Computer Science
University of Manchester

The ELLA Verification Environment: A Tutorial Introduction

Howard Barringer, Graham Gough,
Brian Monahan, Alan Williams

Technical Report UMCS-94-12-2

The ELLA Verification Environment: A Tutorial Introduction

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

Produced under contract for DRA (Malvern)‡

December 1994

---

*Brian Monahan is now with Harlequin Ltd, Alderley Edge

†This report was produced by Manchester Informatics Ltd under contract for DRA (Malvern). The University of Manchester also acknowledges support from the SERC via research grant GR/F 38174.

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.

Contents

1 Introduction to the ELLA Verification Environment

The ELLA Verification Environment[1] (EVE) offers both familiar CAD tool support for the ELLA hardware description language, and formal analysis tools, integrated within the one system, accessed by a common window-based graphical user interface. The aim of the system is to enable hardware design engineers to benefit from the use of formal techniques in the design cycle by embedding them into existing technology, and providing interaction with them at the designer level. The formal analysis tools are also designed to operate mainly automatically, significantly reducing the need for formal methods expertise during design and analysis.

This tutorial first provides an overview of the EVE, and then describes the use of each component tool in detail. Several small design examples are used as illustration, and also act as reader exercises. Further examples are contained in Appendix A. A copy of the source for the examples is included in Appendix B. Different analysis techniques are currently available in the EVE, and these have correspondingly different domains of applicability. In order to help the designer choose the most appropriate technique to apply in a particular situation, a section outlining a Design and Verification Methodology is included in Appendix C. Finally some background information on the *state evolution* verification method is also given in Appendix D. Much of the material is available via the on-line help facility in the EVE.

The EVE is built on top of the LispWorks Environment[2], and it is assumed that the reader has had some experience using LispWorks and its Environment, running under X[3] — if this is not the case, then please refer to the LispWorks documentation, for example [LW94]. It is also assumed that the reader is familiar with the ELLA language; however, [MC93] provides a good tutorial into the use of the language. Note that the EVE currently accepts only Core ELLA as input, which is a covering subset of (full) ELLA. A Full-to-Core ELLA transformation tool is available separately from DRA (Malvern), although this is not currently installed into the EVE.

2 An Overview of the System

The basic design unit in the EVE is an ELLA module, which contains a sequence of ELLA declarations, similar to the usual (Unix) file. Each module is placed in the EVE database as a member of a context, which in turn can form part of a context tree. The database also stores other information about the designs, such as user-specified interdependencies between modules.

A designer can then use the EVE to invoke different operations on a module. Figure 1 shows an overall picture of several of the tools available — the detail is not important here. Moving anti-clockwise from the top left-hand corner, the first window is the top-level entry tool, containing the currently selected database, or context-tree. Below this, a text editor buffer contains one of the DELAND designs. Next, an ELLA-level rewrite system is being used to apply behaviour-preserving transformations to the design. The bottom-right window shows the Verification Condition Generation tool, with a verification clause ready for validation. Finally, a simulation run for one of the designs is shown in the Soft Logic Analysis tool.

[1]The ELLA Verification Environment has resulted from a collaborative research project, funded by the DTI and SERC, involving the University of Manchester, DRA (Malvern), Harlequin Ltd. (Cambridge) and GEC Plessey Semiconductors (Swindon).

[2]LispWorks is produced by, and a registered trademark of, Harlequin Ltd., Cambridge.

[3]X is a trademark of MIT.

A brief description of each tool is now given. First a module can be edited, either as a text object using a standard Emacs-style text editor or as a schematic. The text editor enables existing ELLA files to be imported into the EVE as text objects. The schematics editor supports the use of predefined ELLA functions, control branching via CASE statements, and a subset of the ELLA composition operators, such as indexing or sequencing. It also uses on-the-fly static semantics checking during design development, for example checking the consistency of signal types before circuit nodes are joined. The environment keeps a check on any modules that are being edited, and automatically updates any compiled versions of the modules as and when necessary.

The Soft Logic Analyser (SLA) tool is similar in many respects to a standard CAD tool simulator — it provides a graphical control and display interface to the simulation. The EVE automatically assembles and compiles the required ELLA-text for the module under simulation. The SLA supports the graphical display of inputs and outputs in a variety of formats, and has the usual display scrolling, scaling and cursor facilities. Inputs are specified by entering them via the graphic display. The values on internal signals can be monitored by the use of probes, and can be forced to a specific user-defined value by using forcers. Different simulations can be named and are stored for later recall.

The ELLA-level rewrite tool allows the designer to manipulate the form of a design, by application of behaviour-preserving transformations, or generic "laws", to the ELLA-text. This may be used for example to reduce the size of a circuit, before it is passed on to other tools in the EVE. The user can also specify groups of laws to be combined into tactics. The equivalence between ELLA descriptions related by simple transformations may also be established using the tool.

The EVE currently offers two formal analysis tools. For designs which are restricted to the booleans[4], and where the number of possible states is relatively small, then a state-machine based equivalence-checking tool can be applied to two designs to determine whether or not they are behaviourally equivalent. This compiles the two designs into state machines, called input/output automata (IOAs), and then compares the sets of traces described by the two machines. The implementation contained in the EVE uses a fully enumerated description of the state machines, thus limiting its capabilities. However, the applicability of the techniques can be extended in certain circumstances by using special encodings of the state machine. One popular technique is to use Binary Decision Diagrams (BDDs), and the EVE contains a BDD-based tautology checker for combinatorial[5] ELLA-texts (i.e. stateless designs). The BDD checker is automatically invoked by the IOA tool when it detects that the ELLA-texts under analysis are combinatorial.

A second, higher-level, approach to analysis has been included, in order to help avoid some of the problems encountered with the IOA tool, for example the rapid increase in complexity of equivalence-checking as the design size grows, and the low-level unstructured form of any feedback information supplied to the user when two designs are found to be inequivalent. A *state evolution* rule is used to establish the equivalence of two designs by effectively symbolically simulating them for a finite number of steps, and comparing the resulting output expressions, both from an arbitrary starting state, and from the initial (ground) state. If, from this information, it is possible to infer that the two outputs will remain in agreement in the next step, then the two designs are equivalent. The EVE enables a designer to establish the state evolution rule by generating a set of logical verification conditions (VCs) for the designs under analysis, using the Verification Condition Generation (VCG) tool, which is in effect a compiler from ELLA-text into a be-

[4]This restriction is not an inherent limitation of the approach — only a boolean based tool was prototyped for this system. There now exist some excellent state-based model-checking tools which could be incorporated into a production version of EVE.

[5]Again, this is not a serious limitation. Sequential circuits can be modelled/encoded using BDDs and indeed there now exist commercial tools, for example VFORMAL from COMPASS Design Automation, based on just such technology.

haviourally equivalent logical expression. The designer can then output the VCs in a format suitable for the PVS proof assistant[6] — if the validity of the VCs can be established, then the two designs are equivalent. If any VC is false, then the two designs are inequivalent, and the expressions contained within the VC can be used in debugging — they can be related back to the original ELLA-text from which they were derived, because much of the structure of the original design is retained by the VCG. Since this verification approach is carried out in stages, then the designer can intervene, for example to add design-specific information which can then be used in the theorem-prover, or to perform only a partial verification. Refer to Appendix C for a guide to the applicability of each of the two verification tools.

Finally, a symbolic simulator is included, which enables a designer to explore design structure and signal flow, by allowing arbitrary expressions as input/output, instead of just the ground values allowed with the SLA. The designer can control — in much the same way as in the SLA — what expressions are displayed, by setting circuit node monitors, and by selectively expanding or in-lining the expressions bound to nodes or function calls of interest. The output from the symbolic simulator can be displayed using an ELLA syntax, and retains much of the structure used in the original ELLA-text.

[6] PVS is an independent system available from SRI International in the public domain.

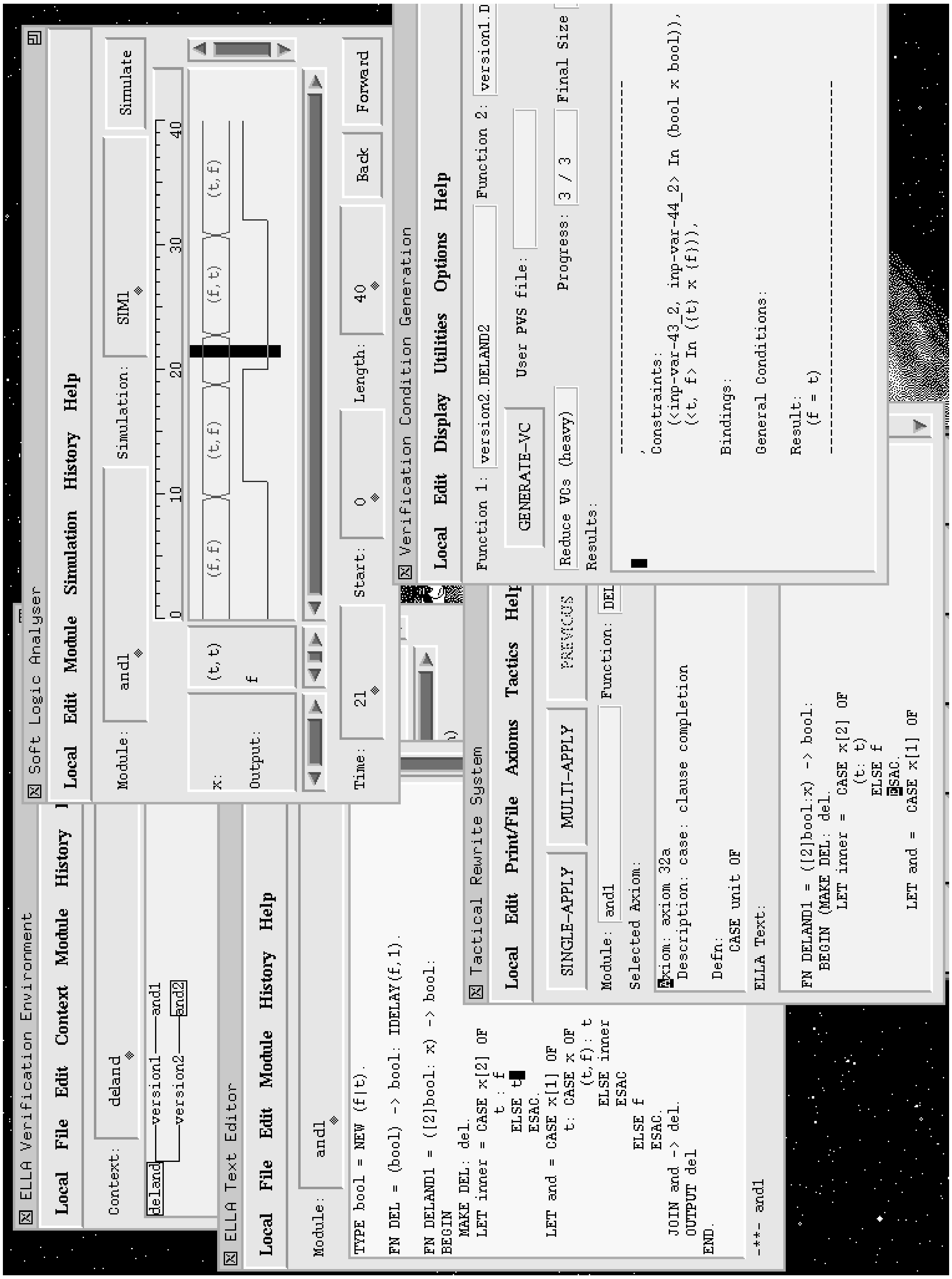

Figure 1: The ELLA Verification Environment: Overview

3 Getting Started: Entering and viewing designs

Let us now describe how to start the EVE, how to edit designs and how to store them in the database. It is assumed that the EVE system has been installed and saved in a LispWorks image, and that you have already started LispWorks and the LispWorks Environment in the usual way. Throughout the rest of this tutorial, the directory from which LispWorks/EVE has been started will be referred to as `<ella>`. It is assumed that `<ella>` has an `examples` sub-directory, containing a copy of files from the `examples` directory distributed with the EVE system. The directory `<ella>` needs to be writeable.

3.1 Starting the EVE

To start the ELLA environment, select the menu item *ELLA* under the *Appl* menu on the main LispWorks window. This will create a new window called the ELLA Verification Environment, from which all of the tools within the EVE can be invoked.

To quit the ELLA environment, then select the *Quit* menu item from the *Local* menu on the ELLA Verification Environment window. If you have modified any modules or contexts, then you will be asked if you wish to save any of these before quitting — any modifications that you have made in contexts that you choose not to save at this point will be lost. You will then be asked to confirm whether you wish to quit or not.

[**Note**: it is a good idea to always exit the ELLA environment in this way before leaving LispWorks. If you quit the system using the LispWorks environment (select *Exit LispWorks* from the *Appl* menu on the Lispworks window), then you will also quit the ELLA environment, but any changes you have made to modules/contexts will be lost — you will not be asked if you wish to save the changed contexts before quitting.]

To obtain on-line help about a particular tool, or to invoke the on-line tutorial — which contains much of the material given here — then select the *Help* menu on the tool of interest and select the *About Tool* or *Tutorial* menu items respectively. On-line help about the LispWorks environment can be obtained from the on-line manuals — select the *Manuals* menu in the LispWorks pane.

