Computer Science
University of Manchester

A Process Algebra Foundation for Reasoning about Core ELLA

Howard Barringer, Graham Gough,

Brian Monahan, Alan Williams

Technical Report UMCS-94-12-1

FORMAL VERIFICATION SUPPORT FOR ELLA

IED 4/1/1357

A Process Algebra Foundation for Reasoning about Core ELLA*

Howard Barringer Graham Gough Brian Monahan† Alan Williams ‡

Department of Computer Science§
University of Manchester
Oxford Road, Manchester, UK.
email: howard@cs.man.ac.uk

December 1994

Recent technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from ftp.cs.man.ac.uk in the directory pub/TR. The files are stored as PostScript, in compressed form, with the report number as filename. They can also be obtained on WWW via URL http://www.cs.man.ac.uk/csonly/cstechrep/index.html. Alternatively, all reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

†Brian Monahan is now with Harlequin Ltd., Alderley Edge

‡Alan Williams acknowledges additional support from DRA (Malvern) and Harlequin Ltd (Cambridge)

§The University of Manchester acknowledges support from the SERC via research grant GR/F 38174.

Abstract

A process algebraic foundation is developed, for formal analysis of synchronous hardware designs using the commercially available hardware design language, ELLA. An underlying semantic foundation, based on input/output trace sets, is presented first through the use of state machines. Such a representation enables direct application of standard, fully automated, trace equivalence checking tools. However, to overcome the computational limitations imposed by such analysis methods, the input/output trace semantics is re-presented through a synchronous process algebra, EPA. Primitive processes in EPA denote the behaviour of primitive hardware components, such as delays or multiplexers, with composition operators corresponding to the different ways in which behaviours may be built. Of particular significance is the parallel composition operator which captures the machinery for building networks from other components/networks. Actions in EPA are structured and signify the state of input and output signals. This structure, however, is abstracted by developing an algebra for the actions. In particular, parallel composition on processes neatly lifts to a special (synchronous) product operation on actions.

The EPA representation forms a good basis for semi-automated high-level symbolic manipulation and reasoning tools. Firstly, the original design structure can be maintained, thus easing the problems of user level feedback from tools. Secondly, the application of EPA to ELLA enables a *normal form* for EPA terms. This provides a route to tractable symbolic verification and simulation, using a *state evolution* method to establish strong bisimulation properties. The method has been successfully applied to classes of unbounded state space systems.

1 Introduction

This paper is one of a series describing different aspects of work on the development of design and formal verification environments for commercial hardware description languages (HDLs). The HDL addressed here is ELLA[MC93]. The techniques described, however, have broader applications, for example, it would be possible to apply them to other HDLs, such as VHDL, in future work.

The work reported here focusses initially on the development of a formal semantics for ELLA (the starting point for any formal analysis tools). We show how a state-based approach using *input/output automata* (IOAs) as semantic representations of ELLA expressions is developed as a structured symbolic intermediate-level process algebraic approach using *ELLA Process Algebra* (*EPA*). We concentrate on showing how ELLA expressions are translated into semantic objects represented by *EPA* terms, which are suitable for further formal analysis.

Our principle rationale for developing the semantics is of course to provide a foundation for formal reasoning tools. In general, the IOA approach suffers from a combinatorial explosion in the complexities of representation and verification as the size of system increases (even if encodings such as Binary Decision Diagrams (BDDs) are used), hence the need to develop our process algebraic approach. An outline of a truly symbolic verification method for determining trace equivalence between *EPA* representations of ELLA expressions is included. The approach generates a set of verification conditions (VCs) that, if provable, guarantees equivalence of the two given ELLA designs. The VCs are expressed in a many-sorted first order predicate logic. A prototype of such a verification condition generator (VCG) has been implemented that automatically simplifies as many VCs as possible (without implementing a full first order theorem prover/proof assistant) and then submits those that are non-trivially true or false to a proof assistant (in our case PVS[SOR93]) for proof. One advantage of such an approach over other symbolic model checking methods, such as BDDs, is the ability to handle trace equivalence of infinite state space designs, either fully automatically or with minimal user interaction.

A full description of the verification approach is given in another paper in the series[BGMW94a]. A reference manual containing the complete semantic definition of ELLA using *EPA* is also available as a technical report[BGMW94b].

1.1 Background

Increasingly sophisticated hardware design environments are becoming more readily available to the hardware engineer. Typically, these are equipped with hardware description languages (HDLs) for design representation, and schematic capture and other editing tools for design entry and manipulation. Library and versioning tools provide for design management. Simulation tools provide sophisticated display and manipulation facilities for input and output data, allowing the user to monitor and force values on circuit nodes. The tool suite is fully integrated and accessed via a graphical user interface. Such environments go a long way in assisting designers in their task — the fast production of 'right-first-time' hardware systems. However, they do not generally provide formally-based verification and analysis tools, the development of which still lies chiefly in the research domain. The work presented here lays the foundations for such tools and their integration into design environments already familiar to the hardware engineer.

Our approach has been to work in the context of a commercial HDL, namely ELLA[MC93], so that the means of design representation is familiar to the designer, who can still use existing CAD support tools. The ELLA design and verification environment currently under development includes many of the facilities provided by conventional environments listed above. The crucial differences here are that all of the design analysis tools, including the simulator, operate from the formal semantic representation of the original ELLA design, and that there are additional tools for formal design transformation and equivalence-testing. The user still interacts with the design environment at the ELLA level, whilst the environment takes care of maintaining the underlying semantic representation. For example, if a user requests a simulation on a modified design, then the environment automatically compiles the ELLA expression before invoking the simulation tool, which operates on the updated semantic representation.

In order to provide a formal foundation to the appropriate verification methods, it has therefore first been necessary to define a formal semantics for ELLA. An extensively automated approach to design reasoning and verification is then required, so that a designer, who is not necessarily an expert in formal methods, can operate the tools. It is also essential to ensure that the designer can interact with these tools at a familiar level — for example, any "debugging" feedback from verification attempts should be presented back to the user at the level of the original design description.

1.1.1 ELLA

ELLA is a hardware design system created by DRA Malvern for the high-level description of complex VLSI designs[MC93, MPT85], then developed as a commercial product by Praxis Systems[Com91] and now available in the public domain as ELLA2000[1]. The system includes the ELLA language and compiler, an application support environment, and simulation and other tools. ELLA is thought of as a parallel language in which user defined processes/functions are used to model the behaviour of hardware structures. Users define datatypes, to model signal values, and then build up complex systems by connecting together previously defined functions. ELLA uses a simple timing model, and is effectively deterministic, meaning that a deterministic state-based semantic model can be used with standard decidable equivalence procedures. Although the language is syntactically large, all complex features in ELLA can

[1]ELLA is currently marketed by R-Cube Systems, Bristol, England, and available by anonymous ftp from Imperial College, London, in src.doc.ic.ac.uk:packages/ELLA

be reformulated in terms of a small set of basic features, known as Core ELLA. This reduces the amount of work required in providing a formal semantics for the whole language.

1.2 Overview

The behaviour, or semantics, of an ELLA construct is given by a set of input/output traces. The first approach to representing this trace semantics uses input/output automata (referred to hereafter as IOAs) as semantic carriers, i.e. representations of semantic objects, and is outlined below in §2. This approach is attractive since it is intuitive, compositional, and enables standard automated equivalence-checking and model-checking tools to be applied. As is well known however, the representation suffers an exponential growth in size and complexity that renders it unsuitable for large-scale verification. There is some hope in the form of special encoding of the IOAs, such as binary decision diagrams (BDDs). Even so, these currently have only a limited domain of applicability. An IOA is also a low-level object — any structure or abstraction in the original HDL design representation is lost in its semantic representation, since the IOA simply denotes the *flat* input/output behaviour or language of its corresponding design construct. This restricts the use of any user-level feedback information that can be provided by analysis tools.

In §3 we present the *ELLA Process Algebra* (*EPA*), which provides the basis for developing a truly symbolic representation for the semantics of ELLA, that lifts the level at which the semantic objects operate and thereby retains design structure. This leads to a very compact semantic representation of an ELLA design at an intermediate-level, that can support high-level symbolic simulation and reasoning.

Section 4 presents the *EPA* semantic definitions for ELLA expressions, and shows how they are translated into *EPA* terms. This compilation process is illustrated in §4.6, where results from a prototype implementation of the tools are described. The *EPA* terms resulting from this translation can be processed further to produce *normal form* terms — these provide the basis for simulation tools and symbolic verification techniques. Section 5 introduces normalisation, and then briefly describes symbolic simulation and *symbolic verification*. The latter is based on a *state bisimulation* rule which is equivalent to Milner's (strong) bisimilarity. For ELLA, where designs are essentially deterministic, bisimulation is just (IO) trace equivalence. An accompanying *state evolution* method shows how state bisimulation can be established, by generating a set of verification conditions from a *symbolic bisimulation* of the *EPA* terms under analysis — if the verification conditions are valid, then the *EPA* terms are bisimilar.

1.3 Related Work

There has been extensive research into state-based verification methods such as model-checking[CES86, BFG89]. Although these have been traditionally restricted by the problem of 'state-explosion', the capabilities of these techniques have been dramatically enhanced by the introduction of compact state-space encodings, namely BDDs — see [BCL91, BCMD90, CBM89] for key expositions.

The process algebraic approach described here has its origins in the foundational works of CCS[Mil89] and CSP[Hoa85]. There have recently emerged techniques for efficiently modelling systems; for example [HL93] describes a value-passing process algebra, where the structured actions consist of channels which can carry messages or data.

There have been other recent approaches to providing a formal semantics for ELLA. At Edinburgh, operational semantics are provided for two simplified versions of ELLA, called *micro*ELLA[Goo90a] and *pico*ELLA[Goo90b]. At Cambridge, a system was being developed to *semantically embed* ELLA into HOL[BGHT90], so that the HOL verification system and its associated tools can be employed.

Our work on foundations and verification support has formed part of an on-going collaborative project, involving Manchester, DRA (Malvern), Harlequin Ltd. (Cambridge), and GEC Plessey Semiconductors (Swindon), and sponsored by the DTI and SERC. Work from the project is documented in [GB91, BGL+91, BGL+92, BGMW94b]; a World-Wide Web page containing an overview of the project, and a full list of project reports is also available[EP].

Furthering work reported in this paper, Formal Systems (Europe) Ltd. were commissioned to indicate how Receptive Process Theory could be developed to be used as a representation for a subset of the ELLA semantics.

1.4 Acknowledgements

We would like to thank John Morison, Mike Hill, Peter Ryan and Richard Moore (DRA, Malvern) for their useful comments on earlier drafts of this paper, and helpful interaction on earlier versions of our semantics work. Additionally, we thank Chris Tofts (University of Manchester) for his useful comments on the paper.

2 Semantic Approach

The semantics of synchronous HDLs such as ELLA can be represented by input/output trace sets due to the straightforward timing model of the HDLs, their effectively deterministic behaviour, and simultaneous input/output. In ELLA, every design element is assumed to have zero delay, except for explicit delay elements, which have integer delays.

In turn, a trace set can be represented using a structure similar to a standard Mealy machine from automata theory [HU79, Shi87]. These ideas are slightly adapted and generalised here to form an input/output automaton (IOA), which has some similarity to the structure of the same name studied by [LT88]. The set of input/output traces, or IO language giving the behaviour of an ELLA construct, is then represented by an IOA.

For the purpose of defining the semantics of hardware components, an IOA is taken to be a 5-tuple comprising a set *states* of states, an input alphabet, an output alphabet, an input/output-labelled transition relation *trans*, and a distinguished set *start* of states — the initial states. All infinite runs of an automaton will be *accepting*. Invariants on the structure of an IOA are posed to ensure that the start states are genuine states of the IOA and that outputs may not also be inputs. Furthermore, the input and output mappings for each transition must have appropriate domains and the 'from' and 'to' states must be in the state set.

The semantics of basic hardware components are represented by atomic IOAs; these are then composed using a specialised product operation, together with input/output renaming and output hiding, to give the semantics of more complex constructs. The product needs to be 'synchronous' so that the resulting automaton only produces a transition when each of the participants can make a transition of its own. In particular, if an output channel of one automaton is 'connected' to an input on the other, then only those transitions where the values on that channel agree are permitted in the resulting automaton. Furthermore, any such channel is hidden within the resulting product automaton — it still remains as an output, however.

Verification for IOA semantic objects requires the application of well-known language equivalence and containment algorithms[Moo56, RS59]. If two automata are inequivalent, then it is possible to generate a counter-example or difference automaton that, when simulated, shows how the trace sets differ. Note also that verification occurs at the semantic object level; for IOAs, this is effectively at the level of the IO languages of the hardware components — low-level and divorced from the original design description constructed by the hardware engineer. All of the structure inherent in the design has been lost, making structured, modular, high-level design analysis almost impossible. This also creates problems when attempting to provide a designer with useful debugging feedback if a design fails under some analysis. However, the state-based verification approach has the great advantage that it can be fully automated. Of course there is still the problem of state-explosion, even when using encodings such as BDDs.

