "exists s.  s === inp /\ " ^ ( Behaviour e )
|   Behaviour ( Call (d,e) ) =
    "(exists out.  " ^ ( Behaviour e ) ^
    " /\ out === d_in ) /\ out === d_out "
|   Behaviour ( ...) = ...
```

The idea is that every construct is a box with an output called `out`. Thus, when we want to call delay `d` we connect its input (`d_in`) to the output of the expression which is its argument: `( Behaviour e ) ^ " /\ out === d_in " )`. The output of the call construct at time t is its input at time $t - N$, except at times $1, \ldots, N-1$ when it is the initial value of the delay.

A concrete example is:

```
LET d = DELAY (1,?)  IN
INPUT (x) IN
             d ( IF x MATCHES (true,true)
                 THEN true
                 ELSE IF x MATCHES (false,false)
                      THEN true
                      ELSE false ).
```

The two if statements may be combined into one: `IF x MATCHES (true, true)| (false, false) THEN true ELSE false`. The given program is, however, more illustrating. It is represented in the abstract syntax as:

```
Delay ("d",1,Undef,
       Input ("x",
              Call ("d",
                    If ( Name "x", Value True,
                         If ( Name "x", Value True,
                              Value False, ...), ...))))
```

This would be translated to the string below to be interpreted by `parseDecList`. Note that the fragment of `Behaviour` listed above would produce a considerably more verbose string, as it does not optimise on generated internal (`out`) wires. Shown before the translation is `DELAY_BEH#`, a LAMBDA macro expressing that the relation between `d_in, d_out` is a delay. Also shown is the datatype `values` that represents the values a circuit manipulates in LAMBDA. `IF_BEH#` describes the behaviour of the If construct (a form of multiplexor). Intuitively it means that if the output of the first box (`out_e1`) is equal to the chooser then we select the second box, if `out_e1` is not equal to the chooser output the second box, and if we can neither match nor not match output a bottom value.[12]

```
datatype values = True | False | Undef;
val DELAY_BEH# ( n, v, d_out, d_in ) =
    forall t.  ( ( 0 < t /\ t <= n ) ->> d_out t === v ) /\
               ( d_in t === d_out ( t + n ) );
val IF_BEH# ( out_e1, out_e2, out_e3, out, ch, bottom ) =
    ( match# ( ch, out_e1 ) === True ->> out === out_e2 ) /\
    ( match# ( ch, out_e1 ) === False ->> out === out_e3 ) /\
    ( match# ( ch, out_e1 ) === Undef ->> out === bottom )

val CIRCUIT#(inp,out) = forall t.
    exists d_in,d_out.  DELAY_BEH#(1,Undef,d_in,d_out) /\
                        exist x.  x === inp /\
                                  ( exists out_e3.
                                    IF_BEH#(x,True,False,out_e3,...) ) /\
                                    IF_BEH#(x,True,out_e3,d_in,...) /\
                        out === d_out;
```

We could now reason about the behaviour of the circuit using this macro. For example, we could show that this circuit exhibits the same behaviour as an exclusive or gate with a unit delay, initialised to undefined. Thus assuming

```
val XOR#(in1,in2,out) = out === ( in1 && ~in2 ) || ( ~in1 && in2 );
val UNIT_DELAY#(init,in1,out) = forall t.in1 t === out (t+1) /\
                                          out 1 = init;
val SPEC#(in1,in2,out) = exists x.XOR#(in1,in2,x) /\
                                  UNIT_DELAY#(Undef,x,out);
```

we would type:

[12] This is needed to secure monotonicity of ELLA constructs; see the ELLA reference manual [Pra86a] section 7.3.3.