3.2 Creating a new context

To create a new context, select the *New* menu item under the *File* menu in the ELLA tool. You will be prompted for a name for this new context – enter `Demo` and press return. You will see the name `Demo` appear twice in the tool, as shown in Figure 2, once as the (top-level) context name in the Context: pane, and secondly as a single node in the main database pane, where the context/module 'tree' will be displayed.

[**Note**: A database node can be deleted or moved by using the *Cut/Paste* menu items from the *Edit* menu. A node can also be renamed by selecting the *Rename...* menu item from either the *Context* menu or the *Module* menu, depending upon whether the node is a context or a module. You can switch between different top-level contexts, either by typing the context-name into the Context: window, or by selecting the context-name from the *History* menu.]

There are now two methods for editing a module, either by using an Emacs-style text editor, or by using a schematics editor. You cannot use both methods on the same module.

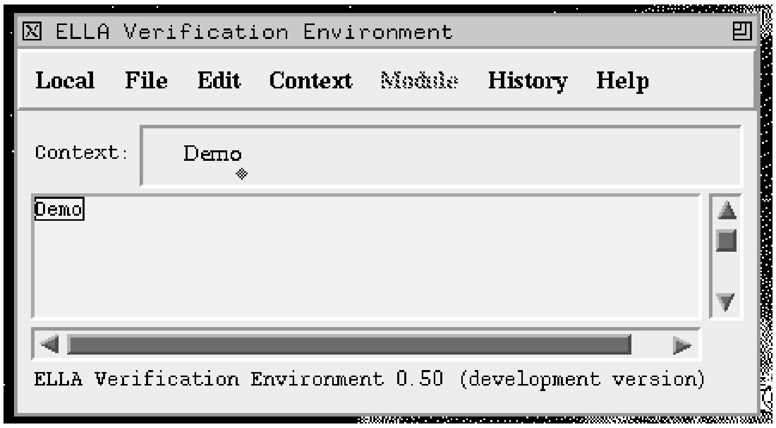

Figure 2: The EVE: Context Tree

3.2.1 Importing/Editing ELLA text

Let us now create a new module containing some existing ELLA-text. To do this, select the *Import Text...* menu item under the *Context* menu and select the file `<ella>/examples/pump.elt` in the ELLA installation directory, from the pop-up Import File window. You will be prompted for a name for this module. Enter the name `Pump` in the pop-up window and hit return. The ELLA tool will show a graph containing the module `Pump` as a child of the context `Demo` (see Figure 3). An ELLA text editor window will also appear containing the text for the file `pump.elt`[7].

You are now in a position to edit the ELLA-text module `Pump` in the ELLA Text Editor window, using standard Emacs-like commands. Any tools applied to the module will automatically operate on the copy of the ELLA-text contained in the editor buffer. For example, the pump controller can now be simulated — skip to §4.1 to find out how to do this.

3.2.2 Schematic Input

Now we will construct a simple NAND-gate design using the schematics editor, constructed from `NOT` and `OR` gates, so that `NAND(x,y) = OR(NOT(x),NOT(y))`.

Return to the top-level ELLA window, and select the `Demo` context node in the main context-tree window. Create a new sub-context by selecting the *New Context...* item under the *Context* menu. When prompted, enter the context name `Scheme` — a new 'leaf' called `Scheme` will appear in the context-tree window. Select the *New Schematic...* menu item under the *Context* menu and enter a module name `Nand` when prompted in the pop-up window. A new window called the ELLA Schematic Editor will appear, with the module name `Nand` displayed in the Module: pane.

[7] The pump controller function `CONTROL` controls the operation of three pumps `a,b,c` which pump water from a lake into a reservoir, and are switched on according to its water level, where `lv/0` is full and `lv/3` is empty. Two enumeration types `pump` and `level` denote the pump names and water level respectively. For example if the reservoir is empty, i.e. `in = lv/3`, then all three pumps should be switched on, indicated by `CONTROL = abc`. If the water level rises to `in = lv/2` then two pumps `ab` are switched on. If the water stays at `lv/2` then the two pumps `bc` will be switched on.

Importing gates: To proceed with the schematic design, you will first need to *import* the already defined `OR` and `NOT` gates into the `Nand` module. These gates are contained in the `Kernel` context. Load this context by selecting the *Open...* item on the *File* menu of the ELLA Verification Environment window, and select the `Kernel` context from the pop-up list presented. The `Kernel` context will now be displayed in the context-tree pane. Switch back to the `Demo` context by selecting the *Demo* item in the *History* menu.

Now import the two required gates into the `Nand` module. To do this, select the *Import...* menu item under the *Module* menu on the ELLA Schematic Editor window (be careful to distinguish this from the *Import →* menu item on the same menu). From the multiple-selection pop-up list, select the `Not` and `Or` entries (use the middle mouse button to make multiple selections) and click the OK button. Now you will be able to instantiate these functions in the schematic.

Editing a schematic Create an instance of the `Or` gate, by selecting the `OR` item from the *Module* ...-> *Add* menu item — notice that the function list contains the two functions `OR` and `NOT` imported into the module (together with other ELLA constructor operators). The cursor shape changes to a cross-hair, so that the instance can be placed in the main schematic pane by clicking the left mouse button. An object with two input nodes, a single output, a function name `OR` and an instance name `or_1` will appear in the pane (a graphics item can be re-positioned by dragging with the middle mouse button, and modified by clicking the right mouse button). In a similar way, create two instances of the `Not` function, to be connected to the `OR` gate inputs, by selecting the `NOT` item from the *Module* ...->*Add* menu item, and placing the instance each time as before.

Wire together the three instances, by dragging with the left mouse button from output nodes (solid pointers on the right-hand side of an instance) on the `NOT` instances to the input nodes (hollow pointers on the left-hand side) on the `OR` instance. Be careful to drop the wire precisely onto the node, because if the wire is dropped onto the background, then a separate wire node will be created, and the wire will not be connected as desired (to test that a connection has been made, then move each function instance and check that the appropriate wires move with the instance).

Finally the inputs and output to the NAND-gate need to be made. This can be done manually or automatically. To do this manually, fetch an input pin, from one of the 'input dispensers'  at the top-left or bottom-left corners of the schematics pane, by dragging the left mouse button from the dispenser around the perimeter of the schematics pane — note how the pin is constrained to the perimeter. Now wire the input pin to one of the `NOT` gates. You can connect the remaining pins automatically, by selecting the *Module* ...->*Complete Pins* menu item.

The basic schematic for the nand-gate is now finished — Figure 4 shows the state of the Schematics Editor at this stage. The corresponding ELLA-text can be viewed by selecting the *Module* ...->*Display ELLA* menu item. The ELLA-text will be displayed in the ELLA Output window, and includes the ELLA-text for the imported functions `NOT` and `OR`, as follows:

```
--- ELLA code for nand ---

TYPE bool = NEW( f | t ).

COM --- Imported "Or" --- MOC

FN OR = ( bool: a, bool: b ) -> bool:
CASE (a,b) OF
  (f,f): f
ELSE t
ESAC.
```

```
COM --- Imported "Not" --- MOC

FN NOT = ( bool: x ) -> bool:
  CASE x OF
    t: f
  ELSE t
  ESAC.

FN NAND = (bool: input_1, bool: input_2) -> bool:
 BEGIN
  MAKE NOT: not_2.
  MAKE NOT: not_1.
  MAKE OR: or_2.
  JOIN input_2 -> not_2.
  JOIN input_1 -> not_1.
  JOIN (not_1, not_2) -> or_2.
  OUTPUT or_2
 END.
```

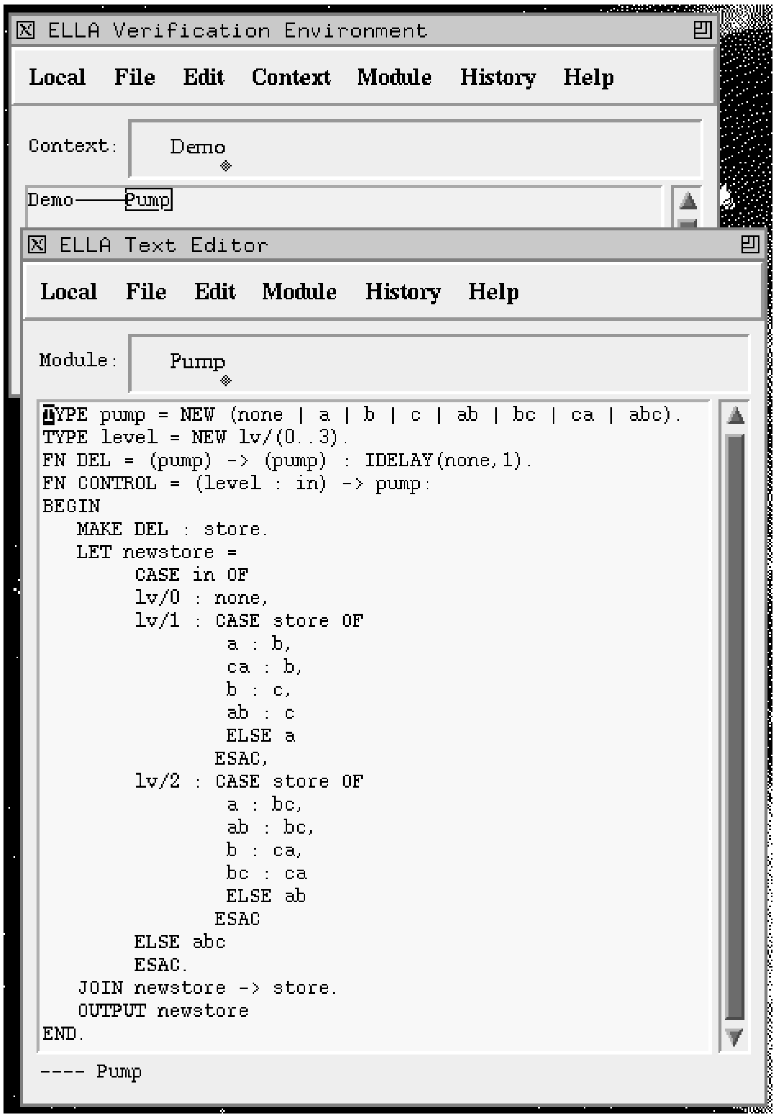

Figure 3: ELLA Text Editing in the EVE

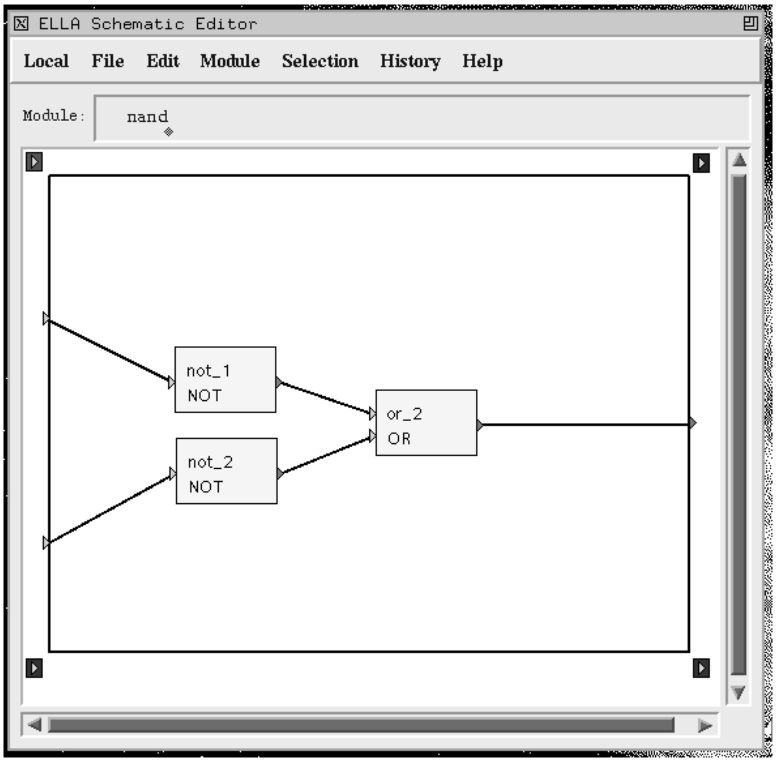

Figure 4: The Schematic Editor: Completed NAND-Gate

4 Analysis of designs

This section covers design simulation using ground and symbolic simulation tools, and then describes how to verify designs using the various analysis tools included in the EVE.

4.1 Simulation

The ELLA environment allows ELLA definitions to be simulated using the Soft Logic Analyser, which provides a graphical interface to the simulation. This supports the display of inputs and outputs in a variety of formats, the use of probes of internal signals, and forcers to set the value of internal signals to user-defined values.

To start a simulation for a particular module then select the *Simulate* menu item from the *Module* menu in the relevant window displaying the module. For example, use the *Module* menu in the Schematic Editor to simulate the Nand gate constructed above. Having done this, a Soft Logic Analyser window (SLA) will appear, with `nand` displayed in the Module: pane, as shown in Figure 5.