An early prototype compiler was implemented, based on the IOA semantics for a subset of Core ELLA called Boolean Kernel ELLA (BKE), using an enumerated representation of the IOAs. A simulator, equivalence-checker and counter-example generator were also built, and installed within a prototype design environment developed by Harlequin Ltd[BGL+91, BGL+92]. The environment contains a design library facility, and textual and schematic editors, all embedded within a graphical user interface. Also included was a BDD-encoded version of the equivalence checker[Pei92].

2.1 IO Automaton Examples

The IOA representation of the semantics of synchronous hardware systems is not fully described here; instead three examples of IOA representations of simple hardware components are given.

A typical trace for a standard NOR gate component is shown in Table 1(a), where it has inputs x, y and output $o1$, all ranging over the boolean input/output alphabet $\{\mathtt{f},\mathtt{t}\}$. At each step in the trace, the output is the logical nor of the inputs. The complete trace set or behaviour of the NOR gate is represented by an IOA with four transitions — its transition table is given in Table 1(b). The IOA has the same input

Time	0	1	2	3	4	5	6	···
x	t	f	t	f	f	f	t	···
y	f	t	t	f	t	f	t	···
o1	f	f	f	t	f	t	f	···

(a)

From	Input		Output	To
State	x	y	o1	State
n	f	f	t	n
n	f	t	f	n
n	t	f	f	n
n	t	t	f	n

(b)

Table 1: NOR Gate: (a) Trace, (b) Transition Table

and output labels, x, y and $o1$ respectively, a state set $states = \{n\}$ with a single state, initial state set $start = \{n\}$, and boolean input/output alphabet $Val = \{\mathtt{f},\mathtt{t}\}$. It is clear that the output on any transition is the logical nor of the two inputs, as required. For example, the first entry in Table 1(b) is interpreted as a labelled transition $n \xrightarrow{x(\mathtt{f}),y(\mathtt{f})/o1(\mathtt{t})} n$ going from state n to itself, where both input labels x, y are set to f and the output $o1$ is set to t.

A unit delay models delays in hardware, where the input is delayed by one (notional) step. A typical trace is shown in Table 2(a), with input z and output $o2$ — at any time t, the output is equal to the value input at time $t-1$, with the initial output value f. The complete trace-set is represented by a two-state automaton, shown in Table 2(b). Let state 0 be the initial state. Consider the possible transitions: starting in the initial state, if the input is f, then the output is f and the state does not change. If, however, the input had been t, then f would again have been output but the new state would have been state 1. The idea is that state 0 corresponds to having input the value f on the previous transition, i.e. all incident transitions have input

Time	0	1	2	3	4	5	6	⋯
z	f	t	f	t	t	f	f	⋯
o2	f	f	t	f	t	t	f	⋯

(a)

From State	Input z	Output o2	To State
0	f	f	0
1	f	t	0
0	t	f	1
1	t	t	1

(b)

Table 2: Unit Delay: (a) Trace, (b) Transition Table

`f`, and its emergent transitions have output `f`, thus effecting a unit delay on `f` inputs. Furthermore, the incident transitions of state 1 all have input `t` and the emergent transitions have output `t`. To represent the trace in Table 2(a) the IOA would go through the state sequence 0-0-1-0-1-1-0-0 $\cdots$.

2.2 Product IO Automata

Having illustrated the semantic objects representing two elementary hardware components, we now show how to compose these, forming more complex (non-elementary) structures. Suppose we connect the unit delay onto the output of the NOR gate to form a unit-delay-NOR gate, as shown in Figure 1. This com-

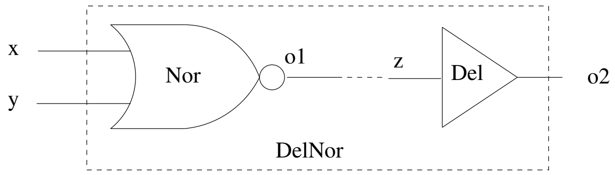


Figure 1: Delay Nor Gate

position is performed by renaming the unit-delay IOA input z to $o1$, so that the two IOAs are 'connected', taking the product of this new automaton with the NOR gate IOA, and then 'hiding' the connecting channel $o1$. Intuitively, the IOA product is formed by first taking all combinations of the four transitions from both the NOR gate and the (renamed) unit-delay automata, resulting in a set of 16 pairs[2]. Then conflicting transition pairs are removed from this intermediate transition table, and new transitions are constructed from the remaining pairs. Table 3(a) shows four examples of transition pairs, after the unit delay input z has been renamed to $o1$ — notice that the state component has been omitted from the NOR gate transitions, because it is always n. The first two transition pairs have conflicting values for the communicating channel $o1$, and therefore will not appear in the final transition table. The second two transition pairs are valid since the values for $o1$ are in agreement. For each of these, a new transition for the product IOA is constructed. The new *from* and *to* state labels are formed by combining the component state labels, for example by pairing them. However in this case, the NOR gate only has a single state, so the new state labels are formed by simply copying the states from the unit-delay transition. The new transition label is now calculated: the new output set is the union of the component outputs, i.e. $o1$ and $o2$. The new input set is also the union of the component inputs, except that any inputs that also appear as outputs are removed since they are now redundant — this simply results in the inputs x, y because the unit-delay input $o1$ is removed. Each input/output takes the same unique value found in the component transitions.

[2] In our implementation of this particular product operation, the complete transition tables are not constructed in this way. It is presented in this way purely for explanatory purposes.

Under a separate *hiding* operation — not part of the IOA product — the output $o1$ is removed from the output/transition-sets. Table 3(b) shows the resulting transitions for the unit-delay NOR gate IOA, where the first two entries correspond to the two undeleted transitions shown in Table 3(a). The resulting IOA

Nor Gate			Unit Delay				
Inputs		Output		Input	Output		
x	y	o1	From State	z o1	o2	To State	Conflict Removal
f	f	t	1	f	t	0	deleted
f	t	f	0	t	f	1	deleted
f	f	t	0	t	f	1	included
f	t	f	0	f	f	0	included

(a)

From State	Input x	Input y	Output o2	To State
0	f	f	f	1
0	f	t	f	0
0	t	f	f	0
0	t	t	f	0
1	f	f	t	1
1	f	t	t	0
1	t	f	t	0
1	t	t	t	0

(b)

Table 3: Delay-NOR Gate: (a) Example Calculation of Transitions, (b) Transition Table

is therefore a two-state machine, having input labels x, y, output label $o2$, state set $states = \{0, 1\}$, initial state set $start = \{0\}$, and value set $Val = \{\mathtt{f}, \mathtt{t}\}$.

A typical trace for the delay-NOR gate is shown in Table 4; to represent this the delay-NOR IOA would go through the state sequence 0-0-1-1-0-1-0-0 $\cdots$.

Time	0	1	2	3	4	5	6	$\cdots$
x	t	f	f	f	f	t	f	$\cdots$
y	f	f	f	t	f	f	t	$\cdots$
o2	f	f	t	t	f	t	f	$\cdots$

Table 4: Trace for Delay-NOR Gate

3 Abstracting from IO Automata

The state-based compositional approach for representing the semantics illustrated above is an attractive one for a hardware description language such as ELLA because it captures the physical concept of creating complex designs by connecting together smaller units. Indeed, the exercise in developing a semantic description in the first instance for BKE, and then for Core ELLA, proved extremely valuable to both ourselves and some of the original designers of ELLA in giving a better understanding of the semantic issues. However, this composition process destroys the hierarchical structure of the original design, as expressed in ELLA, and the semantics reduces to a single (unstructured) set of transitions. Reasoning about the design at a symbolic level, or performing design transformations, e.g. via rewriting, from this description, is thus made particularly difficult. The IO automaton model also suffers from the problem of exponential growth in the size of the state/transition-sets required — even standard functions, such as multipliers, demand non-trivial representation, because of the combinatorial explosion. The representation is not well-suited to state-space reducing techniques such as data/control separation or data-set reduction.

However, if the effects of state-explosion can be avoided, then state-based representations offer the chance for fully automatic verification methods, both for equivalence-checking and for proving temporal and other logical properties about a design. There is clearly a need to devise a mixed symbolic state-based representation of the automaton — not just an encoding of the transition relation or state set using for example BDDs — so that tractable higher level reasoning and transformation tools can be developed. This has led to the development of a representation of the ELLA semantics in a process algebra, *EPA*, which can be viewed as a symbolic representation of the underlying input/output trace set.

3.1 Representing Hardware Components using *EPA*

EPA is a process algebra designed specifically for representing the behaviour of synchronous hardware systems described in ELLA[MC93, Com91], although it does have wider applications. *EPA* has some similarities to SCCS[Mil89]. An *EPA* process term evolves into a new term by performing *actions*, the evolution determined by a set of transition rules, with respect to an *EPA context*, or set of *EPA* definitions. An action in *EPA* is structured; it associates value expressions with input and output *channels*. The inputs and outputs occur simultaneously, instead of being separate actions. Process calls can also take *parameters* for values and channel names, thus providing a mechanism for value-passing between process terms, and for renaming channels.

The channel names effectively provide labels for values, which must belong to a particular type. Processes are composed using a parallel composition operation, which combines component actions via an action product. This is defined so that if an output channel in one component has the same name as an input channel in another component, then a new valid action can only be formed when the channels are associated with equal values. In this case the input channel is removed, its value now being 'supplied' by the output. This product models synchronous communication between processes, and corresponds to the IOA product presented above. Clearly, there must be restrictions on the names that channels can take in a valid action: outputs must all have distinct names, and the sets of inputs and outputs must be disjoint. An action can also be modified by hiding certain channels.

The three examples discussed in §2, namely the NOR gate, unit-delay and delay-NOR gate, are used again here to illustrate how hardware components are represented in *EPA*. First, the (delay-less) NOR-gate of §2.1 can be represented by a process Nor_P defined by:

$$\textsf{process}\ Nor_P(c_1, c_2\colon bool\ /\ c_3\colon bool)\ \underline{\triangle}$$
$$\qquad\qquad \Box\, a, b\colon bool\ \cdot\ c_1(a), c_2(b)\ /\ c_3(nor(a,b)) :: Nor_P(c_1, c_2\ /\ c_3)$$
$$\textsf{function}\ nor(a,b\colon bool) : bool\ \underline{\triangle}\ \textsf{case}\ (a,b) : (\texttt{f},\texttt{f})\ \rightarrow\ \texttt{t}\ \textsf{else}\ \texttt{f}\ \textsf{end}$$
$$\textsf{type}\ bool\ \underline{\triangle}\ \{\texttt{f},\ \texttt{t}\}$$

The left hand side of the process definition declares[3] two input channel variables c_1, c_2 and an output channel c_3 which range over channels of type *bool*; when a call to Nor_P is made, then c_1, c_2, c_3 will be

[3]An *EPA* process definition can also take an optional sequence of value-variable declarations, separated from the channel declarations by a ';' — see §3.2. If there are no value-variables, as in this case, then the semi-colon is dropped. Note also that the input and output channel declarations are separated by a ' / ', which can be dropped if there are no inputs. Each of the three declaration sequence in the process definition consists of a comma-separated list of elements.

substituted by the actual channel parameters used in the call (see below). A call to the process will have the following behaviour: the part ‘$\Box\ a, b$: *bool* $\bullet$ ’ means that the values substituted for a, b are to be non-deterministically chosen from the set *bool*. These are then used in the remaining term, which has a *prefix* action $c_1(a), c_2(b)\ /\ c_3(nor(a,b))$ and a *continuation* $Nor_P(c_1, c_2\ /\ c_3)$. The prefix action is performed before proceeding to the continuation. The values chosen for a, b are bound to the input channels substituted for c_1, c_2 — the process is *input deterministic*, because there is a unique output value and continuation for every input value. The value given by the function $nor(a,b)$ is then placed on the output channel substituted for c_3 — the case statement in this function definition clearly evaluates to the logical ‘nor’ of its two arguments. The continuation process is just a call to itself. Finally, the *EPA* type definition *bool* represents the usual two-valued type $\{\mathtt{f}, \mathtt{t}\}$. The following sequence of (ground) transitions could result from a call $Nor_P(x, y\ /\ o1)$, where $x, y, o1$ represent concrete channels of type *bool*, with distinct channel names:

$$Nor_P(x,y\ /\ o1) \xrightarrow{x(\mathtt{t}), y(\mathtt{f})\ /\ o1(\mathtt{f})} Nor_P(x,y\ /\ o1) \xrightarrow{x(\mathtt{f}), y(\mathtt{t})\ /\ o1(\mathtt{f})}$$

$$Nor_P(x,y\ /\ o1) \xrightarrow{x(\mathtt{t}), y(\mathtt{t})\ /\ o1(\mathtt{f})} Nor_P(x,y\ /\ o1) \xrightarrow{x(\mathtt{f}), y(\mathtt{f})\ /\ o1(\mathtt{t})}$$

$$Nor_P(x,y\ /\ o1) \xrightarrow{x(\mathtt{f}), y(\mathtt{t})\ /\ o1(\mathtt{f})} Nor_P(x,y\ /\ o1) \xrightarrow{x(\mathtt{f}), y(\mathtt{f})\ /\ o1(\mathtt{t})}$$

$$Nor_P(x,y\ /\ o1) \xrightarrow{x(\mathtt{t}), y(\mathtt{t})\ /\ o1(\mathtt{f})} Nor_P(x,y\ /\ o1)$$

The sequences of input/output values obtained was chosen to coincide with those given in Table 1(a).