```
setgoal Penv "G // CIRCUIT(inp,out) $ H |- inp === (in1,in2) /\
                                           SPEC#(in1,in2,out)"
```

to start proving that the circuit implements the specification.

The reader will have noticed that the above discussion was fairly fuzzy, for example the use of `~`, `&&` and `||` in the definition of `XOR` above; these are defined on Type[13] `bool` only in LAMBDA. We would have to define similar operators on the type `values` defined above.

It would be no problem to take up the suggestion to provide more than one *Behaviour* function, stressing different views of a circuit description, as each version would be independent from the others.

Embedding the *Behaviour* functions in the proof system, as outlined in section 2.1, could pose some problems. In the case of LAMBDA, the ML that may be reasoned about is more limited than the ML the functions are likely to be written in. More problematic would be the fact that they may use things like environments to keep track of variables used in formulae. This forces us to formalise these quite powerful (sets of) functions too.

References

[Bar84] Harry G Barrow. Verify: A program for proving correctness of digital hardware designs. *Artificial Intelligence*, 24:437–491, 1984.

[BG87] Graham Birtwistle and Brian Graham. new_theory 'HOL';; An introduction to hardware verification in higher order logic. Draft technical report, Computer Science Department, University of Calgary, October 1987. Draft 1.2.

[BGL+85] Mario R Barbacci, Steve Grout, Gary Lindstrom, Michael P Malony, Elliot I Organick, and Don Rudisill. Ada as a hardware description language: An initial report. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 272–302, Amsterdam, 1985. North Holland.

[BP87] D Borrione and J L Paillet. An approach to the formal verification of VHDL descriptions. Technical Report RR 683-I-, IMAG/ARTEMIS, November 1987.

[Bry85] Randal E Bryant. Can a simulator verify a circuit? In G Milne and P A Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 125–136, Amsterdam, 1985. North Holland.

[CG86] Avra Cohn and Mike Gordon. A mechanised proof of correctness of a simple counter. Technical Report 94, University of Cambridge Computer Laboratory, July 1986.

[13] Use Type to denote LAMBDA types and type to denote ML types.

[Coh87] Avra Cohn. A proof of correctness of the VIPER microprocessor: The first level. In Graham Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71, Boston, 1987. Kluwer Academic Publishers. Also as University of Cambridge Computer Laboratory report number 104.

[Coh88] Avra Cohn. Correctness properties of the VIPER block model: The second level. In G Birtwistle and P A Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 1–91, New York, 1988. Springer Verlag. Also as University of Cambridge Computer Laboratory report number 134.

[Cul87] W J Cullyer. Implementing safety-critical systems: The VIPER microprocessor. In Graham Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 1–25, Boston, 1987. Kluwer Academic Publishers.

[DM87] Bruce S Davie and George J Milne. Contextual constraints for design and verification. In Graham Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 257–265, Boston, 1987. Kluwer Academic Publishers.

[FFM89] Mick Francis, Simon Finn, and Ellie Mayger. *Reference Manual for the Lambda System.* Abstract Hardware Limited, version 2.0 edition, October 1989.

[FTMO85] Masahiro Fujita, Hidehiko Tanaka, and Tohru Moto-Oka. Logic design assistance with temporal logic. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 129–138, Amsterdam, 1985. North Holland.

[GH] M J C Gordon and J Herbert. A formal hardware verification methodology and its application to a network interface chip. Technical Report 66, University of Cambridge Computer Laboratory.

[Goo89a] K G W Goossens. An operational semantics for a subset of the HDDL ELLA. Version 2.0, June 1989.

[Goo89b] K G W Goossens. Semantics for picoELLA. June 1989.

[Gor] Mike Gordon. LCF_LSM. Technical Report 41, University of Cambridge Computer Laboratory. Second Printing with Corrections and Additions.

[Gor85] Mike Gordon. Why higher-order logic is a good formalisation for specifying and verifying hardware. In G Milne and P A Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 153–177, Amsterdam, 1985. North Holland.