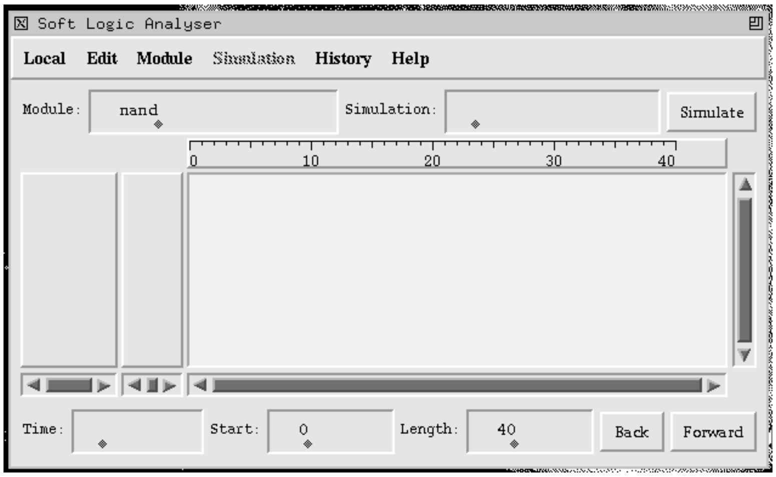

Figure 5: The SLA Startup Window

This has the following layout: the main SLA trace pane will show the waveform-like results of any simulation that you perform upon the function. The ruler above the trace pane gives the currently visible times for the simulation — the default simulation length is 40 time-steps. The far left pane lists the signal names associated with each trace, and the intermediate pane will show the current trace values (see below).

Every function can have a number of named simulations, and so the first thing you must do is create a new simulation. To do this, enter the name `sim1` into the Simulation: text input window in the SLA and hit return. A simulation of that name will now be created and displayed inside the SLA. For the nand gate, the trace pane will contain two red input traces, and a single blue output trace (assuming that you are using a colour display). The inputs default to a reasonable value — in this case to `f` — and the waveform output shows the current result, i.e. `t`. The names of each element in the trace will be displayed in the signal-name pane, i.e., `input_1`, `input_2` and `input_3`.

Using the cursor: determining the value of a trace Now place the cursor, by clicking with the middle mouse button at some point in the trace pane — the cursor will appear as a solid vertical bar, which is 1 time-step wide. The values underneath it will appear in the trace values pane, i.e. `(f, f, t)`, and the Time: pane will show the time-step underneath the cursor, e.g. 20, if the cursor is placed at time-step 20.

Entering inputs into the SLA To change the inputs in the SLA, all you need to do is click with the left mouse button on the trace pane along the trace element of the input you wish to set, and the input will be set to the nearest value to the point at which you clicked. For the two-valued booleans in the nand gate, then the effect of this is to toggle the input. You can also drag the left mouse button to change an input at several time-steps. Try setting `input_1` to `t` between time-steps 0 and 10, and `input_2` to `t` between 5 and 15.

Once you are satisfied with the inputs that you have entered, press the Simulate button and the SLA will simulate the function with the new inputs. The trace pane will be updated with the new output values, so here `output_1` will be `f` between time-steps 5 and 10. You can check the values at any point in the trace by placing the cursor with the middle mouse button.

Changing the trace display You can modify the trace pane display in a number of ways to suit your requirements, e.g. making the results easier to interpret. You can change the time-scale by clicking with the right mouse button somewhere on the ruler above the trace pane. A pop-up window will ask for a new integer scale, the default value being 10. Try entering 20 to double the scale, so that it displays time-steps between 0 and 20, approximately. You can move the trace pane around the simulation trace either by using the horizontal scroll bar on the trace pane in the usual way, by clicking the Back and Forward buttons, or by entering a time-step in the Time: pane, which places the cursor at the new point.

You can change the trace element display from the (default) waveform/enumerated, display, to a value switch display, which explicitly labels each epoch with its value. If the label will not fit in, then the epoch is left blank — if this occurs, you can view the value either by expanding the time-scale sufficiently or by using the cursor. Click the right mouse button on the trace element you wish to modify, and select the *Value Switch* menu item from the pop-up menu. Try this for signal `output_1`.

To change inputs displayed using the value-switch display mode, you can *expand* an epoch by dragging with the left mouse button — this will of course reduce the size of the neighbouring epoch, and will eventually remove it. You can insert a new-value epoch by double-clicking on the trace element with the left button, and entering a new value of the appropriate type into the pop-up window.

You can change which signal names are included in the trace display by clicking with the right mouse button in the signal-name pane. Select the signal names for display from the pop-up list, using the middle mouse button for multiple selection in the usual way — the initial list will show all of the traces as selected.

When a trace is vector-valued (which is not the case for the Nand-gate example) then you can expand the trace into its immediate sub-components by clicking with the right button. Select the *Expand* option from the pop-up list. You can then remove the expansion by clicking with the right button and selecting the *Contract* option.

Setting probes and forcers You can monitor internal circuit nodes by using *probes*, and you can force them to specific values by using *forcers*. To set a probe, select the *Probes...* menu item from the *Simula-*

tion menu. A list of internal circuit nodes will be displayed — you can make a multiple selection from the list by using the middle mouse button in the usual way, and clicking the OK button when you have finished. For example, select the two internal nodes for the nand gate, `or_1/OR/a` and `or_1/OR/b` . The trace pane will be updated, and will now display two more black trace elements labelled `or_1/OR/a` and `or_1/OR/b` — these will have values which are the logical negation of the two inputs `input_1` and `input_2` , respectively.

To set a forcer, then select the *Forcers...* menu item from the *Simulation menu* and select the internal circuit nodes you wish to force, as above. For example, select the `or_1/OR/a` input to the OR-gate. The trace pane will be updated, by the addition of the selected node name in the signal-name panel. However, the associated trace element will initially appear blank, indicating that the node is not currently being forced to any value. You can set a forcing value on a forcer in the usual way, by clicking on the trace with the left mouse button. For example, set the `or_1/OR/a` node to `f` between time-steps 5 and 15 — you will have to click twice, because the first time the value will be toggled to `t` . When you re-run the simulation, by pressing the Simulate button, notice how the value on `output_1` is affected by the forcer, which is overriding `input_1` between time-steps 5 and 15.

You can toggle the effect of forcing, by selecting the *Forcing* menu item in the *Simulation* menu. The simulation is immediately re-run without the forcer being applied, and the trace pane display is updated. If you wish to remove part of a forcer trace, you will need to switch the display style to *Value Switch* when setting values for forcers, because you cannot currently change the waveform/enumeration input to be unspecified thus removing any forcing; however, you can expand an 'unspecified' epoch to remove any specified forcing value.

Figure 6 shows the SLA simulating the NAND-gate example, having performed the operations described above.

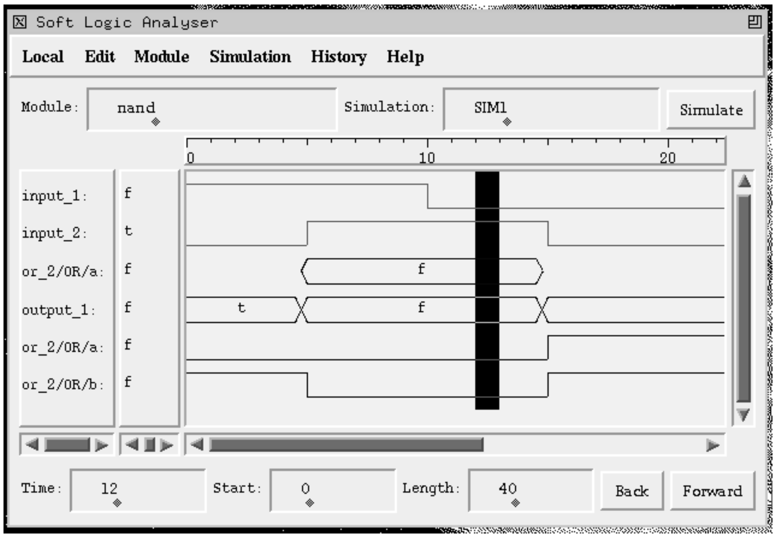

Figure 6: The SLA: Simulating the NAND-Gate

4.2 Symbolic Simulation

In addition to being able to perform (ground) simulations on a design, it is also possible to perform *symbolic* simulations, in which variables can be used to denote values within a system, rather than ground values. This can be performed using the symbolic simulator (*SymSim*). Before simulation can start, the design must first be compiled into the EPA representation[8] — unlike the SLA, this is performed explicitly in the analysis tools so that you are free to select those compiled design components which you require for analysis.

Once a design has been compiled and loaded into SymSim, then a symbolic simulation proceeds in steps; first the desired input expression is supplied, and then the expressions are simplified or evaluated as far as possible. You can then monitor circuit nodes, and perform selective expansion or in-lining on the nodes and function calls. Having analysed the results of one simulation step, then you can advance onto a new step. It is also possible to examine the simulation history. If required you can perform multiple simulation steps in one go.

Preparation: compiling the design Create a context called `del`, for example, and import the ELLA text for `<ella>/examples/deland.elt`, as described in §3.2.1 — call the new module `deland`. Now select the *Analyse* menu item under the *Module* menu. This will create a new window called the Analysis Tool, and the `deland` module will be listed in the Modules pane. Now press the EPA Compile button. This will compile the ELLA-text into EPA and then list the types and functions in the Compiled EPA Objects pane.

[Note: If you wish, you can view the complete compiled EPA expressions for DELAND by selecting *Display EPA Objects Pane ...->Print* from the *Display* menu. The output will appear in the ELLA Output window. You may also need to change the display style, for example to *EPA*, by selecting the *Select Style ...->EPA* menu item from the *Display* menu, so that you obtain a display similar to the following:

```
#----- Analysis Tool: EPA Functions + Environment -----#

#TypeName bool:#

SET bool ==  {f, t}

#Process del.DEL:#

PROCESS
   del.DEL(  ; ch-1:bool / ch-2:bool) ==
      NORMAL-FORM
          CHOICE ARGS:
             (X2 in bool)
          INPUTS:
             ch-1(X2)
          OUTPUTS:
             ch-2(STV-1)
          NEXT-STATE:
             STV-1:bool ::= {f}
                         = X2,
          LETS:
```

[8]The behaviour of an ELLA expression is represented internally in the EVE as an *EPA* (*ELLA Process Algebra*) term. The details of *EPA* are not important in this tutorial, except notethat it uses value expressions which are defined to have a very similar structure to those used in ELLA, which in many cases can in fact be reflected back into corresponding ELLA expressions. However, in this tutorial you are also invited to inspect the underlying *EPA* terms, to demonstrate how they are used in this case to represent symbolic state machines. Refer to [BGMW94a, BGMW94c] for details about *EPA* and the semantics of ELLA.

```
#Process del.DELAND:#

PROCESS
   del.DELAND(  ; ch-17:[2]bool / ch-3:bool) ==
      NORMAL-FORM
         CHOICE ARGS:
            (N-10 in [2]bool)
         INPUTS:
            ch-17(N-10)
         OUTPUTS:
            ch-3(del-3)
         NEXT-STATE:
            STV-2:bool ::= {f}
                         = and-5,

         LETS:
            inner-4 : bool ::= Case Ind(N-10, 2) :
                                  {t} -> t
                                  Else f,
            and-5 : bool ::= Case Ind(N-10, 1) :
                                {t} -> inner-4
                                Else f,
            del-3 : bool ::= STV-2,

#---------- End of EPA Environment ----------#
```

Each process definition has a *normal form* which corresponds to a symbolic representation of a state machine — the `INPUTS` and `OUTPUTs` expressions would be used to label the edges in the state machine, and the `NEXT-STATE` expressions determine the value of the next state, including the initial value(s). The `LETS` expressions bind variables, to be used in the outputs and next-state expressions.]

Select the function name `del.DELAND` appearing in the Compile EPA Objects pane and select the *Symbolic Simulation* menu item from the *Utilities* menu. This will create a new window called the Symbolic Simulation Tool, as shown in Figure 7, with the function name `del.DELAND` appearing in the Function: text pane, the current input expression `(<IN1-0, IN2-0>)` shown in the Input: text pane, and the current response expression `del-3-0` shown in the Response: pane. The Time display will read `0/0`. If your display is not at all similar to this, then perhaps you have a different selection for the Display Style — this should ideally be set to *EPA* at present.

Starting simulation To begin with, monitor all of the component EPA expressions contained in the process for `DELAND`, by selecting *Monitors ...->All* from the *Utilities* menu — these will correspond to the `LET` and state variables used in the ELLA-text for `DELAND`. All of the expressions bound to variables are now displayed in the Monitors pane.

Now press the EVAL button and the following will occur, as SymSim simplifies the current expressions for DELAND:

a) the `del-3-0` in the Response pane changes to `f`,

b) in the expressions listed under the *Lets:* section in the Monitors pane, the `STV-1-0` changes to `f`, and the indexed expressions `Ind(<IN1-0, IN2-0>, 2)` and `Ind(<IN1-0, IN2-0>, 1)` change to `IN2-0` and `IN1-0` respectively.