This expansion down to ground terms may appear to gain little over the IOA representation. However, it is the compact *symbolic* version which is of interest — this captures all of the required information: at each step a process Nor_P evolves into an identical process via an action, in which the output is the ‘nor’ of the two inputs. The advantages of this structured, compact, representation will become apparent when considering more complex expressions.

A unit delay for data of any type *ty*, where *ty* is a variable ranging over possible concrete datatypes, can be represented by a *generic* process definition $Udel_{[ty]}$ which has a ‘state’ variable w:

$$\text{process } Udel_{[ty]}(w\colon ty\ ;\ z\colon ty\ /\ o2\colon ty) \triangleq \Box\ v\colon ty\ \bullet\ z(v)\ /\ o2(w) :: Udel_{[ty]}(v\ ;\ z\ /\ o2)$$

At every transition, $Udel_{[ty]}$ chooses a value $v \in ty$ for input channel z, and binds the current state value w to output $o2$. It then evolves into a recursive call, using the value v just received on its input as the new ‘state’; this will be output at the next transition, thus effecting a unit delay. For example, a unit delay for the booleans *bool* with initial value `f`, can be represented by the process definition Del_P:

$$\text{process } Del_P(c_1\colon bool\ /\ c_2\colon bool) \triangleq Udel_{[bool]}(\mathtt{f}\ ;\ c_1\ /\ c_2)$$

where the process $Udel_{[bool]}$ is called with $w = \mathtt{f}$. Again, notice that c_1, c_2 are channel *variables* ranging over channels of type *bool*. A typical transition sequence represented by a process call $Del_P(z\ /\ o2)$, where $z, o2$ represent concrete channels of type *bool* with distinct channel names, would be as follows:

$$Del_P(z\ /\ o2) \xrightarrow{z(\mathtt{t})\ /\ o2(\mathtt{f})} Udel_{[bool]}(\mathtt{t}\ ;\ z\ /\ o2) \xrightarrow{z(\mathtt{t})\ /\ o2(\mathtt{t})} Udel_{[bool]}(\mathtt{t}\ ;\ z\ /\ o2) \xrightarrow{z(\mathtt{f})\ /\ o2(\mathtt{t})}$$

$$Udel_{[bool]}(\mathtt{f}\ ;\ z\ /\ o2) \xrightarrow{z(\mathtt{t})\ /\ o2(\mathtt{f})} Udel_{[bool]}(\mathtt{t}\ ;\ z\ /\ o2) \xrightarrow{z(\mathtt{t})\ /\ o2(\mathtt{t})} Udel_{[bool]}(\mathtt{t}\ ;\ z\ /\ o2)$$

$$\xrightarrow{z(\mathtt{f})\ /\ o2(\mathtt{t})} Udel_{[bool]}(\mathtt{f}\ ;\ z\ /\ o2) \xrightarrow{z(\mathtt{f})\ /\ o2(\mathtt{f})} Udel_{[bool]}(\mathtt{f}\ ;\ z\ /\ o2) \cdots$$

This corresponds to the trace given in Table 2. Again, the advantage of the compact representation becomes evident for more complex structures, for example if the cardinality of the data set *ty* is large.

To represent a unit-delay-Nor gate, the two process terms Nor_P and Del_P must be combined to evolve in synchrony using a composition operator ' | '. A common channel c_4 is introduced for the output of Nor_P and the input of Del_P, which is then hidden, indicated by the part of the term '$(\ \cdots\ )\backslash\{c_4\colon bool\}$'.

$$\textsf{process}\ DelNor_P(c_1, c_2\colon bool\ /\ c_3\colon bool)\ \underline{\triangle}\ \Big(\ Nor_P(c_1, c_2\ /\ c_4)\ \mid\ Del_P(c_4\ /\ c_3)\ \Big)\backslash\{c_4\colon bool\}$$

At each step, the actions from the component processes are combined with the action product operator '$\star$', which forms a valid composite action only when the values bound to like channel names, i.e. c_4, are in agreement — see §3.2 below for its definition. The continuation term is then the product of the component continuation terms. For example, the following is a (short) trace represented by $DelNor_P(x, y\ /\ o2)$, where $x, y, o2$ are distinct concrete channels of type *bool*, corresponding to the first three steps in Table 4:

$$DelNor_P(x, y\ /\ o2) \xrightarrow{x(\mathtt{t}), y(\mathtt{f})\ /\ o2(\mathtt{f})} \Big(\ Nor_P(x, y\ /\ o1)\ \mid\ Udel_{[bool]}(\mathtt{f}\ ;\ o1\ /\ o2)\ \Big)\backslash\{o1\}$$

$$\xrightarrow{x(\mathtt{f}), y(\mathtt{f})\ /\ o2(\mathtt{f})} \Big(\ Nor_P(x, y\ /\ o1)\ \mid\ Udel_{[bool]}(\mathtt{t}\ ;\ o1\ /\ o2)\ \Big)\backslash\{o1\}$$

$$\xrightarrow{x(\mathtt{f}), y(\mathtt{f})\ /\ o2(\mathtt{t})} \Big(\ Nor_P(x, y\ /\ o1)\ \mid\ Udel_{[bool]}(\mathtt{t}\ ;\ o1\ /\ o2)\ \Big)\backslash\{o1\} \cdots$$

3.2 *EPA* Syntax

Having presented, informally, examples of the way *EPA* provides a compact representation of the behaviour of atomic and composite hardware components, a more detailed account of the *EPA* language is now given.

An *EPA process definition* has the following structure:

$$\textsf{process}\ pnm(vdecls\ ;\ icdecls\ /\ ocdecls)\ \underline{\triangle}\ P$$

Its left hand side (of the $\underline{\triangle}$) consists of a process name *pnm*, followed by a list of formal parameters divided into three groups: the first group $vdecls = v_1,\ \cdots\ , v_k$ are value-variable definitions which are assigned when the process is called; the second and third groups, $icdecls = i_1,\ \cdots\ , i_m$ and $ocdecls = o_1,\ \cdots\ , o_n$ are *input* and *output channel variable* declarations, respectively. In general these parameters are typed, but for the sake of clarity a single value type will be assumed in this section, and therefore omitted. Note that the sequences of value-variables and input channels may be empty, and can therefore be omitted, together with the respective separators ';' and ' / '.

The right-hand side $P \in ProcExpr$ is a *process expression* in which the process parameters may be used. It can take various forms:

- A *process call* expression '$pnm(vs \ ; \ ics \ / \ ocs \ )$' instantiates the right-hand side of a process definition $pnm(vdecls \ ; \ icdecls \ / \ ocdecls \ ) \triangleq P_1$ with actual parameters vs, ics, ocs substituted for the formal parameters $vdecls, icdecls, ocdecls$, i.e., the call evolves to

  $$P_1 \, [\, vs/vdecls, ics/icdecls, ocs/ocdecls \,]$$

  which simply denotes that in the process expression P_1, the formal arguments $vdecls, icdecls$ and $ocdecls$ have been substituted by vs, ics and ocs, respectively. Again, the value-variables vs, and inputs ics may be omitted.

  A *generic* process definition is also available (for example, the unit delay example above), so that families of process definitions can be introduced, indexed by parameters such as type.

- A *prefix* expression '$ae :: P_1$' evolves by first performing the action ae and then proceeding to the continuation P_1.

- A *process choice* expression '$\square \, x \colon \, se \ \bullet \ P_1$' evolves to one of the process expressions $P_1 \, [\, v/x \,]$ where $v \in se$ and $se \in SetExpr$ is a set of values. The value variable x is bound in P_1.

- A *summation* expression '$(P_1 + \ \cdots \ + P_{n+2})$' evolves to one of the process expressions P_i, $i \in \{1, \ldots, n+2\}$

- A *product* expression '$(P_1 \ \mid \ \cdots \ \mid \ P_{n+2})$' combines the prefixes found in P_i, $i \in \{1, \ldots, n+2\}$ using an action product operation '$\star$' and evolves to the product of the continuations, i.e., if P_i evolves to $ae_i :: P_i'$ then the product evolves to

  $$ae_1 \star \ \cdots \ \star ae_{n+2} :: (P_1' \ \mid \ \cdots \ \mid \ P_{n+2}')$$

- A *hiding* expression '$P_1 \backslash ch\text{-}set$' removes channels in the set of s channels $ch\text{-}set = c_1, \ \cdots \ , c_s$ from P_1. The channels in $ch\text{-}set$ are bound in P_1.

An *action expression* '$ae \in ActExpr$' consists of a pair of sequences of channel bindings '$a \ / \ b$', where $a = i_1(x_1), \ \cdots \ , i_m(x_m)$ and $b = o_1(e_1), \ \cdots \ , o_n(e_n)$. Here, for $j \in \{1, \ldots, m\}$, $k \in \{1, \ldots, n\}$, i_j, o_k are channel names, x_j are value variables, and e_k are value expressions. In fact, these binding sequences are simply a convenient short-hand for the *action summation* of the channel bindings, described below. For example:

$$i_1(x_1), \ \cdots \ , i_m(x_m) \ = \ i_1(x_1) \oplus \ \cdots \ \oplus i_m(x_m)$$

Value expressions are terms over an abstract data type; for *EPA*, this includes a case operator and function application.

The action product combines actions in a way that matches simultaneous communication. If two actions are inconsistent, i.e. an input of one action binds a channel name to a value that is different from that given by an output binding to the same channel name of the other action, then a *zero action* $\boxtimes$ results. Zero actions are factored out of equivalence notions, such as bisimulation defined in §5.3.

3.2.1 Transition-Based Semantics for *EPA*

An outline of the transition rules for *EPA*, together with definitions for the action product and hiding operations, is presented below. It is assumed that there is an *evaluation* for ground terms of the action algebra, which is also equipped with a characterisation of their *equivalence*. By defining an appropriate *normalisation* for the action algebra, matching of value expressions, as required here, can be performed. Action *expressions* $ae \in ActExpr$ will therefore be interpreted to produce concrete actions $a \in Act$ as follows:

$$
\begin{aligned}
a \in Act &::= io\text{-}act_1 \; / \; io\text{-}act_2 \\
io\text{-}act \in IOAct &::= bd_1, \cdots, bd_{n+1} \\
bd \in Binding &::= ce \; (\; v \;)
\end{aligned}
$$

where $ce \in ChannelExpr$, and $v \in ALL\text{-}VALUE$. The set $ALL\text{-}VALUE$ is the set of literal values for the action algebra. Note that a binding bd is now assumed to take a literal (ground) value v instead of a value *expression*. Two special actions are defined: the *zero action* $\boxtimes$ introduced above representing an illegal action for *EPA*, and an *empty action* ε, which is legal but does not contain any bindings. Actions within $Act \cup \{\varepsilon\}$ are called *proper actions* and $\boxtimes$ is an *improper action*.

A transition relation $\longrightarrow_C$ is defined for *closed process terms* — terms in which there are no occurrences of free variables. The semantics of closed terms is given as the *least* transition relation between closed terms, indexed by actions from $Act \cup \{\boxtimes, \varepsilon\}$, satisfying the following rules:

Let $a, a_1, a_2 \in Act \cup \{\boxtimes, \varepsilon\}$, $x \in Var$, $pnm \in ProcName$, $set \subseteq ALL\text{-}VALUE$, $ch\text{-}set \in ChannelSet$, $P_1, P_1', P_2, P_2' \in ProcExpr$:

$$\boxed{\text{Vacuum}}\ \frac{}{P \xrightarrow{\boxtimes}_C P'} \qquad \boxed{\text{Prefixing}}\ \frac{}{a :: P_1 \xrightarrow{a}_C P_1} \qquad \boxed{\text{Binary Sum}}\ \frac{P_1 \xrightarrow{a}_C P_1'}{P_1 + P_2 \xrightarrow{a}_C P_1'}$$

$$\boxed{\text{ProcDefn}}\ \frac{pnm(\overline{x}) \triangleq P_2, \quad P_2[\overline{v}/\overline{x}] \xrightarrow{a}_C P'}{pnm(\overline{v}) \xrightarrow{a}_C P'} \qquad \boxed{\text{Hiding}}\ \frac{P_1 \xrightarrow{a}_C P_1'}{P_1 \backslash ch\text{-}set \xrightarrow{hide\text{-}act(a, ch\text{-}set)}_C P_1' \backslash ch\text{-}set}$$

$$\boxed{\text{Product}}\ \frac{P_1 \xrightarrow{a_1}_C P_1', \quad P_2 \xrightarrow{a_2}_C P_2'}{P_1 \mid P_2 \xrightarrow{a_1 \star a_2}_C P_1' \mid P_2'} \qquad \boxed{\text{ProcChoice}}\ \frac{\text{for some } v \in se, \quad P_1 [\, v / x \,] \xrightarrow{a}_C P_1'}{\square x \colon se \bullet P_1 \xrightarrow{a}_C P_1'}$$

where $\overline{x}$ is a sequence of formal process parameter and $\overline{v}$ is a corresponding sequence of arguments, of the appropriate type.