[Gor87] Michael J C Gordon. HOL: A proof generating system for higher-order logic. In Graham Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128, Boston, 1987. Kluwer Academic Publishers.

[HD85a] F K Hanna and N Daeche. Specification and verification using higher-order logic. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 418–433, Amsterdam, 1985. North Holland.

[HD85b] F K Hanna and N Daeche. Specification and verification using higher-order logic: A case study. In G Milne and P A Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 179–213, Amsterdam, 1985. North Holland.

[Her85] John Herbert. The application of formal specification and verification to a hardware design. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 434–445, Amsterdam, 1985. North Holland.

[HMT89] Robert Harper, Robin Milner, and Mads Tofte. The definition of standard ML version 3. LFCS Report Series ECS-LFCS-89-81, LFCS, Department of Computer Science, University of Edinburgh, May 1989.

[Hun85] Warren A Hunt, Jr. FM8501: A verified microprocessor. Technical Report 47, Institute for Computing Science. The University of Texas at Austin, December 1985. Dissertation.

[Ins88] The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY10017 USA. *IEEE Standard VHDL Language Reference Manual*, IEEE std 1076-1987 edition, 1988.

[JBG86] Jeff Joyce, Graham Birtwistle, and Mike Gordon. Proving a computer correct in higher order logic. Technical Report 100, University of Cambridge Computer Laboratory, December 1986. HOL version of Technical Report 42.

[Joy87] J J Joyce. Formal verification and implementation of a microprocessor. In Graham Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157, Boston, 1987. Kluwer Academic Publishers.

[Mel87] Thomas F Melham. Abstraction mechanisms for hardware verification. In Graham Birtwistle and P A Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 267–291, Boston, 1987. Kluwer Academic Publishers.

[Mey89] Ernest Meyer. VHDL opens the road to top-down design. *IEEE Computer Design*, pages 57–62, February 1989.

[Mil85a] George J Milne. Simulation and verification: Related techniques for hardware analysis. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 404–417, Amsterdam, 1985. North Holland.

[Mil85b] George J Milne. Towards verifiably correct VLSI design. In G Milne and P A Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 1–22, Amsterdam, 1985. North Holland.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[Mos83] Ben Moszkowski. A temporal logic for multi-level reasoning about hardware. In T Uehara and M Barbacci, editors, *CHDL 83: 6th International Symposium on Computer Hardware Description Languages and their Applications*, pages 79–90, Amsterdam, 1983. North Holland.

[MPT85] J D Morison, N E Peeling, and T L Thorp. The design rationale of ELLA, a hardware design and description language. In C J Koomen and T Moto-Oka, editors, *CHDL 85: 7th International Symposium on Computer Hardware and Description Languages and their Applications*, pages 303–320, Amsterdam, 1985. North Holland.

[Nar88] Paliath Narendran. Formal verification of the Sobel image processing chip. In G Birtwistle and P A Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 92–127, New York, 1988. Springer Verlag.

[Par87] Joachim Parrow. Verifying a CSMA/CD-protocol with CCS. LFCS Report Series ECS-LFCS-87-18, LFCS, Department of Computer Science, University of Edinburgh, January 1987.

[Pau85] Lawrence C Paulson. Interactive theorem proving with Cambridge LCF. Technical Report 80, University of Cambridge Computer Laboratory, November 1985.

[PB85] Robert Piloty and Dominique Borrione. The Conlan project: Concepts, implementations and applications. *IEEE Computer*, pages 81–92, February 1985.

[Pez87] Mauro Pezzé. Behavioural abstraction and circuit verification using CIRCAL. Internal Report CSR-251-87, Department of Computer Science, University of Edinburgh, November 1987.

[Pra86a] Praxis Systems plc, 20 Manvers Street, Bath BA1 1PX. *The ELLA Language Reference Manual*, issue 3.0 edition, 1986.

[Pra86b] Praxis Systems plc, 20 Manvers Street, Bath BA1 1PX. *The ELLA Tutorial*, issue 3.0 edition, 1986.

[SLM+85] Moe Shahdad, Roger Lipsett, Erich Marschner, Kellye Sheenan, Howard Cohen, Ron Waxman, and Dave Ackley. The VHSIC hardware description language. *IEEE Computer*, pages 94–103, February 1985.