The *EVAL* operation simplifies the EPA expressions within the analysed process term as far as possible — when there are non-ground inputs or state variables (i.e. variables) then this is likely to lead to

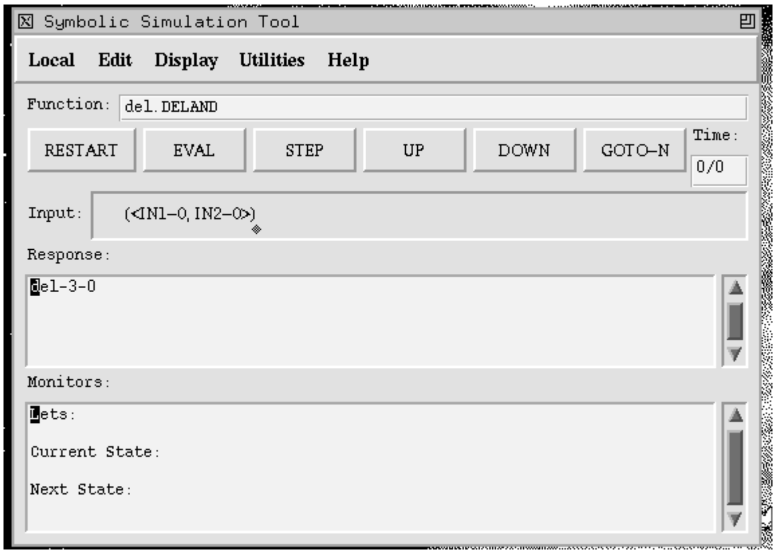

Figure 7: SymSim: Startup Window

non-ground expressions even after simplification using *EVAL*. Note that SymSim does *not* automatically undertake variable substitution, if a variable is bound to a non-ground value. For example, observe that the variable `inner-4-0` has not been replaced in the case expression bound to the let variable `and-5-0` during the above evaluation. This aspect of operation is entirely under your control, so that you can *selectively* substitute those expressions in which you are interested. If SymSim were to perform all substitutions automatically, then the size of the expressions would grow rapidly during the course of a symbolic simulation, and would soon become unmanageable.

Changing inputs You can change the input (which is by default a variable, or tuple of variables in this case) by selecting *Change Input* from the *Utilities* menu. A pop-up window will offer you the ELLA expression `(x, y)` — edit this to be `(t, t)` and press *OK* (some extra information will appear in the ELLA Output pane). Notice that `<IN1-0, IN2-0>` is now replaced by `<t,t>` in all the EPA expressions. Press the EVAL button again, and all of the EPA expressions will be simplified to either `f` or `t`. Note that the input supplied at this step will serve as the default input for any following step.

Now advance the simulation one step, by pressing the STEP button — similar expressions to the pre-evaluation step above will be displayed. Provide a variable input by selecting the *Variable Input* menu item from the *Utilities* menu and simply pressing *OK* on the resulting pop-up window. The `<t,t>` expressions are replaced by the variable expression `<IN1-1, IN2-1>`. If you now press *EVAL* then only some of the expressions will be simplified to `t` or `f`.

It is also possible to observe the effect of evaluation on a mixed ground/non-ground input. For example, set the input to be `(t,y)`, using the *Change Input* menu item from the *Utilities* menu. Notice that any variable component of the input must correspond to the variable name offered in the Variable Input dialogue — it would be an error to have entered `(t, myvar)` here, for example. Again, simplify the expressions by pressing the EVAL button.

When you have performed several simulation steps, then you can review the simulation history by using the *UP*, *DOWN* and GOTO-N buttons. You can start a fresh simulation by pressing the RESTART button — any previous simulation history will be lost when you do this however.

Expansion and monitoring Now try substituting variables by the expressions bound to them — for example select *Expand*...`->`*Current Lets* from the *Utilities* menu, and notice that the variables `inner-4-1` and `and-5-1` are replaced by the expressions to which they are bound. The expansion can be removed by selecting the *Expand*...`->`*Undo* menu item from the *Utilities* menu, and the unexpanded expressions can be retrieved by selecting the *Expand* ...`->`*None* menu item from *Utilities*.

If you wish to expand nodes from previous time-steps, then use the *Old Lets* or *Old States* menu items from the *Utilities* ...`->`*Expand...* menu. These produce a pop-up window in which you can enter the time-step of the nodes you wish to expand. By default, it offers the previous time-step, or the time-step that you entered on your last expansion.

Note that each variable name has the form `name-m-n` where `name` is the base name, taken from the ELLA text in the case of Let variables, `m` is a unique identity number, and `n` is the time-step to which the variable belongs. Thus you can ascertain the time-step to which a node belongs by examining the final number in its name. You can also distinguish between Let variables and state variables, because the latter have names of the form `STV-m-n`.

To expand specific nodes only, i.e. not just all of the Let or State variables from a particular time-step, then use the *Add Vars...* menu item from the *Utilities*...`->`*Expand...* menu. This again asks for the time-step of the nodes you wish to expand, but then offers you a multiple-selection list of the nodes from that time-step, which you can select for expansion.

You can selectively monitor nodes by using facilities on the *Utilities*...`->`*Monitors...* menu. Initially in this example, you selected all of the nodes for monitoring. Any individual group of nodes — (current) Lets, Current State, Next state — can be monitored by selecting the appropriate menu item. You can also add or remove arbitrary nodes from those monitored by using the *Add...* and *Remove...* menu items — these each offer a multiple-selection list containing all of the unmonitored circuit nodes, or all of the monitored nodes, respectively. When you select from these lists then the nodes will be added or removed from those nodes already monitored. The display will be updated appropriately. Finally, you can remove all monitors by using the *None* menu item. Note that any monitors set at the current time-step will by default be set in the following time-step, thus if you need to monitor a fixed set of nodes throughout a simulation, then you only need to select the nodes for monitoring once.

With the selective use of monitoring and expansion you can explore the structure of the design while maintaining control of the size of the expressions displayed. For example, start a fresh simulation by pressing the RESTART button and set the input to be `(t,y)`. Use the *Add..* menu item from the *Utilities* ...`->`*Monitors...* menu to monitor the nodes `del-3-0` and `STV-2-0`. Perform a couple of simulation steps, by pressing *EVAL* and *STEP* alternately. Now repeatedly expand Let and State variables, both current and past, until they have all been eliminated, The result will be similar to that shown in Figure 8, where two simulation steps have been performed. The display shows that the output is dependent on the previous `y` input, at step 1, and the next state is dependent on the current `y` input.

4.3 Verification

One of the main features of the EVE is that it gives the capability of formally verifying that the behaviours of two designs are equivalent (or perhaps that one behaviour is contained in the other), using a variety of

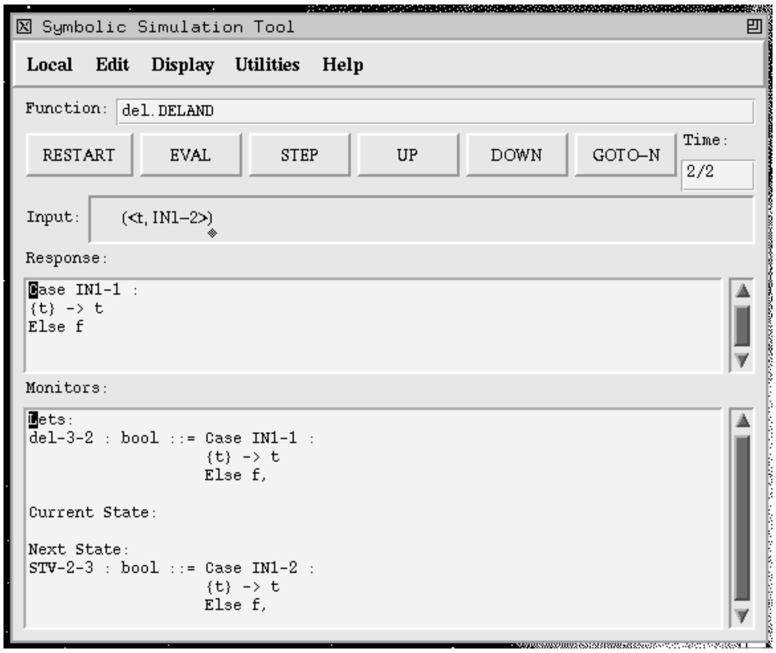

Figure 8: SymSim: Symbolic Simulation of the delay AND-Gate

techniques. In this section, we will introduce design comparison first using a state-based approach and then using symbolic techniques. The suitability of these methods to different types of design is discussed in Appendix C.

4.3.1 IOA Comparison

For designs containing standard boolean types (e.g. `TYPE bool = (f|t)`, `[2]bool`) a state-based compilation/verification tool is provided, which compares the trace-sets of two designs under analysis (for state-less designs, a binary decision diagram (BDD) encoding of this method is used). The two designs under analysis are first compiled into input/output automata which are then compared for trace-equivalence or containment. The method is fully automatic, and results in a yes/no answer. If the designs are not equivalent then counter-example traces can be displayed, showing possible paths leading to the divergence in behaviours.

Preparation: loading the designs Quit any Analysis tool window you have running, and create a `twist` context in the ELLA Verification Environment window, with two sub-contexts `spec` and `impl`. Import the files <ella>/examples/twist_spec.elt and <ella>/examples/twist_impl.elt into the Environment, calling the modules `twist_spec` and `twist_impl`, for example.

Now analyse both `twist_spec` and `twist_impl` modules, by selecting each module and then selecting the *Analyse* menu item from the *Module* menu — the two module names should appear in the Modules pane of the resulting Analysis Tool window.

Invoking the comparator Select the *IOA Compare* menu item from the *Options* menu and then simply press the IOA Compare button. The designs will then be compiled into IO automata (IOAs), and their behaviours checked for equivalence. If the two designs are not equivalent then the *counter-example* traces, or paths leading to a divergence in behaviour, can be displayed. In this case, a deliberate mistake has been introduced into `twist_spec.elt`, so that the message

```
------------------------
1st not <= 2nd
------------------------
```

is displayed in the ELLA Output window, and a Show Traces? pop-up pane offers to display the error traces. Click the Yes button in the Show Traces? pop-up pane, and again in the next Show another trace? pop-up pane, so that the (only) two error-traces are displayed in the ELLA Output window. These have the following interpretation: the inputs to the IOA are labelled `-1, -2,...` and the outputs are labelled `1,2,3...`, separated by a `>`. Each row then displays an element in the trace, with (`|` = `f`), (`%%` = `t`), and `E` or `EE` indicating a difference in the behaviours of the two IOAs.

```
-1   >   1   2   3
========================

|    >   |   |   |
|    >   |   |   %%
|    >   |   %%  %%
|    >   %%  %%  %%
|    >   %%  %%  |
|    >   %%  |   |
|    >   |   |   E
```

```
-1   >   1   2   3
========================

|    >   |   |   |
|    >   |   |   %%
|    >   |   %%  %%
|    >   %%  %%  %%
|    >   %%  %%  |
|    >   %%  |   |
%%   >   |   |   E
```

For example the third row '`| > | %% %%`' of the first trace has an `f` on the input (reset) and `<f,t,t>` on the outputs. The row '`| > | | E`' again has an `f` on the input, but the output of one machine is now `<f,f,f>` and the output of the second machine is `<f,f,t>`. This means that the behaviours of the two machines will differ on the 7th time-step, having kept the input set to `f`. You can see the cause of the difference by examining the ELLA text files for the two designs.

Now edit the erroneous design, by clicking on the module in the *ELLA Verification Environment* window and then selecting the *Edit* menu item from the *Module* menu. Correct the mistake, and simply press the IOA Compare button again in the Analysis Tool window — you do not have to re-load the designs into the Analysis Tool. After a few seconds a pop-up window displaying "*Machines Twist_spec and Twist_impl are equivalent*" will confirm that the two designs are equivalent — assuming that you have properly corrected the mistake.

Stateless design comparison Create a new context, called `alu_spec` for example, and import the file `<ella>/examples/alu16_spec.elt`, calling the module `spec16`. Create a second context `alu_impl` and import the file `<ella>/examples/alu16_spec.elt` calling the module `impl16`. Remove any modules in the Modules pane of the Analysis Tool, by using the *Cut* menu item on the *Edit* menu, and then analyse the two modules, by first selecting each module in the ELLA Verification Environment context-tree, and then selecting the *Analyse* menu item from the *Module* menu.

In the Analysis Tool window, check that the *IOA Compare* menu item is selected in the *Options* menu — the button should be labelled *IOA Compare*. Now simply press the IOA Compare button to start the design comparison. After a few seconds a pop-up window displaying "*Machines spec16 and impl16 are equivalent*" will indicate that the two designs are equivalent. Note that the BDD comparator was automatically invoked on these designs, because they do not contain any delay elements.

You could now try introducing a deliberate error into one of the designs and repeating the comparison — you should then obtain a set of counter-example input/output pairs showing how the behaviours of the two designs differ (a default maximum of 20 pairs are available).

You will find several files deposited in `<ella>`; these contain various descriptions of the IOAs that are generated during IOA compilation/comparison. You can safely delete these when you have finished the exercise.

4.3.2 Verification Condition Generation (VCG)

A second *symbolic verification* system is installed in the EVE, which is suitable for larger, structured designs involving complex datatypes — see Appendix C for a discussion. This semi-automated verification approach is based on the *state evolution* rule. It determines the equivalence of two designs by effectively symbolically simulating them for a finite number of steps and comparing the resulting output expressions, starting from both an arbitrary state, and the initial (ground) state. The state evolution rule is established by using the VCG to generate a set of logical expressions, or verification conditions (VCs) — if these can be proven then the two designs being analysed are equivalent. If any VC is invalid, then the two designs are inequivalent, and the VCs can be utilised during debugging. The VCG is effectively another compiler, enabling ELLA-text to be translated into a set of behaviourally equivalent logical expressions. The proof of the VCs generated is performed separately, using for example a standard theorem-prover/proof assistant such as PVS — the VCG can output the VCs in a format suitable for PVS.