Hiding The operation $hide\text{-}act(a, ch\text{-}set)$ simply removes any binding $ch(v)$ appearing in the action a, when ch: ty is a channel declaration in the channel-set $ch\text{-}set \in ChannelSet$.

Action Product ($\star$) combines two input/output actions so that the resulting product action has consistent values if an input channel in one action has the same name as an output channel in the other. When this is the case, the input channel is removed from the final action. The union of the component outputs is taken. If there is an inconsistent value between an input and output channel having the same name, then the zero action $\boxtimes$ results. The action product is defined in terms of action summation and difference operators, $\oplus$ and $\ominus$ respectively:

$$\star : (Act \cup \{\boxtimes, \varepsilon\})^2 \to Act \cup \{\boxtimes, \varepsilon\}$$
$$\forall\ a_1, a_2, b_1, b_2 \in IOAct, \gamma \in Act \bullet$$
$$(a_1\ /\ b_1) \star (a_2\ /\ b_2) = ((a_1 \oplus a_2) \ominus (b_1 \oplus b_2))\ /\ (b_1 \oplus b_2)$$
$$\boxtimes \star \gamma = \boxtimes = \gamma \star \boxtimes$$

Action summation $a \oplus b$ combines two consistent bindings, and returns $\boxtimes$ otherwise and action subtraction $a \ominus b$ removes a binding b if it matches binding a. The following examples illustrate how the *EPA* action product operates. Let w, x, y, z represent distinct concrete channels of type natural numbers:

$$(a) \quad (y(1)\ /\ z(2)) \star (w(1)\ /\ x(3)) = y(1), w(1)\ /\ z(2), x(3)$$
$$(b) \quad (w(2)\ /\ y(1)) \star (y(1)\ /\ x(3)) = w(2)\ /\ y(1), x(3)$$
$$(c) \quad (w(2)\ /\ y(1)) \star (y(4)\ /\ x(3)) = \boxtimes$$

In (a) all the channels are different and so the product action is simply the combination of the primitive actions. In (b), input channel y is now 'connected' to the output y from the first action, and is removed from the resulting inputs. Finally, in (c), the value for y is inconsistent and so $\boxtimes$ results.

4 *EPA* Semantics for ELLA

We now present the *EPA* semantics definitions for a subset of ELLA constructs, in order to outline the process of translating an ELLA expression into its representing *EPA* term. In order to distinguish the syntax of ELLA and *EPA*, then ELLA terms are delimited by semantic brackets '$[\![\ \_\ ]\!]$' and *EPA* terms are delimited by the special brackets '$(\!|\ \_\ |\!)$'. A pre-condition for applying the ELLA-to-*EPA* translation operations, is that the ELLA expressions are well-formed — the process of checking well-formedness also generates typing information about the ELLA expressions which is used by the semantic operations, for example to create channels of the correct type.

Full details of the *EPA* semantics for Core ELLA can be found in [BGMW94b].

4.1 ELLA Closure

An ELLA closure consists of a sequence of well-formed type- and function-declarations $[\![decl_1, \cdots, decl_k]\!]$, which must observe a define-before-use order. The semantic representation of a closure is a dynamic environment, env_k, built via application of the meaning function of declarations to the given sequence:

$$\textit{M-Closure} : \textit{Declaration}^* \to \textit{DynamicEnvironment}$$
$$\textit{M-Closure}[\![decl_1, \cdots, decl_k]\!] = env_k$$
where
$$\forall\ i \in \{1, \ldots, k\} \bullet env_i = \textit{M-Declaration}[\![decl_i]\!](env_{i\text{-}1})$$
$$env_0 = \emptyset$$

4.2 ELLA Type Declarations

ELLA types are simply represented by a name binding for the declared type, translating the ELLA type declaration terms into a broadly similar *EPA* set expression. Below, we give the representation firstly for enumerated types and then for finite sets of tagged integers — here the set has elements $\mathsf{tag}(tag, i)$, for $lwb \leq i \leq upb$:

$$\begin{aligned}
&\textit{M-Declaration}[\![\, \texttt{TYPE}\ tynm =\ \texttt{NEW}\ newty.]\!](\sigma) = \sigma \cup \{tynm \mapsto OBJ\} \\
&\quad \text{where} \\
&\qquad OBJ = (\!\!\{\ tynm \mathrel{\underline{\triangle}} \textit{M-NewType}[\![newty]\!](\sigma)\ \}\!\!) \\
&\textit{M-NewType}[\![(enum_1 \mid\ \cdots\ \mid enum_k)]\!](\sigma) = (\!\!\{\ \{enum_1,\ \cdots\ , enum_k\}\ \}\!\!) \\
&\textit{M-NewType}[\![tag/(lwb..upb)]\!] = (\!\!\{\ \mathsf{tag}(tag, \mathsf{int}\ lwb\ upb)\ \}\!\!) \\
&\quad \text{where} \\
&\qquad \forall\ i \in \{1, \ldots, k\} \bullet enum_i \in Name \\
&\qquad tag \in Name \\
&\qquad lwb, upb \in \mathbb{Z}
\end{aligned}$$

Each type expression is placed into the environment σ, accessed by its name *tynm*, for use later during the translation, and eventually for interpretation of the resulting *EPA* terms.

4.3 Function Declarations

The semantics for an ELLA function declaration simply places a representing *EPA* process definition into the dynamic environment, so that it can be accessed later when a call to the function is made. The process definition uses *EPA* process parameterisation to define the input/output channels used in the function:

$$\begin{aligned}
&\textit{M-Declaration}[\![\, \texttt{FN}\, fnnm = (ty_1\!: sig_1) \rightarrow ty_2 : unm.]\!](\sigma) = \sigma \cup \{fnm \mapsto procdefn\} \\
&\quad \text{where} \\
&\qquad procdefn = (\!\!\{\ \mathsf{process}\, fnnm(sig_1\!:\ ty_1\ /\ oc\!:\ ty_2) \mathrel{\underline{\triangle}} procterm\ \}\!\!) \\
&\qquad procterm = \textit{M-Unit}[\![unm]\!](oc)(\sigma)
\end{aligned}$$

where ty_1, ty_2 are the input and output types of the function[4], and *unm* is the function body *unit* (see below). Note that only a single input to the function is shown here for simplicity — in general multiple inputs can be declared.

4.4 Inertial Delay

In order to introduce predefined ELLA primitives, the body of an ELLA function declaration can contain a primitive construct instead of a unit. The semantics of a primitive is represented in *EPA* by an atomic process definition. Here, we shall only describe a unit delay, when $del = 1$, which is a special case of the more general inertial delay:

[4] *EPA* follows the usual convention of declaring parameters, with the parameter name followed by its type, while ELLA does the reverse, as exhibited by certain programming languages.

$$M\text{-}Declaration[\![\texttt{FN}\,fnnm = (ty) \rightarrow ty\text{: } \texttt{IDELAY}\ (val, del)]\!](\sigma) =$$

$$(\!|\ \mathsf{process}\ fnnm_P(i\colon ty\ /\ o\colon ty)\ \underline{\triangle}\ \square\, v\colon \{val\}\ \bullet\ Udel_{[ty]}(v\ ;\ i\ /\ o)\ |\!)$$

where

$$del = 1$$

where the process *Udel* is the same as that given in §3.1:

$$(\!|\ \mathsf{process}\ Udel_{[ty]}(w\colon ty\ ;\ ic\colon ty\ /\ oc\colon ty)\ \underline{\triangle}\ \square\, v\colon ty\ \bullet\ ic(v)\ /\ oc(w) :: Udel_{[ty]}(v\ ;\ ic\ /\ oc)\ |\!)$$

4.5 Units

The main facility for control and connectivity within ELLA is the *unit* construct. This contains constant values, node or signal naming, function instantiation, control flow using *case*, and composition using *block*. The semantic definition for a unit follows its structure, by giving *EPA* process definitions for atomic expressions and then composing these using the *EPA* product and hiding operations:

$$\begin{array}{rcl}
Unit & ::= & enumerated \mid signal \mid instance \mid case \mid block. \\
Instance & ::= & fnname\ unit. \\
Case & ::= & \texttt{CASE}\ unit_1\ \texttt{OF}\ alt_1,\ \cdots\ , alt_k\ \texttt{ELSE}\ unit_2\ \texttt{END}\ . \\
Alt & ::= & setexpr : unit. \\
Block & ::= & \texttt{BEGIN}\ step_1,\ \cdots\ , step_k\ \texttt{OUTPUT}\ unit\ \texttt{END}\ . \\
Step & ::= & let \mid make \mid join \\
Let & ::= & \texttt{LET}\ signal = unit\ . \\
Make & ::= & \texttt{MAKE}\ fnname : signal\ . \\
Join & ::= & \texttt{JOIN}\ unit \rightarrow signal\ .
\end{array}$$

4.5.1 Enumerated

An ELLA ground value *val* is represented by a call to a basic process definition $Constant_{[ty,val]}$, which has no inputs and always produces *val* on its output channel *oc*:

$$M\text{-}Enumerated : Enumerated \rightarrow Name \rightarrow DynamicEnvironment \rightarrow ProcDefn$$

$$M\text{-}Enumerated[\![val]\!](oc)(\sigma) = (\!|\ Constant_{[ty,val]}(oc)\ |\!)$$

where *ty* is the type containing *val*. The process definition $Constant_{[ty,val]}$ is given by:

$$(\!|\ \mathsf{process}\ Constant_{[ty,val]}(oc\colon ty)\ \underline{\triangle}\ o(val) :: Constant_{[ty,val]}(oc)\ |\!)$$

4.5.2 Signals

The behaviour of a signal *sig* is simply to name a channel, so that connections between ports on different design modules can be made. This is represented by a call to the basic process definition $Chan\text{-}Proc_{[ty]}$ with the input name *sig*, which copies the value on input *sig* to its output *oc*:

$$M\text{-}Unit(sig)(oc)(\sigma) = \llparenthesis\ Chan\text{-}Proc_{[ty]}(sig\ /\ oc)\ \rrparenthesis$$

The process definition $Chan\text{-}Proc_{[ty]}$ simply copies the value of type *ty* on its input to its output:

$$\llparenthesis\ \text{process}\ Chan\text{-}Proc_{[ty]}(i\colon ty\ /\ o\colon ty) \underline{\triangle} \square x\colon ty\ \bullet\ i(x)\ /\ o(x) :: Chan\text{-}Proc_{[ty]}(i\ /\ o)\ \rrparenthesis$$

4.5.3 Instance

The semantics for a function call or instantiation requires the behaviour of the function definition to be invoked upon the function argument *unit*. This is represented in *EPA* by composing a process call to the process definition for the function with a process term representing the arguments. The 'connection' is made via a channel called *pipe* here, which is then hidden:

$$M\text{-}Unit[\![fnname\ unit]\!](oc)(\sigma) = \llparenthesis\ (fnname_P\ \mid\ UN)\backslash\{pipe\colon ty\}\ \rrparenthesis$$

where

$$UN = M\text{-}Unit[\![unit]\!](pipe)(\sigma)$$

where *ty* is the output type of the argument *unit*.

4.5.4 Case

The main control facility in ELLA is the case statement. This has a selector unit *sel-unm*, *k* alternatives alt_i, $i \in \{1,\ldots,k\}$ and an else unit *else-unm*. Each alternative contains a choice set $constset_i$ and a choice unit unm_i. The case is essentially a multiplexer, which selects one of the outputs of the alternatives units unm_i, depending upon whether the value produced by *sel-unm* matches with one of the values contained in the *ith* chooser set $constset_i$. If there is no match then the output of the else unit *else-unm* is produced. The chooser sets must all be disjoint, and so exactly one choice will be made.

The multiplexing function is represented in *EPA* by a process definition *Chooser* which takes an input *sc* to be used as a selector, *k* inputs c_i, $i \in \{1,\ldots,k\}$ to accept the values produced by the choice units, and an input *elc* which takes the value produced by the else unit. Choice within this processed is effected by using the *EPA* case expression to select the appropriate value for the output *oc* of the process.