The VCG also performs extensive simplification on the VCs as they are being generated and is able to reduce the final number produced, by removing those VCs which it can determine to be valid.

For further details of the state evolution method for verification, then see Appendix D.

Example 1: Delay AND Gate This first example checks the equivalence of two implementations of the basic delay AND gate, one of which was used to illustrate the SymSim in §4.2. The aim of this example is to introduce the basic concepts and operation of the symbolic verification approach using the VCG. However, note that for this size and type of design, it may be better to use the IOA comparison tool.

Analyse the files `<ella>/examples/deland.elt` and `<ella>/examples/deland2.elt` by importing the texts into contexts named `del` and `del2` respectively, and calling the modules `deland` and `deland2`. Note that if you followed the exercise for SymSim in 4.2, then you will already have installed `deland.elt` into the EVE, so you do not have to repeat this. Then click on each module and select the *Analyse* menu item on the *Module* menu, so that they are loaded into the Analysis Tool. Make sure you remove any other modules from the Modules pane of the Analysis Tool window, by using the *Cut* menu item on the *Edit menu*, or by quitting the Analysis Tool first.

Start Before invoking the VCG, you must first compile the two designs into EPA terms: compile the modules in the Analysis Tool by pressing the EPA Compile button — if the button label reads *IOA Compare* then first switch to the EPA compiler by selecting the *EPA Compile* menu item in the *Options* menu. A list of the types and functions compiled will appear in the Compiled EPA Objects: pane. Select `del.DELAND` and `del2.DELAND2` from this multiple-selection list. Now select the *VCG* menu item from the *Utilities* menu. This will create a new window called the Verification Condition Generation tool with the names of the selected functions, i.e. `del.DELAND` and `del2.DELAND2`, appearing in the panes Function 2 and Function 1 respectively — if the names are swapped around here, then that is because the two modules appear in a different order in the Modules pane of the Analysis Tool. For this example, the order is not important.

In order to run the VC generation phase of the verification cycle, all you need to do is to supply the number of steps, or level, for the state evolution (refer to the introduction of this section (§4.3.2), or Appendix D for a more detailed explanation of this). The VCG will then automatically produce and simplify the necessary VCs.

Press the GENERATE-VC button in order to start the VCG. When prompted, supply the number of VC steps required — for the `DELAND` example only one level will be needed (i.e. the default value). The VCG will then take a few seconds to complete here.

[**Note**: the current operation being performed by the VCG appears in the untitled mode pane located directly below the GENERATE-VC button. This will show *Initial VCG* and then *Step VCG* as the VCG generates expressions for the two premisses in the state evolution rule (see Appendix D). The Progress and Merge Size panes will also be updated as VC generation progresses — the former shows the number of antecedent expressions processed, and the latter gives an indication of the size and number of the two VC sets being "merged": in each pair (x, y), x is the number of expressions in one VC in the set, and y is the number of VCs in the set]

For this particular example, the VCG detects that there are no (non-trivial) VCs resulting, and so it indicates that the two functions are equivalent. Note that now the Final Size: pane shows the size and number of VCs representing the initial and step premisses — in this case both figures are $(0, 0)$ indicating that the two sets of VCs are empty.

Generating non-trivial VCs You can now try repeating the above procedure, using the DELAND example when a deliberate error has first been introduced, so that non-trivial VCs will be generated by the VCG (they will of course be invalid, showing that the two designs are inequivalent). For example, select the `deland` module in the Modules: pane of the Analysis Tool, and select the *Edit* menu item from the *Module* menu. In the resulting ELLA Text Editor containing the ELLA-text for `DELAND` change the expression:

```
LET inner = CASE y OF
              t : t
            ELSE f
            ESAC.
```

to read:

```
LET inner = CASE y OF
              t : f
            ELSE t
            ESAC.
```

Now re-run the above procedure starting from **Start**.

Inspecting VCG results When the VCG has completed, VCG results can be inspected using the *Display* menu. The *Display VC...* menu item enables objects created by the VCG to be selected and displayed; the output can be pretty-printed to the Results display pane, or to a file, using the *Print* and *File* menu items, respectively. If you wish to store the output in a format suitable for reloading the objects back into the environment, then use the *Dump* menu item. For example, try displaying the *step* VCs for the current DELAND example (there are no *initial* VCs here, so there is not any point in viewing them). First check that the display style is set to *EPA*, using the *Display ...->Select Style...* menu. Now select the *Display VC ...->Print VC* menu item from the *Display* menu. Select *Step VCs* from the list presented; you will be asked whether you wish to change the display style to *EPA-AS-PVS* — answer no to this for now. The VCs will then be displayed in the Results panel, which should now contain a text similar to the following:

```
;;Step VCs:

 Global Vars:
   inp-73_1 : [2]bool,
   rhs-st-75 : bool,
   lhs-st-74 : bool,
   STV-3 : bool,
   STV-1 : bool,
   inp-73 : [2]bool,

 Variable Bindings:

 Variable Constraints:

 Antecedent Forms:
      (del2.RSP_DELAND2(inp-73, STV-1) = del.RSP_DELAND(inp-73, STV-3)),
      (rhs-st-75 = del.EVL_DELAND(inp-73, STV-3)),
      (lhs-st-74 = del2.EVL_DELAND2(inp-73, STV-1)),

 Goal Formula:
       (del2.RSP_DELAND2(inp-73_1, lhs-st-74) = del.RSP_DELAND(inp-73_1, rhs-st-75))

 Verification Conditions:

        Local Variables :
           inp-var-66_2 : bool,
           inp-var-67_2 : bool,

        Local Assignments :

        Clauses :

             Constraints:
                (<t, t> In ({t} x {t})),
                (<inp-var-66_2, inp-var-67_2> In (bool x bool)),

             Bindings:

             General Conditions:

             Result:
                (t = f)
             -------------------------------------------
             ,
             Constraints:
                (<inp-var-66_2, inp-var-67_2> In (bool x bool)),
                (<t, f> In ({t} x {f})),

             Bindings:
```

```
General Conditions:

Result:
   (f = t)
------------------------------------------
```

Apart from the (simplified) set of two VCs at the end, this contains the step premiss from the state evolution rule (see Appendix D) to which the VCs are equivalent:

```
Antecedent Forms:
     (del2.RSP_DELAND2(inp-73, STV-1) = del.RSP_DELAND(inp-73, STV-3)),
     (rhs-st-75 = del.EVL_DELAND(inp-73, STV-3)),
     (lhs-st-74 = del2.EVL_DELAND2(inp-73, STV-1)),

Goal Formula:
      (del2.RSP_DELAND2(inp-73_1, lhs-st-74) = del.RSP_DELAND(inp-73_1, rhs-st-75))
```

where `del.RSP_DELAND`, `del2.RSP_DELAND2` are the response functions, and `del.EVL_DELAND`, `del2.EVL_DELAND2` are the evolution functions for `DELAND` and `DELAND2`, respectively. This logical expression says that if the response functions are equal at one (arbitrary) time-step, starting from arbitrary states, then the response functions must agree at the next time-step, for all possible inputs.

The VCs generated are thus now non-trivial, and can be disproved, showing that the two designs are not equivalent in this case. The VCs can be analysed by hand, or can be supplied to a theorem-proving tool such as PVS for validation (see below for details of how to convert the VC contexts to a format suitable for PVS).

If you wish, you can also display other objects produced by the VCG, such as the response and evolution functions, by selecting them from pop-up list given by the the *Display VC ...->Print VC* menu item from the *Display* menu.

Example 2: Integer Latch Load the file `<ella>/examples/simple-latches.elt` into the environment as before, placing it in a `simple_latches` module in a `latches` context, for example. The file contains two implementations of an integer latch design, which are to be checked for equivalence. You can examine the ELLA-text by invoking the text editor on the `latches` module.

The function `F` takes a control input `cntrl` which determines its action: if `cntrl = read`, then it stores and outputs the current input integer. Otherwise, when `cntrl = ok` it outputs the sum of the stored value and the current input, but retains the old store value. The second function `G` performs the same operation, but internally it splits the stored integer into a ‘low’ part and a ‘high’ part, and stores these separately — the two parts are recombined on output. Checking that `F` and `G` are equivalent will therefore involve some reasoning about integer arithmetic expressions, when validating the VCs generated. Also note that the datatypes involved are clearly much more complex than in the previous example, which only used two-valued booleans.

As before, load the module into the Analysis Tool, removing any other modules which may be loaded. Compile the module by pressing the EPA Compile button. Select the two functions `latches.F` and `latches.G` from the Compiled EPA Objects pane. Now select the *VCG* menu item from the *Utilities* menu, which will pop up the Verification Condition Generation Tool window with the names of the selected functions, i.e. `latches.F` and `latches.G`, appearing in the panes Function 1 and Function 2.

Start the VCG as before, by pressing the GENERATE-VC button. Again, the design only requires one step, or level, in the state evolution rule, and so enter the default value ‘1’ for the number of levels, when prompted. When the VCG has completed, display the resulting VCs as before, using the *Display VC ...->Print VC* menu item from the *Display* menu. There will be a single VC for the *initial* premiss, and two VCs for the *step* premiss.

Further VC simplification The VCs generated contain structured expressions involving variable bindings. When trying to prove the VCs it may be easier if the bindings are removed, using substitution, thus potentially reducing the number of expressions in each VC. The VCG provides both ‘light’ and ‘heavy’ expression reduction — the light option only substitutes variables bound to simple expressions (ground values, or other variables), while the heavy option substitutes all variables. These operations are invoked by selecting either the *Reduce VCs (light)* menu item or the *Reduce VCs (heavy)* menu item from the *Utilities* menu. Try these out, and observe their effect on the VCs — you will need to re-display the VCs after each operation.

VC conversion into PVS format Now you can convert the (heavily reduced) VCs generated for the integer latch into a format suitable for the PVS theorem-prover. You will need an additional axiom about integer arithmetic in order to prove that the latch designs are equivalent.

```
div_ax: FORALL (x,y: int):
        ((y /= 0) IMPLIES ((mod(x,y) + y * div(x,y)) = x))
```

i.e. this will enable us to show that splitting the stored integer into ‘low’ and ‘high’ parts, and then re-assembling the two parts, results in the same integer. The axiom can be included by selecting the *Load PVS File* item from the *Utilities* menu and loading the file `<ella>/examples/div-axiom.pvs` (of course you could edit the PVS text later). The filename `div-axiom.pvs` will appear in the User PVS File pane. The axiom will be included in the VC contexts when you use the *EPA-AS-PVS* display style.

Now select the *Convert to PVS* menu item from the *Utilities* menu, and this time change to the PVS display style by selecting the *Select Style ...->EPA-AS-PVS* item from the *Display* menu. Re-display the VCs in the same way as above. Notice that the format has been modified and additional type and function declarations have been included. Notice also that the ‘`div_ax`’ axiom appears in the text. The Results pane should now contain text similar to the following:

```
;;Step VCs:

epa_vcgs :  THEORY
   BEGIN

   %%% Global Vars:
      epa_int : TYPE = { x: int | (x >= -2147483648) AND  (x <= 2147483647) }
        control: DATATYPE
         BEGIN
            ok : ok?
            read : read?
         END  control
        epa_bool: DATATYPE
         BEGIN
            false : false?
            true : true?
         END  epa_bool
```

```
      data_intrng : TYPE = { x: int | (x >= 0) AND  (x <= 2000) }
      tag_i: DATATYPE
       BEGIN
          i : i?
       END  tag_i
    data: TYPE = [ tag_i , data_intrng ]
      mod : [ epa_int, epa_int -> epa_int ]
      latches_rsp_f : [ [ data, control ], data -> data ]
      latches_rsp_g : [ [ data, control ], data, data -> data ]
      div : [ epa_int, epa_int -> epa_int ]
      latches_evl_g : [ [ data, control ], data, data -> [ data, data ] ]
      latches_evl_f : [ [ data, control ], data -> data ]
      inp_50_1 : VAR [ data, control ]
      rhs_st_52 : VAR data
      rhs_st_53 : VAR data
      lhs_st_51 : VAR data
      stv_5 : VAR data
      stv_6 : VAR data
      stv_4 : VAR data
      inp_50 : VAR [ data, control ]

%%% User Declarations:
        div_ax: AXIOM (FORALL (x: int), (y: int):
                        ((y /= 0) IMPLIES (mod(x,y) + y * div(x, y) = x)))


%%% Verification Conditions:

        %%% Local Variables :

           arith_var_66 : VAR data_intrng
            arith_var_65 : VAR data_intrng
            arith_var_64 : VAR data_intrng
            arith_var_63 : VAR data_intrng
            arith_var_62 : VAR data_intrng
            inp_var_42_3 : VAR data
            inp_var_42_2 : VAR data
           %%% Clauses :
            clause_1: THEOREM

           (
           %% Constraints:
              ((ok = ok))
                 AND ((ok = ok))
           AND
           %% Bindings:

              (arith_var_62 = proj_2(inp_var_42_2))
                 AND (arith_var_66 = proj_2(stv_6))
                 AND (arith_var_65 = proj_2(stv_5))
                 AND (arith_var_64 = proj_2(stv_4))
                 AND (arith_var_63 = proj_2(inp_var_42_3))

           AND
           %% General Conditions:

              ((i, (arith_var_63 + arith_var_64)) =
               (i, (arith_var_63 + (arith_var_65 + arith_var_66))))

           %% Result:
           )
            IMPLIES

              ((i, (arith_var_62 + arith_var_64)) =
               (i, (arith_var_62 + (arith_var_65 + arith_var_66))))
           %% -------------------------------------------
```

```
          clause_2: THEOREM

         (
         %% Constraints:
            ((ok = ok))
               AND ((read = read))
         AND
         %% Bindings:

            (arith_var_63 = proj_2(inp_var_42_3))
               AND (arith_var_62 = proj_2(inp_var_42_2))

         %% Result:
         )
          IMPLIES

            ((i, (arith_var_62 + arith_var_63)) =
                  (i, (arith_var_62 + (mod(arith_var_63, 1000)
                       + (1000 * div(arith_var_63, 1000))))))
         %% -------------------------------------------
END epa_vcgs
```

The VCs can now be output to files, so that you can load them into PVS. Output one set of VCs, for example the *Step VCs*, by selecting *Display VCs* ...->*File VCs* item from the *Display* menu. Having selected the objects to output, you will then be prompted for a filename — enter `simple-latch`, for example. The VCG will use this to generate a set of suitable filenames for the objects you have selected to output: if you supply a filename `<filename>`, having a basename `<file>` and optional extension `<ext>`, then the objects will be placed in the following files by the VCG:

- if you have selected *Initial VCs* then each VC will be put into a separate file, named `<file>_initX.<ext>`, where `X` is a number,
- if you have selected *Step VCs* then each VC will be put into a separate file, named `<file>_stepX.<ext>`, where `X` is a number,
- any other objects will be placed in `<filename>`.

The filename extension defaults to 'pvs', because this is the default used by PVS. Thus in this case, you will get two files, called `simple-latch_step1.pvs` and `simple-latch_step2.pvs`. Each file contains a PVS theory with a single theorem, corresponding to one of the VCs. It also contains the `div_ax` axiom input above.

Using PVS:

[Note: PVS is a separate stand-alone theorem-proving system. Please refer to the PVS documentation and local instructions for details of its operation and how to run PVS on your system. The following is a brief outline of the steps required to prove the VCs using PVS.]

You have completed the VC generation phase of verification, and now need to establish the validity of the VCs. Start PVS and import the file `simple-latch_step1.pvs` using the PVS command

```
M-x import-pvs-file
```

This should contain the theorem labelled `clause_1` from above. You can now use PVS to construct a proof for this theorem using the following PVS commands:

Command	Description
`M-x prove`	invoke the PVS parser and type-checker, followed by the prover if the expression is well-formed
`(skolem!)`	introduce Skolem constants for universally-quantified variables
`(flatten)`	disjunctive simplification – transform into sequent form
`(simplify)`	simplify using PVS decision procedures
`(assert)`	simplify using PVS decision procedures and prove if possible

To construct a proof for the theorem `clause_2` contained in `simple-latch_step1.pvs`, then load the file into PVS and perform the following PVS operations, which includes instantiation of the `div_ax` axiom: commands:

Command	Description
`M-X prove`	invokes the PVS parser/type-checker, and starts the prover
`(skolem!)`	
`(flatten)`	
`(auto-rewrite "div_ax")`	installs axiom `div_ax` as an automatic rewrite rule, to be used for simplification by the `assert` command the sequent
`(assert)`	

The following script is taken from PVS, showing the proof of theorem `clause_2`, using the above commands:

```
<rcl>
clause_2 :

  |-------
1    (FORALL (arith_var_62, arith_var_63: data_intrng),
             (inp_var_42_2, inp_var_42_3: data):
        (((ok = ok))
           AND ((read = read))
             AND (arith_var_63 = proj_2(inp_var_42_3))
               AND (arith_var_62 = proj_2(inp_var_42_2)))
          IMPLIES
          ((i, (arith_var_62 + arith_var_63))
             = (i,
                 (arith_var_62
                    +
                    (mod(arith_var_63, 1000)
                       + (1000 * div(arith_var_63, 1000)))))))

Rule? (skolem!)
For the top quantifier in 1, we introduce Skolem constants:
     (arith_var_62!1 arith_var_63!1 inp_var_42_2!1 inp_var_42_3!1)
```

```
this simplifies to:
clause_2 :

  |-------
1    (((ok = ok))
        AND ((read = read))
          AND (arith_var_63!1 = proj_2(inp_var_42_3!1))
            AND (arith_var_62!1 = proj_2(inp_var_42_2!1)))
        IMPLIES
        ((i, (arith_var_62!1 + arith_var_63!1))
           = (i,
              (arith_var_62!1
                 +
                 (mod(arith_var_63!1, 1000)
                    + (1000 * div(arith_var_63!1, 1000))))))

Rule? (flatten)
Applying disjunctive simplification to flatten sequent,
this simplifies to:
clause_2 :

-1    ((ok = ok))
-2    ((read = read))
-3    (arith_var_63!1 = proj_2(inp_var_42_3!1))
-4    (arith_var_62!1 = proj_2(inp_var_42_2!1))
  |-------
1    ((i, (arith_var_62!1 + arith_var_63!1))
        = (i,
           (arith_var_62!1
              +
              (mod(arith_var_63!1, 1000)
                 + (1000 * div(arith_var_63!1, 1000))))))

Rule? (auto-rewrite "div_ax")
Installing automatic rewrites:
  div_ax,
this simplifies to:
clause_2 :

[-1]    ((ok = ok))
[-2]    ((read = read))
[-3]    (arith_var_63!1 = proj_2(inp_var_42_3!1))
[-4]    (arith_var_62!1 = proj_2(inp_var_42_2!1))
  |-------
[1]    ((i, (arith_var_62!1 + arith_var_63!1))
        = (i,
           (arith_var_62!1
              +
              (mod(arith_var_63!1, 1000)
                 + (1000 * div(arith_var_63!1, 1000))))))

Rule? (assert)
Rewriting (mod(arith_var_63!1, 1000) + (1000 * div(arith_var_63!1, 1000))) to
    arith_var_63!1.
Invoking decision procedures,
Q.E.D.


Run time  = 5.03 secs.
Real time = 25.45 secs.
NIL
<rcl>
```

4.4 Rewriting designs

In this section, we will show how the ELLA Rewrite Tool (ERT) can be used to manipulate ELLA-text, for the purposes of design exploration and reduction, or to show the equivalence of two designs which are related by a sequence of simple transformations.

Return to the ELLA tool and select the top-level `Demo` context. Create a new context by selecting the *Context*...->*New Context* menu item, and entering the context name `And` when prompted. Now import the file `<ella>/examples/and.elt` by selecting the *Context*...->*Import Text...* menu item, and entering a module name `And_Gate` when prompted. The `And_Gate` module will appear in the context tree as a member of the `And` context, and the text for the module will appear in an ELLA Text Editor window. Three functions are defined, where the function `TEST` is simply a more complex implementation of an AND-gate. The ERT will be used to simplify `TEST` down to the single CASE statement representing an AND-gate.

Select the *Module*...->*Rewrite* menu item. A new window called the Tactical Rewrite System will appear. The module name `And_Gate` appears in the Module: pane, and its text is shown in the ELLA Text: pane. Indicate that the function `TEST` is to be rewritten, by selecting `TEST` from the pop-up list presented by the *Print/File*...->*Select Function...* menu item. The name `TEST` will appear in the Function: pane.

The procedure for rewriting the ELLA function is to *select* an applicable transformations, which will then be displayed for inspection. When you are satisfied that you have chosen the correct transformation then you apply it to the function. For the `TEST` function, select the *Axioms* ...->*select...* menu item, which presents the list of axioms which are applicable to the current version of `TEST`. Initially only a single `AXIOM16` is applicable, so click on this — its description will appear in the Selected Axiom: pane. Now apply `AXIOM16` by pressing the SINGLE-APPLY button. The rewritten text will then appear in the ELLA Text: pane. The same axiom can be re-applied a second time, by pressing the SINGLE-APPLY button again, or repeatedly (until no further changes are detected) by pressing the MULTI-APPLY button. A different axiom can be applied by choosing an axiom from the list of applicable axioms presented by the *Axioms*...->*Select ...* menu item. To simplify `TEST` then the following axioms should each be applied repeatedly until no more change is observed:

Axiom	Description
`AXIOM16`	replace a function call by its body
`AXIOM17`	return the indexed element from a row
`AXIOM32A`	'complete' the CASE statement choices so that they cover the chooser set
`AXIOM33`	combine case elements into rows
`AXIOM36`	simplify nested CASE statements

This will result in a single CASE statement representing a NAND gates, as required. Figure 9 shows the ERT with function `TEST`, after two applications of `AXIOM16`.

The sequence of transformations can be reviewed by pressing the PREVIOUS and NEXT buttons. You can also restrict the displayed ELLA-text to the function that is being rewritten, by selecting the *PRINT-FUNCTION* option from the *Print/File* menu. The complete ELLA closure can be viewed again by selecting the *PRINT-CLOSURE* option.

A collection of axioms may be grouped together into a tactic. First, the tactic must be defined using the

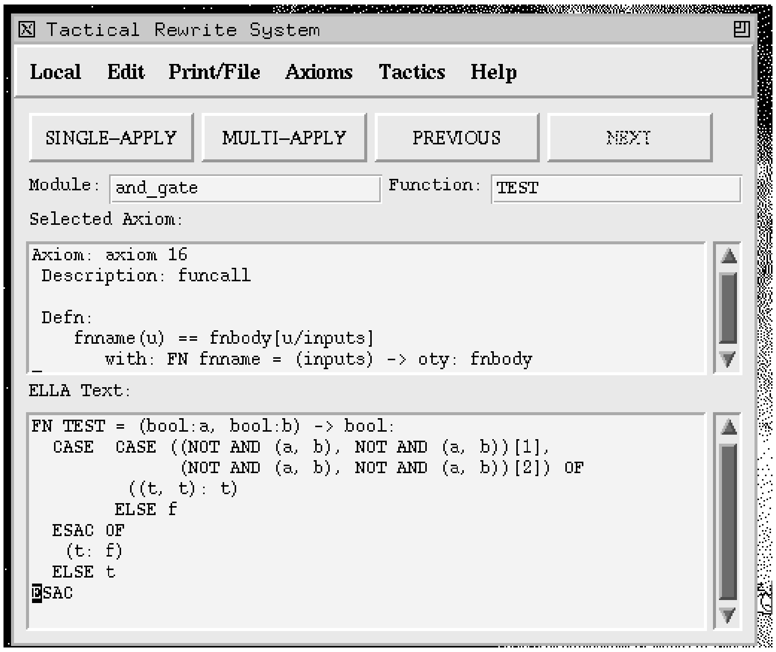

Figure 9: The ERT

Tactics . . . *–>Add...* menu item. A pop-up list of all axioms is presented; select one of these and it will be added to the list of already-selected axioms in a pop-up Current Tactic: pane. Press the MORE button to continue adding axioms to the tactic, or press the FINISH button when done. The resulting tactic is then presented for confirmation and/or hand-editing. After this, supply a name for the tactic when prompted — the tactic is then added to a list of available and already declared tactics. It can be applied by selecting the *Tactics* . . . *–>Select...* menu item and then pressing the SINGLE-APPLY or MULTI-APPLY button.

If you wish to save the rewritten ELLA-text, then select the *Print/File* . . . *–>Save File...* menu item and give a (Unix) filename when prompted. If you then want to include this text in the EVE database, then you must explicitly import it into a module, in the usual way. You can also load (Unix) files directly into the ERT, by using the *Print/File* . . . *–>Open File...* menu item.

5 Extensions

The current version of the EVE is an early prototype, and there are a number features which it would be desirable to add, apart from general improvements in efficiency and interfacing. The following collects together features that it will be desirable to add in future versions of the EVE.

The EVE provides a platform for applying various software tools to ELLA designs. It is envisaged that third-party tools will be included in future versions of the EVE. For example, a front-end transformation tool from DRA will be installed into the EVE to enable full ELLA to Core ELLA translation.

The context database will be re-implemented and based on a *Persistent Object Store* (POS). It will be multi-user, and handle versioning, as well as introducing the capabilities for storing relationships between modules. It will also be able to store ancillary items (for example, designer notes, formal analysis components)

In the SLA, facilities will be provided for a top-down design capability, so that high-level design stub modules can be instantiated before the lower level details of the modules are specified. The SLA will be linked with the schematics editor to provide animation facilities. Different methods of data input will be provided, together with results processing, such as pattern recognition for traces, with sampling and selective storage.

SymSim and the VCG will be re-implemented to take full advantage of term-sharing and lazy substitution facilities offered by the latest version of the GT generic term-manipulation system on which they are built. Strategies for VC generation and simplification will be enhanced, and will include facilities for partial VC generation. A closer correlation between VC term structure and the original ELLA-text will be maintained.

For state-based IOA verification, we intend to install state-of-the-art BDD-based equivalence checkers for sequential circuits. The SLA will be able to display any counter example traces generated.

Finally, the ERT will support an extensible axiom database, so that user-defined axioms can be included. The correctness of existing axioms will be verified.

A Further VCG Verification Examples

In this appendix we present two more examples of symbolic verification using the state evolution rule and the VCG.