A call to the *Chooser* process definition is composed with process terms representing the selector, choice and else units, and the connecting channels are then hidden.

$$M\text{-}Unit[\![\ \texttt{CASE}\ sel\text{-}unm\ \texttt{OF}\ citm_1,\ \cdots\ ,citm_{k+1}\ \texttt{ELSE}\ else\text{-}unm\ \texttt{ESAC}\ ]\!](oc)(\sigma) =$$
$$\llparenthesis\ \Big(\ MUX\ \mid\ SEL\ \mid\ P_1\ \mid\ \cdots\ \mid\ P_{k+1}\ \mid\ EL\ \Big)\backslash chandecls\ \rrparenthesis$$

where

$$SEL = M\text{-}Unit[\![sel\text{-}unm]\!](sc)(\sigma)$$
$$\forall\ i \in \{1,\ldots,k+1\} \bullet P_i = M\text{-}Unit[\![unm_i]\!](c_i)(\sigma)$$
$$EL = M\text{-}Unit[\![else\text{-}unm]\!](elc)(\sigma)$$
$$MUX = \llparenthesis\ Chooser_{[sty,oty]}(sc, c_1,\ \cdots\ ,c_{k+1}, elc\ /\ oc)\ \rrparenthesis$$
$$\forall\ i \in \{1,\ldots,k+1\} \bullet set_i = M\text{-}ConstSet[\![constset_i]\!](\sigma)$$

where *chandecls* is the set of channel declarations for channels $sc, c_1, \cdots, c_{k+1}, elc$, and *sty* and *oty* are the output types of the selector unit and choice units, respectively. The operation $\mathit{M\text{-}ConstSet}[\![set]\!](\sigma)$ accepts an ELLA chooser set *set* and produces its *EPA* representation. The process $\mathit{Chooser}_{[sty,oty]}$ is defined as:

$$
\begin{array}{l}
(\!\!\![\ \text{process } \mathit{Chooser}_{[sty,oty]}(sc\colon sty, c_1\colon oty, \cdots, c_{k+1}\colon oty, elc\colon oty\ /\ oc\colon oty) \underline{\triangle} \\
\quad \Box\, s\colon sty, x_1\colon oty, \cdots, x_{k+1}\colon oty, y\colon oty\ \bullet \\
\qquad sc(s), c_1(x_1), \cdots, c_{k+1}(x_{k+1}), elc(y)\ / \\
\qquad\quad oc(\text{case } s : set_1 \rightarrow x_1, \cdots, set_{k+1} \rightarrow x_{k+1} \text{ else } y \text{ end}) \\
\qquad\quad :: \mathit{Chooser}_{[sty,oty]}(sc, c_1, \cdots, c_{k+1}, elc\ /\ oc)\ ]\!\!\!)
\end{array}
$$

A different instantiation of this is required for each different sequence of chooser sets, as well as for the different selector and output types.

4.5.5 Blocks

The block construct is the mechanism in ELLA used for (explicitly) instantiating previously defined functions, and then connecting them together in the required configuration. Local signal names can be declared within steps $step_1, \cdots, step_k$ and bound to unit expressions. The final output unit *unm* is evaluated within this local environment.

$$
\begin{array}{l}
\mathit{M\text{-}Unit}[\![\ \texttt{BEGIN}\ step_1 \ \ldots\ step_k\ \texttt{OUTPUT}\ unm\ \texttt{END}\ ]\!](oc)(\sigma) = \\
\quad (\!\!\![\ \left(\ P_1 \mid \cdots \mid P_m \mid UN\ \right) \backslash chandecls\ ]\!\!\!) \\
\quad \text{where} \\
\qquad lcl\text{-}env = \mathit{M\text{-}Step}^*[\![step_1 \ \ldots\ step_k]\!](\sigma) \\
\qquad m \leq k \\
\qquad \forall\, i \in \{1,\ldots,m\} \bullet P_i = lcl\text{-}env(step\text{-}nm_i) \\
\qquad UN = \mathit{M\text{-}Unit}[\![unm]\!](oc)(lcl\text{-}env)
\end{array}
$$

$$
\begin{array}{l}
\mathit{M\text{-}Step}^* : \mathit{Step}^* \rightarrow \mathit{DynamicEnvironment} \rightarrow \mathit{DynamicEnvironment} \\
\mathit{M\text{-}Step}^*[\![step_1 \ \ldots\ step_k]\!](\sigma) = lcl\text{-}ev_{k+1} \\
\quad \text{where} \\
\qquad lcl\text{-}ev_1 = \sigma \\
\qquad (\forall\, i \in \{1,\ldots,k\} \bullet lcl\text{-}ev_{i+1} = \mathit{M\text{-}Step}(step_i)(lcl\text{-}ev_i)
\end{array}
$$

The local signal names $step\text{-}nm_1, \cdots, step\text{-}nm_m$ are those resulting from `LET` or `MAKE` steps within the block; the set *chandecls* contains declarations for these local channels, which are to be hidden in the final process term.

It has been assumed here for simplicity that signal names used within the block are globally unique — this is not a requirement of the original ELLA expression, which is equipped with a set of scoping rules to determine behaviour when names are re-used.

4.5.6 Step bindings within blocks

A step binds an ELLA unit to a signal name using either a `LET` expression or a `MAKE` – `JOIN` combination. The latter involves explicit function instantiation under `MAKE` together with input binding under `JOIN` — it is required when building feedback networks, under the define-before-use restrictions in ELLA.

For `LET`, the right-hand side *unm* is evaluated and its output named *sig* — the resulting *EPA* term is then stored in the environment:

$$\textit{M-Step}[\![\ \texttt{LET}\ sig = unm.]\!](\textit{lcl-env}) = \textit{lcl-env} \dagger \{sig \mapsto \textit{M-Unit}(unm)(sig)(\textit{lcl-env})\}$$

For `MAKE`, an instantiation of a previously defined function called *fnname* is required, with its output named *sig*. At this stage, the name of the process $fnname_P$ representing the function is simply recorded and stored in the environment.

$$\textit{M-Step}[\![\ \texttt{MAKE}\ fnname:\ sig.]\!](\textit{lcl-env}) = \textit{lcl-env} \cup \{sig \mapsto (\!\!\lbrack\ fnname_P\ \rbrack\!\!)\}$$

Now, in the corresponding `JOIN` statement, a process call *OBJ* is made to the process associated with *fnname*, with its output called *sig*. The inputs to this process are obtained by composing this term with the evaluated unit *UN*, representing the unit *unm*. A channel, called *pipe* here, is used to effect the 'connection' and is then hidden:

$$\textit{M-Step}[\![\ \texttt{JOIN}\ unm\ \rightarrow\ sig.]\!](\textit{lcl-env}) = \textit{lcl-env} \dagger \{sig \mapsto COMP\}$$

where

$$COMP = (\!\!\lbrack\ \Big(\ UN\ |\ OBJ(pipe\ /\ sig)\ \Big) \backslash \{pipe\colon\ uty\}\ \rbrack\!\!)$$
$$UN = \textit{M-Unit}[\![unm]\!](pipe)(\textit{lcl-env})$$
$$\textit{lcl-env}(sig) = OBJ$$

where *uty* is the output type of *UN*. Note that if any local signals have been used within the various units *unm* of the steps, then only channel *naming* has been carried out at this stage — the actual composition takes place within product operation performed for the complete block structure.

4.6 Example of ELLA-to-*EPA* Translation

In this section, we give an ELLA-text description for a simple water pump controller, and then show the result of applying the semantic operations given above.

Consider the water pump controller, derived from [GB91]:

> *A reservoir is connected to a lake by a pipe line. Water is taken from the lake to the reservoir by a system of two pumps. Two level sensors are installed on the reservoir. Their outputs are denoted by signals a_1, a_2. Signal a_i is 0 when the water is above level i, for $i = 1,2$ and has a value 1 when the water is below level i. The number of pumps that are on at any one time depends on the water level in the reservoir, and is equal to the sum of the signals, $a_1 + a_2$. For example, if the water level is between level 1 and 2, as show in Figure 2, then one pump should be in operation. If the water is above level 1 then no pumps should be in operation. In order to equalise wear on the pumps, they should come into operation in an alternating manner.*

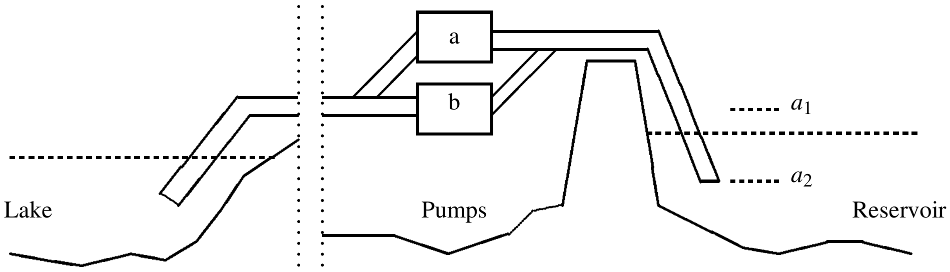


Figure 2: Lake, Pumps, and Reservoir

In the following Core ELLA description two types are used: the first, an enumeration type called `pump`, denotes which pumps are actually operating, the pumps being known as `a` and `b` — the value `ab` means that both pumps are on. The second, a tagged integer type `level`, denotes the level of water in the reservoir, with `lv/0` representing a full reservoir, and `lv/2` an empty one. The function `CONTROL` is the pump controller[5] and its `CASE` clause determines which pumps are switched on, depending on the previous state of the pumps held in `store` and the current water level `in`:

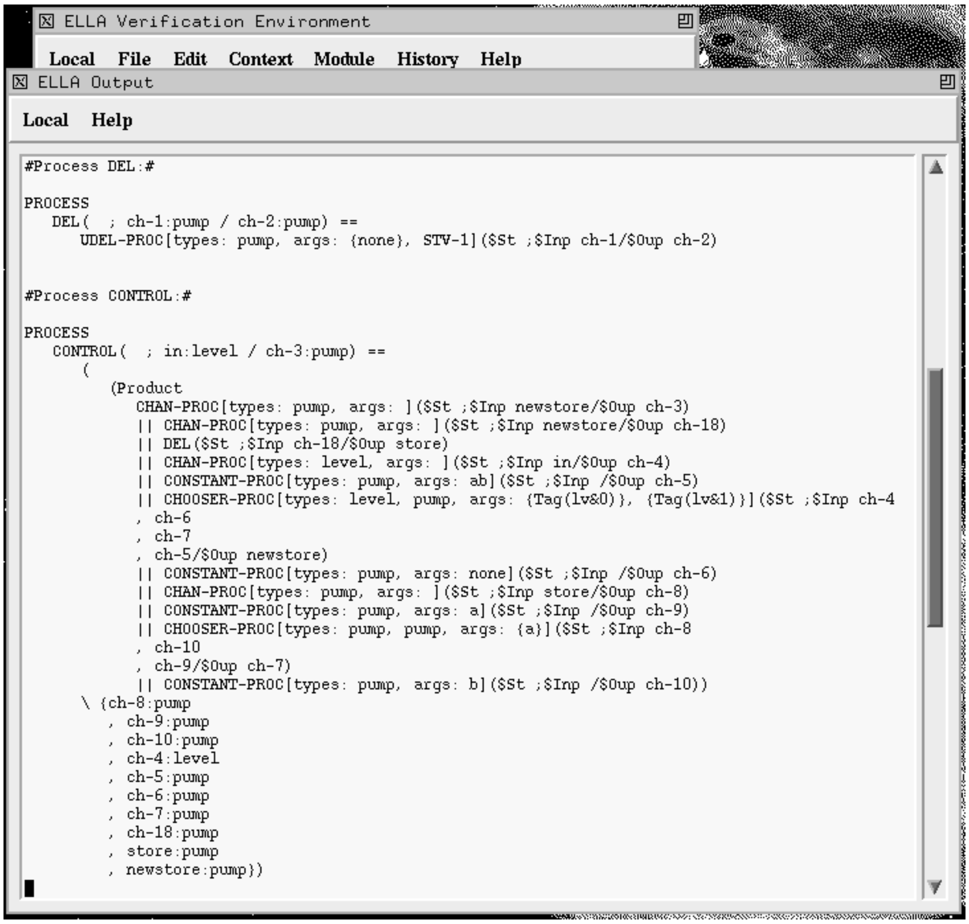
```
TYPE  pump  = NEW (none | a | b | ab).
TYPE  level = NEW lv/(0..2).
FN DEL = (pump) -> (pump): IDELAY(none, 1).

FN CONTROL = ( level: in ) -> pump:
BEGIN
   MAKE DEL: store.
   LET newstore = CASE in OF
                    lv/0: none,
                    lv/1: CASE store OF
                            a : b
                          ELSE a
                          ESAC
                  ELSE ab
                  ESAC.
   JOIN newstore -> store.
   OUTPUT newstore
END.
```

Application of the semantic operations results in the collection of *EPA* definitions given in Figure 3. This shows a display tool in the implemented ELLA Verification Environment (see §5 below), containing the *EPA* terms generated for the pump controller by the ELLA-to-*EPA* compiler. Notice in particular how the right-hand side of the process definition `CONTROL2( ; in:level / ch-19:pump)` contains a product of calls to atomic processes within a single hiding of the internal channels. For example, processes `UDEL`, `CONSTANT-PROC`, `CHAN-PROC` were defined in §4.4, §4.5.1 and §4.5.2, respectively. This regular structure means that the `CONTROL2` process definition can be translated into an equivalent *normal form*, described below in §5.1.

Figure 4 shows a typical simulation for the pump controller, generated using the Soft Logic Analyser (SLA) from the ELLA Verification Environment (again, see §5 for more details).

[5] Under certain input sequences — for example, if `in` alternates between `lv/0` and `lv/1` — then the condition that the pumps are used alternately is not met. This problem could of course be corrected, but additional circuitry would be required, and this would not add further to this illustration of the *EPA* semantics.




```
#Process DEL:#

PROCESS
   DEL(  ; ch-1:pump / ch-2:pump) ==
      UDEL-PROC[types: pump, args: {none}, STV-1]($St ;$Inp ch-1/$Oup ch-2)


#Process CONTROL:#

PROCESS
   CONTROL(  ; in:level / ch-3:pump) ==
      (
         (Product
            CHAN-PROC[types: pump, args: ]($St ;$Inp newstore/$Oup ch-3)
            || CHAN-PROC[types: pump, args: ]($St ;$Inp newstore/$Oup ch-18)
            || DEL($St ;$Inp ch-18/$Oup store)
            || CHAN-PROC[types: level, args: ]($St ;$Inp in/$Oup ch-4)
            || CONSTANT-PROC[types: pump, args: ab]($St ;$Inp /$Oup ch-5)
            || CHOOSER-PROC[types: level, pump, args: {Tag(lv&0)}, {Tag(lv&1)}]($St ;$Inp ch-4
            , ch-6
            , ch-7
            , ch-5/$Oup newstore)
            || CONSTANT-PROC[types: pump, args: none]($St ;$Inp /$Oup ch-6)
            || CHAN-PROC[types: pump, args: ]($St ;$Inp store/$Oup ch-8)
            || CONSTANT-PROC[types: pump, args: a]($St ;$Inp /$Oup ch-9)
            || CHOOSER-PROC[types: pump, pump, args: {a}]($St ;$Inp ch-8
            , ch-10
            , ch-9/$Oup ch-7)
            || CONSTANT-PROC[types: pump, args: b]($St ;$Inp /$Oup ch-10))
      \ {ch-8:pump
         , ch-9:pump
         , ch-10:pump
         , ch-4:level
         , ch-5:pump
         , ch-6:pump
         , ch-7:pump
         , ch-18:pump
         , store:pump
         , newstore:pump})
```

Figure 3: *EPA* Process Terms for Pump Controller

5 Applications: Symbolic Simulation and Formal Verification for *EPA*

We now outline further processing that can be applied to the 'raw' ELLA-to-*EPA* translation produced by a compiler based on the semantic definitions given above. A *normal form* provides the basis for formally-based analysis tools for (symbolic) simulation and verification. This is a state machine-like description in *EPA*, with a response function r_P and an evolution function e_P. These are both presented in a truly symbolic and structured manner, using the ELLA action algebra. They can therefore be transformed using rewrite rules based on the *EPA* transition rules — this provides the core functionality for symbolic simulation. The normal form is also used as the basis for formal verification using a *state evolution* rule. This effectively generates first-order logical formula from the response and evolution function expressions, whose validity ensures that the systems are strongly bisimilar. The methodology is again exploiting the truly symbolic nature of the process terms.

The tools described have been implemented as prototypes, and are embedded within an ELLA Verification Environment, in collaboration with Harlequin Ltd, Cambridge, and DRA, Malvern. Some aspects of the implementation work are highlighted below.

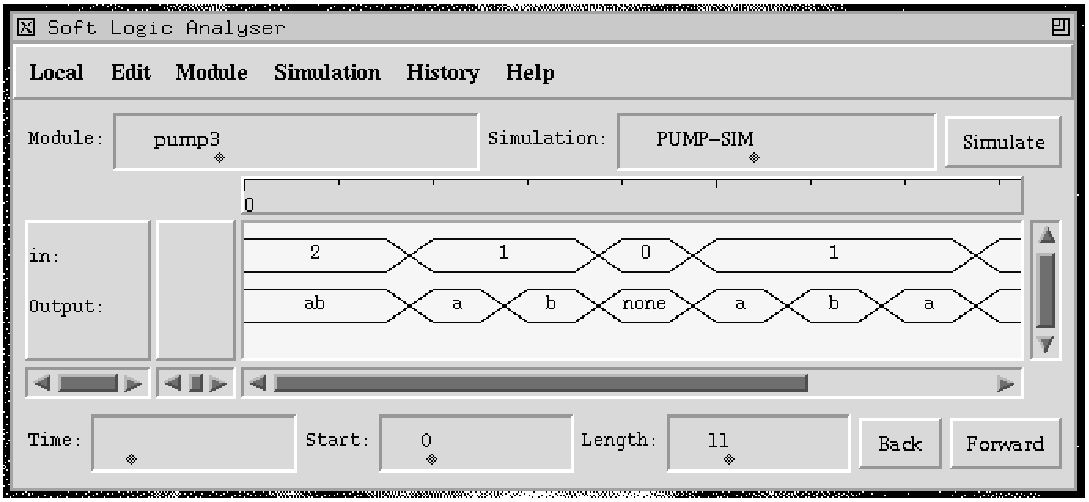


Figure 4: *SLA* Simulation for Pump Controller

5.1 *EPA* Normal Forms for ELLA

When *EPA* is used to represent the semantics of hardware systems described in ELLA, the resulting process definitions can be reduced to a pair of process definitions that constitute an *EPA normal form*. This arises from the regular and static structure of the ELLA hardware model, where the number, and names, of channels for each component remain constant. The normal form is the basis for (symbolic) simulation tools and for verification using state bisimulation as described below in §5.3.

The semantics of any ELLA expression can be represented by a process P, that determines the initial 'state' conditions, and then calls a general process Q which gives the response and evolution behaviour:

$$
\begin{array}{rcl}
P(i\colon se_1 \;/\; o\colon se_2) & \underline{\triangle} & \square y\colon se_3 \;\cdot\; Q(y \;;\; i \;/\; o) \\
Q(v\colon se_4 \;;\; i\colon se_1 \;/\; o\colon se_2) & \underline{\triangle} & \square x\colon se_1, w\colon se_5 \;\cdot \\
 & & \quad i(x) \;/\; o(r_Q(x,v,w)) :: Q(e_Q(x,v,w) \;;\; i \;/\; o)
\end{array}
$$

where r_Q is a *response* function and e_Q is an *evolution* function. The choice $y\colon se_3$, where y ranges over the initial state-space se_3, determines the initial state of the construct. In process Q, the value variable v ranges over the (full) state-space se_4, with $se_3 \subseteq se_4$. The choice $x\colon se_1$ binds the variables occurring in the input actions, and $w\colon se_5$ represents variables used in the response and evolution functions that are not input-assigned, giving the scope for non-deterministic behaviour.

5.1.1 Rules for Normalisation of *EPA* Terms for ELLA

We briefly illustrate how an *EPA* term representing an ELLA expression, and generated according to the semantics for ELLA, can be translated into a behaviourally equivalent call to a normal-form process. The *EPA* terms representing an ELLA expression are built by composing calls to *atomic* process definitions, which represent atomic ELLA constructs, using process call, product and hiding operations. An *EPA* process term resulting from ELLA can then be normalised using a small set of rewrite rules. Each rule can be read as follows: if a process term matches the top of the rule, then it can be replaced by the process term given as the bottom of the rule, which will be equivalent. Below, instances of the three main rules are given. In these it is assumed that names used for processes, channels, and values are suitably unique. The names for the channels and values are also used both as formal and actual parameters. We indicate how the top and bottom processes can be shown to be equivalent.

Process Call Rule: When the right-hand side of a process has been normalised, it will simply be a call to the new process definition constructed by normalisation; any call to the original process definition can now be replaced by a call to the new process. For example:

$$\boxed{\text{Call}}\ \frac{\Gamma \vdash P(v\ ;\ i\ /\ o)}{\Gamma \vdash Q(v\ ;\ i\ /\ o)}$$

where

$$\text{process } P(v\ ;\ i\ /\ o) \mathrel{\underline{\triangle}} Q(v\ ;\ i\ /\ o) \ \in\ \Gamma$$
$$\text{process } Q(v\ ;\ i\ /\ o) \mathrel{\underline{\triangle}} rhs \ \in\ \Gamma$$

where *rhs* is an arbitrary process term representing the definition of Q, and i, o are (sequences of) channels. The rule states that if a set of *EPA* definitions, Γ, contains process definitions for P, Q, where P simply calls Q, then the call to P can be replaced by a call to Q. The equivalence of the process terms follows simply from the transition rule for process definition.

Hiding Rule: The following rule constructs normal form process terms, from which any output channels contained in the set of channels *chanset* have been removed. Here, i, o represent sequences of channels:

$$\boxed{\text{Hide}}\ \frac{\Gamma \vdash P(v\ ;\ i\ /\ o)\backslash chanset}{\Gamma \cup \{defn\} \vdash Q(v\ ;\ i\ /\ o')}$$

where

$$\text{process } P(v\ ;\ i\ /\ o) \mathrel{\underline{\triangle}} \square x{:}I \ \bullet\ prefix :: P(next\ ;\ i\ /\ o) \ \in\ \Gamma$$
$$defn \ =\ \text{process } Q(v\ ;\ i\ /\ o') \mathrel{\underline{\triangle}}$$
$$\square x{:}I \ \bullet\ hide\text{-}act(prefix, chanset) :: Q(next\ ;\ i\ /\ o')$$
$$o' = o - chanset$$

where *prefix* and *next* are the prefix-action and next-state expressions for process P, respectively; Q is assumed to be a fresh process name in Γ. The operation $hide\text{-}act(prefix, chanset)$ removes any output bindings in the prefix action *prefix* which appear in *chanset*, as outlined in 3.2.1; the channel sequence o' is the result of removing channels in *chanset* from the channel sequence o. Again the correctness of the rule is clear from the *EPA* transition rules, for process definition, hiding and choice.

Product Rule: The most complex rule applies to process products, replacing the product term with a call to a new process definition, which is already in normal form. This has the following general structure:

$$\boxed{\text{Prod}}\ \frac{\Gamma \vdash P(v\,;\,i_1\,/\,o_1)\ \mid\ Q(w\,;\,i_2\,/\,o_2)}{\Gamma \cup \{R_{defn}\} \vdash R(v,w\,;\,i_3\,/\,o_1,o_2)}$$

where informally i_1, i_2, i_3 are appropriate (sequences of) input channels, and o_1, o_2 are disjoint (sequences of) output channels. We assume that the environment Γ contains definitions for two processes P, Q which are in normal form. and then a fresh process definition R_{defn} is introduced. The conclusion to the rule is then a call to this new process.

In order to demonstrate how the rule is established, we first examine the processes in terms of the transitions that they may be able to perform, before giving an instance of the rule. So, let the processes P, Q and R take value parameters from some set *VALUE*, and be so defined that they can make moves of the following form:

$$\forall\ v, w \in \mathit{VALUE} \bullet$$

$$P(v) \xrightarrow{\alpha_v} P(v'), \qquad Q(w) \xrightarrow{\beta_w} Q(w'), \qquad R(v,w) \xrightarrow{\alpha_v \star \beta_w} R(v',w')$$

for some $v', w' \in \mathit{VALUE}$, where $\alpha_v, \beta_w \in \mathit{Act}$ are all of the possible (concrete) actions, which may depend on the values v, w. Process R has therefore been defined so that when processes P and Q can make moves labelled α_v and β_w, it can make a move labelled with the action product $\alpha_v \star \beta_w$. Note that, for simplicity, we have assumed that all the channels have *concrete* names within the actions, and have therefore ignored any channel parameters in the process calls. We now claim that:

$$S = \{\langle (P(v)\ \mid\ Q(w)), R(v,w)\rangle \mid v, w \in \mathit{VALUE}\}$$

is a bisimulation relation[6] between the process terms $(P(v)\ \mid\ Q(w))$ and $R(v,w)$, and that $(P(v_0)\ \mid\ Q(w_0))$ and $R(v_0,w_0)$ are therefore bisimulation equivalent, for some particular initial state values v_0, w_0.

This is indeed the case, since for every transition the product element can make, then the R process can also make a transition with an identical action label, and vice versa. In each instance, the pair of continuation processes is also in the relation S:

$$(P(v)\ \mid\ Q(w)) \xrightarrow{\alpha_v \star \beta_w} (P(v')\ \mid\ Q(w')) \qquad \Leftrightarrow \qquad R(v,w) \xrightarrow{\alpha_v \star \beta_w} R(v',w')$$

So if the normal form process definitions for P, Q and R only result in transitions of the above form, then $(P(v_0)\ \mid\ Q(w_0))$ is equivalent to $R(v_0,w_0)$. As before, we can establish this from the *EPA* transition rules.