Example (a): Linear Recursive Shift Register Load the file `<ella>/examples/pipe.elt` into the ELLA environment in a new context, calling the module `pipe`. The file contains two implementations of a two-unit boolean linear recursive shift register[9] (LRSR). These are initialised so that the sequence `...t t f t t f t t f...` is generated. One implementation contains two unit delays piped together, with their outputs fed into a mod2 adder — this feeds back into the first delay. The second implementation contains a pipe of three suitably initialised unit delays with a feedback from last to first. The purpose of this example is to illustrate how the VCG may be used to abstract functional behaviour, by selectively expanding functions, and representing the characteristics of those unexpanded functions with axioms at the proof stage.

Before compiling the designs select the *Set Function Wrap...* ...-> *Wrap (in Some)* option from the *Utilities* menu, and select the `pipe.PLUS` function from the function list presented. This will have the effect of abstracting the selected function during compilation, replacing its body with a function call in the final EPA process definition generated. When the VCG is applied to such a process it will not expand the function call, and will therefore not expand the control statements contained within the abstracted function — in general this will have the effect of significantly reducing the number of verification conditions generated, and in this case it results in the generation of a single VC. Of course, suitable PVS declarations will be required later in order to prove (or disprove) the resulting VCs.

Now compile the designs by pressing the EPA Compile button in the Analysis Tool — make sure the *EPA Compile* option is set in the *Options* menu. Select the functions `pipe.PIPE_1` and `pipe.PIPE_2` from the list appearing in the Compile EPA Objects: window, and then select the *Utilities* ...-> *VCG* menu item. The VCG window will appear with the function names `pipe.PIPE_1` and `pipe.PIPE_2` appearing in the Function 1 and Function 2 panes respectively. Start the VCG by pressing the GENERATE-VC button and enter '3' when prompted for the number of levels — the second design has a path length of three. The VCG will then take a short while to complete. Then select the *Utilities* ...-> *Reduce (heavy)* menu item to remove variable bindings from the single VCs generated.

The ELLA function name `PLUS` has been converted into the *EPA* function name `pipe_outp_plus` during compilation; In order to sufficiently characterise it, you will need the following PVS axioms:

```
pipe_ax1: AXIOM (FORALL (x: bool), (y: bool):
              (pipe_outp_plus(x,y) = pipe_outp_plus(y,x)))

pipe_ax2: AXIOM (FORALL (x: bool), (y: bool), (z: bool):
              (pipe_outp_plus(pipe_outp_plus(x,y),z) =
               pipe_outp_plus(x, pipe_outp_plus(y,z))))

pipe_ax3: AXIOM (FORALL (x: bool):
              (pipe_outp_plus(x,f) = x))

pipe_ax4: AXIOM (FORALL (x: bool):
              (pipe_outp_plus(x,x) = f))
```

[9]The linear recursive shift register example was provided by Peter Ryan (DRA, Malvern).

which equip pipe_outp_plus with the desired modular arithmetic properties. These can be found in the file <ella>/examples/pipe-ax.pvs — supply this to the VCG by selecting the *Load PVS File* item from the *Utilities* menu and then selecting the file <ella>/examples/pipe-ax.pvs. The filename pipe-ax.pvs will appear in the User PVS File pane. Convert the VC contexts into PVS format by selecting the *Utilities* ...->*Convert to PVS* menu item. This should result in a step VC similar to the following:

```
%%Step VCs:

epa_vcgs :  THEORY

BEGIN

%%% Global Vars:
   bool: DATATYPE
      BEGIN
         t : t?
         f : f?
      END  bool
      .
      .
   <variable declaration omitted>
      .
      .
pipe_ax1: AXIOM (FORALL (x: bool), (y: bool):
              (pipe_outp_plus(x,y) = pipe_outp_plus(y,x)))

pipe_ax2: AXIOM (FORALL (x: bool), (y: bool), (z: bool):
              (pipe_outp_plus(pipe_outp_plus(x,y),z) =
               pipe_outp_plus(x, pipe_outp_plus(y,z))))

pipe_ax3: AXIOM (FORALL (x: bool):
              (pipe_outp_plus(x,f) = x))

pipe_ax4: AXIOM (FORALL (x: bool):
              (pipe_outp_plus(x,x) = f))

%%% Verification Conditions:

   %%% Clauses :
   Clause_1: THEOREM

     %% Result:
     (pipe_outp_plus(pipe_outp_plus(pipe_outp_plus(pipe_outp_plus(stv_6, stv_7), stv_6),
                                    pipe_outp_plus(stv_6, stv_7)),
                     pipe_outp_plus(pipe_outp_plus(stv_6, stv_7), stv_6))
        = pipe_outp_plus(stv_6, stv_7))
     %% -------------------------------------------

END epa_vcgs
```

Output the VC to a file, using the *Display VCs* ...->*File VCs* menu item from the *Display* menu, and load the file into PVS.

Once in PVS, you need to use the associative/commutative properties of pipe_outp_plus to restructure the left-hand side of the conclusion, so that pipe_ax3 and pipe_ax4 can be used to reduce it to be identical to the right-hand side. There are of course several different ways of doing this; for example, apply the following the PVS commands:

Command	Description
`(skolem!)`	
`(auto-rewrite "pipe_ax3" "pipe_ax4")`	declares `pipe_ax3` and `pipe\_ax4` to be automatically applied rewrite rules
`(rewrite "pipe_ax1")`	applies the axiom `pipe_ax2` as a rewrite rule, if possible
`(rewrite "pipe_ax2")`	
`(lemma "pipe_ax2")`	instantiate axiom `pipe_ax2`
`(quant -1 ("stv_7!1" "stv_7!1" "stv_6!1"))`	
`(replace -1 (1) RL)`	apply `pipe_ax2` as a right-to-left replacement
`(do-rewrite)`	apply automatic rewrite rules specified above
`(rewrite "pipe_ax1")`	
`(rewrite "pipe_ax1")`	

Example (b): FIR Filter Invoke the VCG on the designs from files `<ella>/examples/fir-1.elt` and `<ella>/examples/fir-2.elt`. These are two implementations[10] of a simple 3-stage Finite Impulse Response (FIR) filter design, and will require 3 state evolution steps/levels in order to prove their equivalence, since the latency of the designs is 3 time-steps. There will only be a single VC generated in the Initial and Step VCs, because the designs do not contain any case statements.

First load the designs into the environment, using separate contexts, and modules `fir1` and `fir2`. Then compile the designs in the Analysis Tool and load the functions `fir1.FIR_1` and `fir2.FIR_2` into the VCG. Then press the GENERATE-VC button and enter '3' when prompted for the number of levels. The VCG will take several minutes to generate the VC; when it has finished, then reduce the size of the VC by selecting the *Reduce VCs (heavy)* menu item from the *Utilities* menu, and then convert the VC into PVS. You do not need any PVS axiom in this case, and so clear any User-PVS file using the *Clear PVS File* menu item on the *Utilities* menu.

Now display and output the step VC to a file, in the same way as above. You should obtain a PVS text similar to the following:

```
%%Step VCs:

epa_vcgs :  THEORY
   BEGIN
     .
     .
 <Type and Variable Declarations omitted>
     .
     .
              %%% Clauses :
               clause_1: THEOREM

              (

              %% Bindings:
```

[10] `fir-2.elt` is a *retimed* version of the basic `fir-1.elt` version. The original descriptions were provided by Nick Ray, GEC Plessey Semi-Conductors, Swindon.

```
            (arith_var_109 = proj_2(stv_16))
               AND (arith_var_108 = proj_2(stv_15))
               AND (arith_var_107 = proj_2(stv_24))
               AND (arith_var_106 = proj_2(stv_25))
               AND (arith_var_105 = proj_2(stv_14))
               AND (arith_var_104 = proj_2(stv_18))
               AND (arith_var_103 = proj_2(stv_17))
               AND (arith_var_102 = proj_2(inp_77_3))
               AND (arith_var_101 = proj_2(inp_77_5))
               AND (arith_var_100 = proj_2(inp_77))
               AND (arith_var_99 = proj_2(inp_77_1))

         AND
         %% General Conditions:

            ((d, ((arith_var_106 * 3) + ((arith_var_107 * 2) + (arith_var_100 * 1)))) =
                  (d, ((arith_var_100 * 1) + (arith_var_103 + arith_var_104))))
               AND ((d, ((arith_var_107 * 3) + ((arith_var_100 * 2) + (arith_var_99 * 1)))) =
                  (d, ((arith_var_99 * 1) + ((arith_var_100 * 2) + (arith_var_105 * 3)))))
               AND ((d, ((arith_var_100 * 3) + ((arith_var_99 * 2) + (arith_var_102 * 1)))) =
                  (d, ((arith_var_102 * 1) + ((arith_var_99 * 2) + (arith_var_100 * 3)))))

         %% Result:
         )
          IMPLIES

            ((d, ((arith_var_101 * 1) + ((arith_var_102 * 2) + (arith_var_99 * 3)))) =
                (d, ((arith_var_99 * 3) + ((arith_var_102 * 2) + (arith_var_101 * 1)))))
         %% ------------------------------------------

  END epa_vcgs

%%%------- End of Clause ------
```

You can prove the theorem by loading your PVS file into PVS, and using the same sequence of PVS commands listed in Example 1 from §4.3.2. Notice that the conclusion in the theorem is a tautology, and in order to be proven, it simply requires the application of the associative/commutative laws of integer addition, which are already part of the reasoning system in PVS.

B Source for ELLA Examples

This appendix contains a copy of the ELLA-text files used as examples in the tutorial. All of the files are also distributed with the EVE system.

B.1 pump.elt

```
TYPE pump = NEW (none | a | b | c | ab | bc | ca | abc).
TYPE level = NEW lv/(0..3).
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,
                ca : b,
                b : c,
                ab : c
                ELSE a
               ESAC,
        lv/2 : CASE store OF
                a : bc,
                ab : bc,
                b : ca,
                bc : ca
                ELSE ab
               ESAC
        ELSE abc
        ESAC.
   JOIN newstore -> store.
   OUTPUT newstore
END.
```

B.2 deland.elt

```
TYPE bool = NEW (f | t).
FN DEL = (bool) -> bool: IDELAY(f,1).

FN DELAND = (bool: x, bool: y) -> bool:
BEGIN
  MAKE DEL: del.
  LET inner = CASE y OF
                t : t
              ELSE f
              ESAC.
  LET and = CASE x OF
              t: inner
            ELSE f
            ESAC.
  JOIN and -> del.
  OUTPUT del
END.
```

B.3 twist_spec.elt

```
TYPE bool = NEW (f|t).
```

```
FN NEXT = (bool: reset, [3]bool: in) -> [3]bool:
CASE reset
  OF f : CASE in
           OF (f,f,f): (f,f,t),
              (f,f,t): (f,t,t),
              (f,t,t): (t,t,t),
              (t,t,t): (t,t,f),
              (t,t,f): (t,f,f),
              (t,f,f): (f,f,t),  # ERROR: last 'f' changed to 't' !!! #
              (f,t,f): (t,f,f),
              (t,f,t): (f,t,f)
         ELSE (f,f,f)
         ESAC,
     t : (f,f,f)
ELSE (f,f,f)
ESAC.

FN BOOL_DELAY = (bool) -> bool:
   IDELAY(f,1).

FN THREE_BOOL_DELAY = ([3]bool: in) -> [3]bool:
 ( BOOL_DELAY(in[1]),
   BOOL_DELAY(in[2]),
   BOOL_DELAY(in[3])
 ).

FN TWIST_SPEC = (bool: reset) -> [3]bool:
BEGIN
  MAKE THREE_BOOL_DELAY: tbd.
  MAKE NEXT: nxt.
  JOIN (nxt) -> tbd.
  JOIN (reset, tbd) -> nxt.
  OUTPUT tbd
END.
```

B.4 twist_impl.tex

```
TYPE bool = NEW (f|t).

FN OR = (bool: x, bool: y) -> bool:
CASE (x, y)
OF (f, f): f
ELSE t
ESAC.

FN NOT = (bool: x) -> bool:
CASE x
OF f: t
ELSE f
ESAC.

FN AND = (bool: x, bool: y) -> bool:
   NOT( OR(NOT(x), NOT(y))).

FN BOOL_DELAY = (bool) -> bool:
   # initial value f, delay 1 unit#
   IDELAY(f,1).

FN LATCH = (bool: i, bool: r) -> bool:
      BOOL_DELAY(AND(i, NOT(r))).

FN TWIST_IMPL = (bool: reset) -> [3]bool:
  BEGIN
     MAKE LATCH: a.
```

```
    MAKE LATCH: b.
    MAKE LATCH: c.
    JOIN (b, reset) -> a.
    JOIN (c, reset) -> b.
    JOIN (AND(NOT(a), OR(NOT(b), c)), reset) -> c.
    OUTPUT (a, b, c)
  END.
```