We now give an example of how the 'product' process definition R is calculated for the most interesting instance of the rule, where the output of one process is connected to the input of the second, to form a pipeline of length two:

[6] See §5.3 for a definition of bisimulation.

$$\boxed{\text{Prod}}\ \frac{\Gamma \vdash P(v_1\ ;\ i\ /\ c)\ \mid\ Q(v_2\ ;\ c\ /\ o)}{\Gamma \cup \{\mathit{defn}\} \vdash R(v_1, v_2\ ;\ i\ /\ c, o)}$$

where

process $P(v_1\ ;\ i\ /\ c) \triangleq$
$$\Box x_1{:}I_1\ \bullet\ i(x_1)\ /\ c(r_1(x_1,v_1)) :: P(e_1(x_1,v_1)\ ;\ i\ /\ c)\ \in\ \Gamma$$

process $Q(v_2\ ;\ c\ /\ o) \triangleq$
$$\Box x_2{:}I_2\ \bullet\ c(x_2)\ /\ o(r_2(x_2,v_2)) :: Q(e_2(x_2,v_2)\ ;\ c\ /\ o)\ \in\ \Gamma$$

$\mathit{defn}\ =$ process $R(v_1,v_2\ ;\ i\ /\ c,o) \triangleq$
$$\Box x_1{:}I_1\ \bullet\ i(x_1)\ /\ c(r_1(x_1,v_1)), o(r_2(r_1(x_1,v_1),v_2))$$
$$:: R(e_1(x_1,v_1), e_2(r_1(x_1,v_1),v_2)\ ;\ i\ /\ c,o)$$

where r_1, r_2 are the response functions and e_1, e_2 are the evolution functions of the two processes. The context Γ contains definitions for P, Q and so the product term $(P(i\ /\ c)\ \mid\ Q(c\ /\ o))$ can be replaced by a call to $R(i\ /\ c,o)$, where a definition for R is added to the context. R has a single input i, from process Q (the input channel c is removed from actions in R, as defined by the process product operation), but has the outputs from both P and Q. The state variables and evolution functions are also simply a combination of those found in P and Q. The action product is formed between the elemental actions for P and Q by *unifying* the value expressions for the same-name channels, and then applying the resulting substitution to the process components. This means that the variable x_2 in the action and evolution expressions contributed by Q, is substituted by the value expression $r_1(x_1,v_1)$ to form the new action and evolution expressions for R. Note that R is assumed to be a fresh process name in Γ.

5.1.2 Example of Normalisation: Pump Controller

The pump controller process *CONTROL* can be normalised, using the above rules. Figure 5 shows the process term generated by the implemented ELLA-to-*EPA* compiler, as displayed in the ELLA Verification Environment. Note that it is presented in a special normal-form style, which contains the same information provided by the initial and general process terms described above, but organised into a more compact format. The process body consists of variable declarations, input bindings, output bindings, and then the next-state expressions — these contain initial values and the evolution function expression.

5.2 Compilation, Simulation and Verification

An ELLA-to-*EPA* compiler has been developed by DRA, Malvern and Harlequin Ltd. Its structure closely follows the semantic definitions outlined in this paper, which has the advantage of making the task of implementation both easier and less error-prone. The compiler converts the resulting process terms into normal form. The response and evolution function expressions of the resulting process definition are then interpreted, producing Lisp code which can then be compiled and executed, forming the core of an efficient ELLA simulator, based on the semantic definitions.

The compiler is embedded into an ELLA design and verification environment, which also contains a schematics editor, and a graphics user-interface to the simulator, called the *soft logic analyser* (SLA),

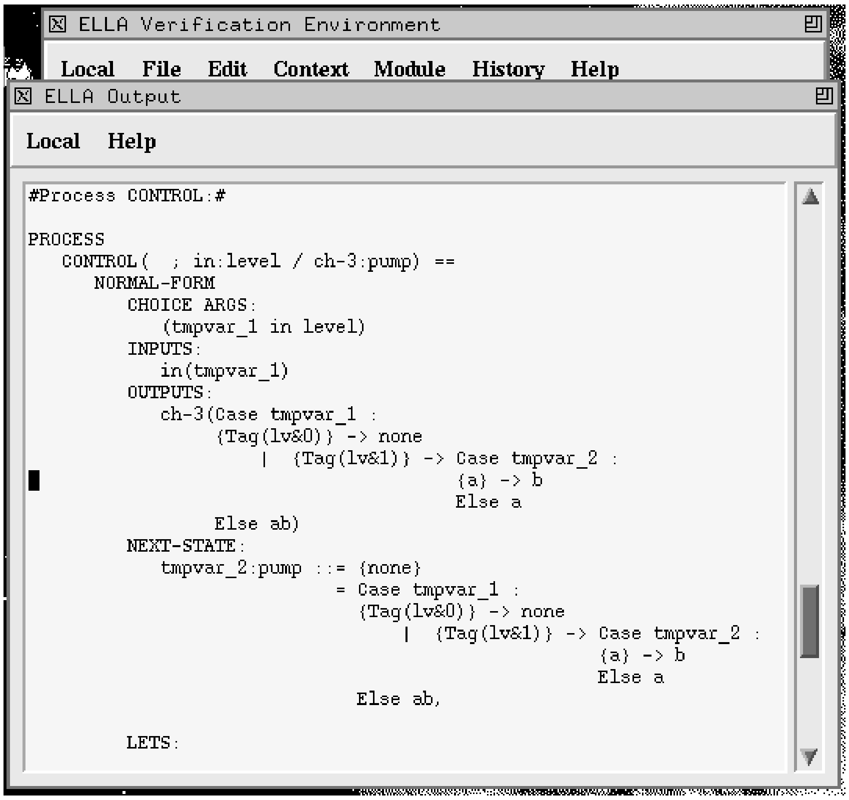


Figure 5: Normal Form *EPA* Process Term Pump Controller

which was illustrated in Figure 4. This provides a waveform-like display for (ground) simulation results, together with signal probing and forcing facilities.

A second back-end to the compiler has been developed by Manchester. This is again based on the formal semantics, but now translates the raw compiler output into the *EPA* normal form. This provides the basis for *symbolic* analysis, such as simulation and verification, which are outlined below.

5.2.1 Symbolic Simulation

A symbolic simulation tool (*SymSim*) has been developed and embedded into the ELLA Verification Environment, so that a designer can analyse the effects of applying *non-ground* expressions to a design. As with the ground simulation tool, design nodes can be monitored, but here the nodes are bound to general *EPA expressions* instead of to literal values. The tool thus allows the evolving structure of a design to be investigated, for example by analysing the progress of signals through the design.

One potential problem with symbolic simulation concerns the size of the resulting expressions which need to be displayed and analysed, especially as the simulation progresses. In *EPA*, much of the original structure of a design can be retained in the *EPA* process terms produced by the compiler. so that *SymSim* only needs initially to display a single layer of each expression. Then, the designer can 'expand' those parts of the expression required for further analysis, by substituting nodes by the expressions to which they are bound at each time-step, or by replacing function calls by the corresponding instantiated function body. Note that each node has a time component appended to it base-name, and so represents a value in a design at a particular point in both space and time. Figure 6 shows *SymSim* applied to the pump controller.

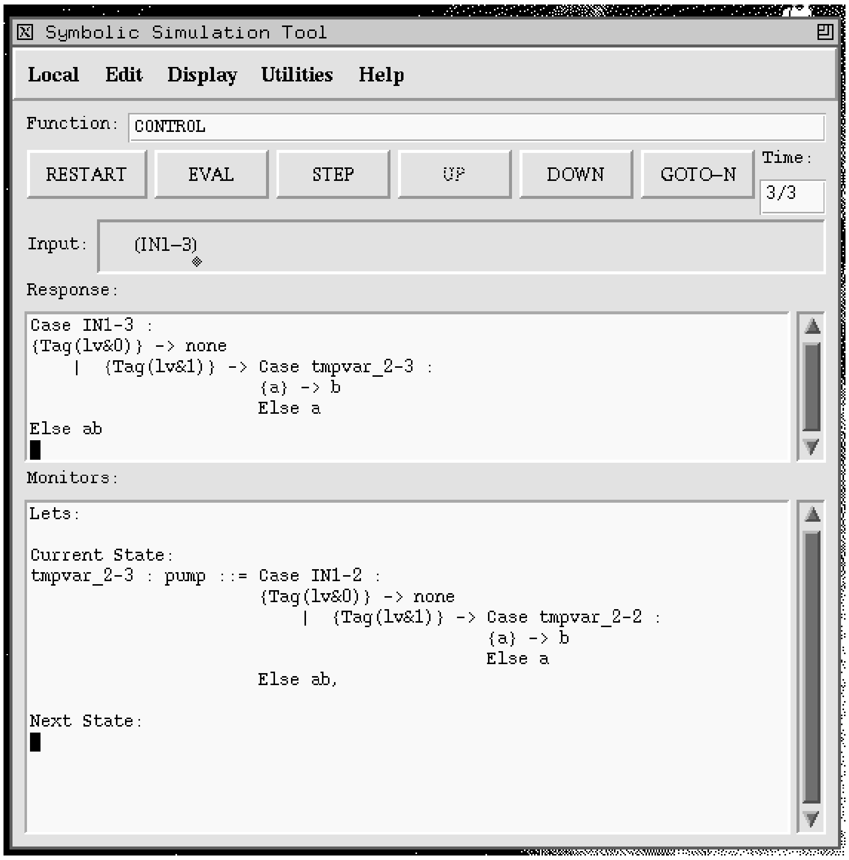


Figure 6: *SymSim* Symbolic Simulation of Pump Controller

It shows a non-ground value as input, and certain nodes within the design are under observation. Only expression layers for the current time-step are displayed — expressions bound to nodes at previous time-steps have not yet been included.

5.2.2 Symbolic Verification: The Verification Condition Generator (VCG)

Having compiled ELLA expressions into a set of *EPA* terms, and then transformed the set into *EPA* normal form, we can consider establishing equivalences between two *EPA* systems. In particular, the notion of (strong) bisimilarity[Mil89] will be used. In the next section we outline a *state bisimulation* relation, equivalent to strong bisimilarity, which operates on the abstract *EPA* terms produces from ELLA. An accompanying *state evolution* method generates verification conditions needed to establish state bisimilarity between two *EPA* systems. The approach offers the possibility of containing the usual growth in complexity of verification, from which state-based verification methods suffer. A high degree of automation is still possible, however. Although we will describe the method for deterministic systems, it is possible to extend it to non-deterministic systems. Note that this method is main subject of [BGMW94a], where it is described in detail.

5.3 State Bisimulation and State Evolution

Each design is represented by an *EPA* process P in normal form. From this are extracted a response function $r_P(x,\sigma_P)$, and an evolution function $e_P(x,\sigma_P)$. At each step, these take an input value x from some set of inputs I and 'current-state' value σ_P from a state-space St_P. P will also have an initial state value i_P which is assigned to σ_P at the initial time-step 0.

There are a number of stages involved in establishing the bisimilarity of two process P and Q when represented by response and evolution functions. First, we give the standard definition of bisimulation equivalence between two processes P and Q:

Definition 5.1 : (Strong) Bisimilarity

The relation S is a (strong) bisimulation if and only if:

- S is symmetric.
- $\langle P,Q\rangle \in S \;\Rightarrow$
  If $P \xrightarrow{\alpha} P'$ then there is some Q, such that $Q \xrightarrow{\alpha} Q'$ and $\langle P',Q'\rangle \in S$

We say that two processes P,Q are *bisimilar*, denoted as $P \sim Q$ whenever there exists a strong bisimulation S such that $\langle P,Q\rangle \in S$.

Now we introduce the notion *state bisimulation* which relates the *state spaces* of P and Q (as opposed to being a relation between process terms):

Definition 5.2 : State bisimulation
Let $SB \subseteq (St_P \times St_Q)$ be a binary relation between the state spaces of P and Q. SB is a state bisimulation for $P,\ Q$ when:

1. $\langle i_P,\ i_Q\rangle \in SB$
2. If $\langle \sigma_P,\ \sigma_Q\rangle \in SB$ then $\forall\, x \in I$,
   (a) $r_P(x,\sigma_P) = r_Q(x,\sigma_Q)$, and
   (b) $\langle e_P(x,\sigma_P),\ e_Q(x,\sigma_Q)\rangle \in SB$

This is clearly related to a similar notion defined by Olderog appearing in [Old91]. The difference between the approaches is that we do not wish to immediately identify the concept of "state" with that of "process" as directly as Olderog does.

We now state the desired association between state and process:

Theorem 5.3 : Correctness of state bisimulation
Let $P,\ Q$ be deterministic systems in normal form.

$$(P \sim Q) \;\Leftrightarrow\; \exists\, SB \subseteq (St_P \times St_Q) \cdot SB \text{ is a state bisimulation for } P \text{ and } Q$$

5.3.1 State Evolution

The problem of establishing the strong bisimulation equivalence of two processes P and Q has been transformed into the problem of establishing a state bisimulation between the response and evolution functions which represent P and Q. This can be performed in a truly symbolic manner by using the *state evolution* method. This generates a set of verification conditions whose validity would ensure that the systems were (state) bisimilar.

The idea is to take an *arbitrary*, but general point $\langle \sigma_P,\ \sigma_Q \rangle$, in the product state space, $(St_P \times St_Q)$. As we know nothing about this point to start with, we can simply name it (or rather its components) using appropriate symbolic variables.

We then proceed to symbolically evolve both of the systems on the working assumption that, for all inputs, the responses produced by each system should be in complete agreement at each stage. Now *either* we show that, for some input, the responses clearly disagree at some stage (i.e. P and Q are different) *or* we demonstrate that the symbolic simulation is 'sufficiently complete', implying that the systems P and Q are behaviourally equivalent.

So the main issues become:

- what needs to be proven at each stage?
- what are the criteria for terminating?

Let us imagine that we are at stage N in applying the method. We have not already terminated — and so we consider the comparison of responses for the current step, based upon previous steps:

$$R_N(\sigma_P, \sigma_Q) \quad \Leftrightarrow \quad \forall\, x_N,\ \cdots\ x_1 \in I \bullet r_P(x_N,\ sp_N) = r_Q(x_N,\ sq_N)$$

where

$$\forall\, i \in \{1, \ldots, N\} \bullet sp_i = e_P(x_i, sp_{i\text{-}1})$$
$$\forall\, i \in \{1, \ldots, N\} \bullet sq_i = e_Q(x_i, sq_{i\text{-}1})$$
$$sp_0 = \sigma_P$$
$$sq_0 = \sigma_Q$$

where sp_N and sq_N are expressions describing the 'state' at stage N — they are dependent upon inputs $x_{N\text{-}1},\ \cdots\ x_1$ and initial product state $\langle \sigma_P,\ \sigma_Q \rangle$.

We should now check two things:

1. check that we continue to achieve agreement assuming that we had begun from the initial product state $\langle i_P,\ i_Q \rangle$. This amounts to instantiating the above formula with the initial state and showing that it is valid.
2. check to see if we can show that, on the basis of agreement on *earlier* steps, agreement of response is actually *forced* for the present step.

If the first check fails, then we have shown that the systems are manifestly different (after iterating from the initial state) and so we should then abort.

If both the first and second check are positive, then we stop with success since, intuitively, we have shown that the next step is implied by the immediate past history. Hence, the systems will continue to behave identically, once a product state is found where they will agree for N-1 steps. But the success of the first check shows that this has already been achieved from the initial state.

If the second check fails, then we proceed to the next stage of the iteration, since we have not been able to show one way or another whether the two systems are equivalent or not. Of course, we must also check that evolution from the initial states also establishes agreement for as many steps.

Definition 5.4 : State Evolution rule

Using the notation introduced above we can now describe the State Evolution inference rule:

$$\boxed{\text{State Evolve}}\ \frac{\begin{array}{l}\text{For some fixed } N \in \mathbb{N}: \\ \quad \text{(a)} \quad \Gamma \vdash R_1(i_P, i_Q) \wedge \cdots \wedge R_N(i_P, i_Q) \\ \quad \text{(b)} \quad \Gamma \vdash \forall\, \sigma_P,\ \sigma_Q\ \bullet R_1(\sigma_P, \sigma_Q) \wedge \cdots \wedge R_N(\sigma_P, \sigma_Q) \ \Rightarrow\ R_{N+1}(\sigma_P, \sigma_Q)\end{array}}{\Gamma \vdash P \sim Q}$$

The above rule says that if we can establish, for some *fixed* N:

(a) equivalent responses for N steps from the initial states, $\langle i_P,\ i_Q \rangle$, and:

(b) equivalent response for the $N+1$'st step from states $\langle \sigma_P,\ \sigma_Q \rangle$ follows from equivalent responses for all preceding steps from the states $\langle \sigma_P,\ \sigma_Q \rangle$

then we can conclude that the two deterministic systems P and Q are strongly bisimilar. The soundness of the state evolution rule with respect to strong bisimilarity, and a discussion of termination conditions can be found in [BGMW94a]

5.3.1.1 Example: Integer Delay Consider the following problem of determining whether two normal form *EPA* process terms, $P(i/o)$ and $Q(i/o)$ are equivalent:

$$\begin{array}{rcl} P(\mathit{in}\colon \mathbb{N} \ /\ \mathit{out}\colon \mathbb{N}) & \underline{\triangle} & P'(0\ ;\ \mathit{in}\ /\ \mathit{out}) \\ P'(v\colon \mathbb{N}\ ;\ \mathit{in}\colon \mathbb{N} \ /\ \mathit{out}\colon \mathbb{N}) & \underline{\triangle} & \square\, w\colon \mathbb{N}\ \bullet\ \mathit{in}(w)\ /\ \mathit{out}(v) :: P'(w\ ;\ \mathit{in}\ /\ \mathit{out}) \\ \\ Q(\mathit{in}\colon \mathbb{N} \ /\ \mathit{out}\colon \mathbb{N}) & \underline{\triangle} & Q'(1\ ;\ \mathit{in}\ /\ \mathit{out}) \\ Q'(v\colon \mathbb{N}\ ;\ \mathit{in}\colon \mathbb{N} \ /\ \mathit{out}\colon \mathbb{N}) & \underline{\triangle} & \square\, w\colon \mathbb{N}\ \bullet\ \mathit{in}(w)\ /\ \mathit{out}(v \dot{-} 1) :: Q'(w+1\ ;\ \mathit{in}\ /\ \mathit{out}) \end{array}$$

Process term $P(\mathit{in}\ /\ \mathit{out})$ is a unit delay over the natural numbers, with initial output 0. We wish to establish that $Q(\mathit{in}\ /\ \mathit{out})$ is equivalent, even though the response and evolution functions are different, involving subtraction[7] and addition of 1 — clearly, the combined effect of these cancel each other out.

To show that the two processes are equivalent, we apply the state evolution rule and find some N such that is holds — in this case only one step will be required. P and Q can be represented by their response and evolution functions:

$$\begin{array}{lll} r_P(x, sp) = sp, & e_P(x, sp) = x, & i_P = 0 \\ r_Q(x, sq) = sq \dot{-} 1, & e_Q(x, sq) = x+1, & i_Q = 1 \end{array}$$

[7]The subtraction '$x \dot{-} y$' is defined in the usual way, so that it behaves as standard integer subtraction when $x \geq y$, and returns zero when $x < y$.

For $N = 1$ the premisses of the state evolution rule therefore become:

$$
\begin{array}{ll}
\text{(a)} & R_1(i_P, i_Q) \\
& \quad \Leftrightarrow\ (\forall\, x \in \mathbb{N} \bullet r_P(x,0) = r_Q(x,1)) \\
& \quad \Leftrightarrow\ (0 = 1 \dot{-} 1) \\
\text{(b)} & (R_1(\sigma_P, \sigma_Q)\ \Rightarrow\ R_2(\sigma_P, \sigma_Q)) \\
& \quad \Leftrightarrow\ \forall\, x_1, x_0, \sigma_P, \sigma_Q \in \mathbb{N} \bullet \\
& \qquad (r_P(x_0, \sigma_P) = r_Q(x_0, \sigma_Q))\ \Rightarrow\ (r_P(x_1, e_P(x_0, \sigma_P)) = r_Q(x_1, e_Q(x_0, \sigma_Q))) \\
& \quad \Leftrightarrow\ \forall\, x_1, x_0, \sigma_P, \sigma_Q \in \mathbb{N} \bullet \\
& \qquad \sigma_P = \sigma_Q \dot{-} 1\ \Rightarrow\ (x_0 = (x_0 + 1) \dot{-} 1)
\end{array}
$$

In this case, expression (a) and the conclusion for (b) are tautologies, and so both premisses are valid. Therefore, by the state evolution rule, the processes P and Q are (strongly) bisimilar. Note that even in this simple case, the state-spaces involved are infinite.

6 Summary

We have shown the development of a formal semantics for the commercial hardware description language ELLA, by defining semantics for Core ELLA, a covering subset of full ELLA — any full ELLA construct can be translated into an equivalent Core ELLA construct. The development of the semantics begins with an enumerated compositional state-based approach using IOAs, which have the great advantage of having automatic verification methods; however they have the problem of exponential growth in representation and verification complexity. They also represent the semantics at a low-level — in terms of input/output behaviour — which does not provide a good mechanism for useful feedback to the designer when an analysis fails.

A second semantics — equivalent to, and derived from the IOA semantics – is then given in terms of a process algebra *EPA*. This supports symbolic value expressions, and therefore compact representation, and high-level structured reasoning. A *normal form* for *EPA* terms resulting from compilation of ELLA expressions provides the basis for simulation and verification tools. A *state bisimulation* rule is introduced, with an accompanying *state evolution* method, which provides a tractable approach to (symbolic) verification of *EPA* systems.

A prototype ELLA verification environment has been implemented and is used to illustrate application of the design and verification tools described. The environment provides the hardware designer with a sophisticated and uniform graphical interface to the tools, and performs as much as possible of the underlying semantic operations automatically, so that the designer can interact with the system at the ELLA level.

In future work, state bisimulation will be extended to apply to non-deterministic systems. The semantics of other computer languages, including those having asynchronous communication models, will be defined using variants of *EPA*, where the value expression algebra and action product are suitably redefined.

7 References

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Representing circuits more efficiently in symbolic model checking. In *DAC91*, 1991.

[BCMD90] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L. Dill. Sequential circuit verification using symbolic model checking. In *DAC90*, 1990.

[BFG89] H. Barringer, M. Fisher, and G.D. Gough. Fair SMG and Linear Time Model Checking. In *Proceedings of Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989. LNCS 407.

[BGHT90] R Boulton, M Gordon, J Herbert, and J Van Tassel. The HOL Verification of ELLA Designs. Technical report, University of Cambridge Computer Laboratory, 1990.

[BGL+91] H. Barringer, G. Gough, T. Longshaw, B. Monahan, M. Peim, and A. Williams. Semantics and Verification for Boolean Kernel ELLA using IO Automata. In *Advanced Research Workshop on Correct Hardware Design Methodologies (CHARME'91)*, Turin, Italy, May 1991.

[BGL+92] H. Barringer, G.D. Gough, T. Longshaw, B.Q. Monahan, M. Peim, and A. Williams. A Semantics and Verification Framework for ELLA. Technical Report UMCS-92-4-6, University of Manchester, March 1992.

[BGMW94a] H. Barringer, G. Gough, B. Monahan, and A. Williams. Verifying Deterministic Systems using a State Evolution Method. Deliverable D2.4a, Formal Verification Support for ELLA, IED 4/1/1357, University of Manchester, February 1994.

[BGMW94b] H. Barringer, G.D. Gough, B.Q. Monahan, and A. Williams. A Process Algebraic Semantics for Core ELLA. Technical Report UMCS-93-2-1, University of Manchester, November 1994.

[CBM89] O. Coudert, C. Berthet, and J-C. Madre. Verification of Synchronous Sequential Machines based on Symbolic Execution. In *Proceedings of Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989. LNCS 407.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of finite state concurrent systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[Com91] Computer General Electronic Design, Chippenham, Wiltshire, United Kingdom. *The ELLA Language Reference Manual*, 5.1 edition, February 1991.

[EP] Formal Verification Support for ELLA, IED/SERC Project 4/1/1357. Reports are available on the World-Wide Web, URL: `http://www.cs.man.ac.uk/fmethods/projects/ella-project.html`.

[GB91] Semantics and Verification Foundations for ELLA. Deliverable D1.1+2.1, FORMAL VERIFICATION SUPPORT FOR ELLA, IED 4/1/1357, University of Manchester and DRA Electronics Division, RSRE Malvern, May 1991.

[Goo90a] K.G.W. Goossens. An Operational Semantics for a Subset of the CHDDL ELLA. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, April 1990.

[Goo90b] K.G.W. Goossens. Semantics for picoELLA. Technical report, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, June 1990.

[HL93] M. Hennessy and H. Lin. Proof Systems for Message-Passing Process Algebras. Technical Report 5/93, University of Sussex, 1993.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 1979.

[LT88] N.A. Lynch and M.R. Tuttle. An Introduction to Input/Output Automata. Technical Report MIT/LCS/TM-373, MIT, November 1988.

[MC93] J D Morison and A S Clarke. *ELLA2000: A Language for Electronic System Design*. McGraw-Hill, 1993.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Moo56] E.F. Moore. Gedanken Experiments on Sequential Machines. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 129 – 153. Princeton University Press, 1956.

[MPT85] J.D. Morison, N.E. Peeling, and T.L. Thorp. The Design Rationale of ELLA, a hardware design and description language. In *CHDL*, Tokyo, 1985.

[Old91] E-R. Olderog. *Nets, Terms and Formulas*, volume 23 of *Tracts in TCS*. CUP, 1991.

[Pei92] M.D. Peim. A BDD-based equivalence checker for IO automata. Technical report, Department of Computer Science, University of Manchester, 1992.

[RS59] M.O. Rabin and D. Scott. Finite Automata and their Decision Problems. *IBM Journal of Research and Development*, 3(2):114–125, April 1959.

[Shi87] M. Shields. *An Introduction to Automata Theory*. Blackwell, Oxford, 1987.

[SOR93] N Shankar, S Owre, and J M Rushby. The PVS Proof Checker: A Reference Manual (Beta Release). Technical report, Computer Science Laboratory, SRI International, March 1993.