B.5 alu16_spec.elt

```
FN OR3 = ( bool: a, bool: b, bool: c ) -> bool:
CASE (a,b,c) OF
  (f,f,f): f
ELSE t
ESAC.

FN AND5 = (bool: a1, bool: a2, bool: a3, bool: a4, bool: a5) -> bool:
CASE (a1,a2,a3,a4,a5) OF
   (t,t,t,t,t): t
ELSE f
ESAC.

FN AND7 = (bool: a1, bool: a2, bool: a3, bool: a4, bool: a5, bool: a6,
           bool: a7) -> bool:
CASE (a1,a2,a3,a4,a5,a6,a7) OF
   (t,t,t,t,t,t,t): t
ELSE f
ESAC.

FN DO_ADD = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3) -> bool:
      AND5(m, NOT(e0), e1, e2, NOT(e3)).

FN DO_SUB = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3) -> bool:
      AND5(m, NOT(e0), e1, NOT(e2), e3).

FN DO_INCA = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3) -> bool:
      AND5(m, NOT(e0), e1, NOT(e2), NOT(e3)).

FN DO_INCB = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3) -> bool:
      AND5(m, NOT(e0), NOT(e1), e2, NOT(e3)).

FN DO_EOR = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3) -> bool:
      AND5(NOT(m), NOT(e0), e1, e2, NOT(e3)).

FN DO_AND = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3) -> bool:
      AND5(NOT(m), e0, NOT(e1), NOT(e2), NOT(e3)).

FN DO_OR = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3) -> bool:
      AND5(NOT(m), e0, e1, e2, NOT(e3)).

FN EOR = (bool: x, bool: y) -> bool:
        OR(AND(x, NOT(y)) , AND(NOT(x), y)).

FN SUM = (bool: x, bool: y, bool: z) -> bool:
        EOR(EOR(x, y), z).

FN CARRY = (bool: x, bool: y, bool: z) -> bool:
        OR3(AND(x, y), AND(x, z), AND(y, z)).

FN ADD_SLICE = (bool: a, bool: b, bool: c) -> (bool, bool):
        (SUM(a, b, c), CARRY(a, b, c)).


FN ADD4 = (bool: a0, bool: a1, bool: a2, bool: a3,
           bool: b0, bool: b1, bool: b2, bool: b3, bool: cin)
```

```
           -> (bool, bool, bool, bool, bool):
BEGIN
  LET r1 = ADD_SLICE(a0, b0, cin).
  LET r2 = ADD_SLICE(a1, b1, r1[2]).
  LET r3 = ADD_SLICE(a2, b2, r2[2]).
  LET r4 = ADD_SLICE(a3, b3, r3[2]).
  OUTPUT  (r1[1], r2[1], r3[1], r4[1], r4[2])
END.

FN ADD16 = (bool: a0, bool: a1, bool: a2, bool: a3, bool: a4,
            bool: a5, bool: a6, bool: a7, bool: a8, bool: a9,
            bool: a10, bool: a11, bool: a12, bool: a13, bool: a14, bool: a15,
            bool: b0, bool: b1, bool: b2, bool: b3, bool: b4,
            bool: b5, bool: b6, bool: b7, bool: b8, bool: b9,
            bool: b10, bool: b11, bool: b12, bool: b13, bool: b14, bool: b15,
            bool: cin)
            -> (bool, bool, bool, bool, bool, bool, bool, bool, bool,
                bool, bool, bool, bool, bool, bool, bool, bool):

BEGIN
  LET r1 = ADD4(a0, a1, a2, a3, b0, b1, b2, b3, cin).
  LET r2 = ADD4(a4, a5, a6, a7, b4, b5, b6, b7, r1[5]).
  LET r3 = ADD4(a8, a9, a10, a11, b8, b9, b10, b11, r2[5]).
  LET r4 = ADD4(a12, a13, a14, a15, b12, b13, b14, b15, r3[5]).
  OUTPUT  (r1[1], r1[2], r1[3], r1[4], r2[1], r2[2], r2[3], r2[4], r3[1], r3[2],
             r3[3], r3[4], r4[1], r4[2], r4[3], r4[4], r4[5])
END.


FN SUB16 = (bool: a0, bool: a1, bool: a2, bool: a3, bool: a4,
            bool: a5, bool: a6, bool: a7, bool: a8, bool: a9,
            bool: a10, bool: a11, bool: a12, bool: a13, bool: a14, bool: a15,
            bool: b0, bool: b1, bool: b2, bool: b3, bool: b4,
            bool: b5, bool: b6, bool: b7, bool: b8, bool: b9,
            bool: b10, bool: b11, bool: b12, bool: b13, bool: b14, bool: b15,
            bool: cin)
            -> (bool, bool, bool, bool, bool, bool, bool, bool, bool,
                bool, bool, bool, bool, bool, bool, bool, bool):

    ADD16(a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15,
          NOT(b0), NOT(b1), NOT(b2), NOT(b3), NOT(b4), NOT(b5),
          NOT(b6), NOT(b7), NOT(b8), NOT(b9), NOT(b10), NOT(b11),
          NOT(b12), NOT(b13), NOT(b14), NOT(b15), cin).

FN INCA16 = (bool: a0, bool: a1, bool: a2, bool: a3, bool: a4,
             bool: a5, bool: a6, bool: a7, bool: a8, bool: a9,
             bool: a10, bool: a11, bool: a12, bool: a13, bool: a14, bool: a15,
             bool: b0, bool: b1, bool: b2, bool: b3, bool: b4,
             bool: b5, bool: b6, bool: b7, bool: b8, bool: b9,
             bool: b10, bool: b11, bool: b12, bool: b13, bool: b14, bool: b15,
             bool: cin)
             -> (bool, bool, bool, bool, bool, bool, bool, bool, bool,
                 bool, bool, bool, bool, bool, bool, bool, bool):

   ADD16 (a0, a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15,
          f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, cin).


FN INCB16 = (bool: a0, bool: a1, bool: a2, bool: a3, bool: a4,
             bool: a5, bool: a6, bool: a7, bool: a8, bool: a9,
             bool: a10, bool: a11, bool: a12, bool: a13, bool: a14, bool: a15,
             bool: b0, bool: b1, bool: b2, bool: b3, bool: b4,
             bool: b5, bool: b6, bool: b7, bool: b8, bool: b9,
             bool: b10, bool: b11, bool: b12, bool: b13, bool: b14, bool: b15,
             bool: cin)
             -> (bool, bool, bool, bool, bool, bool, bool, bool, bool,
                 bool, bool, bool, bool, bool, bool, bool, bool):
```

```
        ADD16 (f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f,
          b0, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15,
          cin).


FN EOR16 = (bool: a0, bool: a1, bool: a2, bool: a3, bool: a4,
            bool: a5, bool: a6, bool: a7, bool: a8, bool: a9,
            bool: a10, bool: a11, bool: a12, bool: a13, bool: a14, bool: a15,
            bool: b0, bool: b1, bool: b2, bool: b3, bool: b4,
            bool: b5, bool: b6, bool: b7, bool: b8, bool: b9,
            bool: b10, bool: b11, bool: b12, bool: b13, bool: b14, bool: b15)
             -> (bool, bool, bool, bool, bool, bool, bool, bool,
                 bool, bool, bool, bool, bool, bool, bool, bool):

        (EOR(a0, b0), EOR(a1, b1), EOR(a2, b2), EOR(a3, b3),
         EOR(a4, b4), EOR(a5, b5), EOR(a6, b6), EOR(a7, b7),
         EOR(a8, b8), EOR(a9, b9), EOR(a10, b10), EOR(a11, b11),
         EOR(a12, b12), EOR(a13, b13), EOR(a14, b14), EOR(a15, b15)).


FN AND16 = (bool: a0, bool: a1, bool: a2, bool: a3, bool: a4,
            bool: a5, bool: a6, bool: a7, bool: a8, bool: a9,
            bool: a10, bool: a11, bool: a12, bool: a13, bool: a14, bool: a15,
            bool: b0, bool: b1, bool: b2, bool: b3, bool: b4,
            bool: b5, bool: b6, bool: b7, bool: b8, bool: b9,
            bool: b10, bool: b11, bool: b12, bool: b13, bool: b14, bool: b15)
             -> (bool, bool, bool, bool, bool, bool, bool, bool,
                 bool, bool, bool, bool, bool, bool, bool, bool):

        (AND(a0, b0), AND(a1, b1), AND(a2, b2), AND(a3, b3), AND(a4, b4),
         AND(a5, b5), AND(a6, b6), AND(a7, b7), AND(a8, b8), AND(a9, b9),
         AND(a10, b10), AND(a11, b11), AND(a12, b12), AND(a13, b13),
         AND(a14, b14), AND(a15, b15)).


FN  OR16 = (bool: a0, bool: a1, bool: a2, bool: a3, bool: a4,
            bool: a5, bool: a6, bool: a7, bool: a8, bool: a9,
            bool: a10, bool: a11, bool: a12, bool: a13, bool: a14, bool: a15,
            bool: b0, bool: b1, bool: b2, bool: b3, bool: b4,
            bool: b5, bool: b6, bool: b7, bool: b8, bool: b9,
            bool: b10, bool: b11, bool: b12, bool: b13, bool: b14, bool: b15)
             -> (bool, bool, bool, bool, bool, bool, bool, bool,
                 bool, bool, bool, bool, bool, bool, bool, bool):

        (OR(a0, b0), OR(a1, b1), OR(a2, b2), OR(a3, b3), OR(a4, b4),
         OR(a5, b5), OR(a6, b6), OR(a7, b7), OR(a8, b8), OR(a9, b9),
         OR(a10, b10), OR(a11, b11), OR(a12, b12), OR(a13, b13),
         OR(a14, b14), OR(a15, b15)).

FN ALU16 = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3,
            bool: cin, bool: a0, bool: b0, bool: a1, bool: b1,
            bool: a2, bool: b2, bool: a3, bool: b3, bool: a4, bool: b4,
            bool: a5, bool: b5, bool: a6, bool: b6, bool: a7, bool: b7,
            bool: a8, bool: b8, bool: a9, bool: b9, bool: a10, bool: b10,
            bool: a11, bool: b11, bool: a12, bool: b12, bool: a13, bool: b13,
            bool: a14, bool: b14, bool: a15, bool: b15)
            ->
            (bool, bool, bool, bool, bool, bool, bool, bool,
             bool, bool, bool, bool, bool, bool, bool, bool, bool):

BEGIN
        LET ra =
              ADD16 (a0, a1, a2, a3, a4, a5, a6, a7,
                     a8, a9, a10, a11, a12, a13, a14, a15,
                     b0, b1, b2, b3, b4, b5, b6, b7,
                     b8, b9, b10, b11, b12, b13, b14, b15,
```

```
                    cin).
        LET rs =
              SUB16 (a0, a1, a2, a3, a4, a5, a6, a7,
                     a8, a9, a10, a11, a12, a13, a14, a15,
                     b0, b1, b2, b3, b4, b5, b6, b7,
                     b8, b9, b10, b11, b12, b13, b14, b15,
                     cin).
        LET ria =
              INCA16 (a0, a1, a2, a3, a4, a5, a6, a7,
                      a8, a9, a10, a11, a12, a13, a14, a15,
                      b0, b1, b2, b3, b4, b5, b6, b7,
                      b8, b9, b10, b11, b12, b13, b14, b15,
                      cin).
        LET rib =
              INCB16 (a0, a1, a2, a3, a4, a5, a6, a7,
                      a8, a9, a10, a11, a12, a13, a14, a15,
                      b0, b1, b2, b3, b4, b5, b6, b7,
                      b8, b9, b10, b11, b12, b13, b14, b15,
                      cin).
        LET reor =
              EOR16 (a0, a1, a2, a3, a4, a5, a6, a7,
                     a8, a9, a10, a11, a12, a13, a14, a15,
                     b0, b1, b2, b3, b4, b5, b6, b7,
                     b8, b9, b10, b11, b12, b13, b14, b15).
        LET rand =
              AND16 (a0, a1, a2, a3, a4, a5, a6, a7,
                     a8, a9, a10, a11, a12, a13, a14, a15,
                     b0, b1, b2, b3, b4, b5, b6, b7,
                     b8, b9, b10, b11, b12, b13, b14, b15).
        LET ror =
              OR16 (a0, a1, a2, a3, a4, a5, a6, a7,
                    a8, a9, a10, a11, a12, a13, a14, a15,
                    b0, b1, b2, b3, b4, b5, b6, b7,
                    b8, b9, b10, b11, b12, b13, b14, b15).
 OUTPUT CASE (m, e0, e1, e2, e3) OF
           (t, f, t, t, f) :
             (ra[1], ra[2], ra[3], ra[4], ra[5], ra[6], ra[7], ra[8], ra[9],
              ra[10], ra[11], ra[12], ra[13], ra[14], ra[15], ra[16], ra[17]),
           (t, f, t, f, t) :
             (rs[1], rs[2], rs[3], rs[4], rs[5], rs[6], rs[7], rs[8], rs[9],
              rs[10], rs[11], rs[12], rs[13], rs[14], rs[15], rs[16], rs[17]),
           (t, f, t, f, f) :
             (ria[1], ria[2], ria[3], ria[4], ria[5], ria[6], ria[7],
              ria[8], ria[9], ria[10], ria[11], ria[12], ria[13], ria[14],
              ria[15], ria[16], ria[17]),
           (t, f, f, t, f) :
             (rib[1], rib[2], rib[3], rib[4], rib[5], rib[6], rib[7],
              rib[8], rib[9], rib[10], rib[11], rib[12], rib[13], rib[14],
              rib[15], rib[16], rib[17]),
           (f, f, t, t, f) :
             (reor[1], reor[2], reor[3], reor[4], reor[5], reor[6], reor[7],
              reor[8], reor[9], reor[10], reor[11], reor[12], reor[13],
              reor[14], reor[15], reor[16], f),
           (f, t, f, f, f) :
             (rand[1], rand[2], rand[3], rand[4], rand[5], rand[6], rand[7],
              rand[8], rand[9], rand[10], rand[11], rand[12], rand[13],
              rand[14], rand[15], rand[16], f),
           (f, t, t, t, f) :
             (ror[1], ror[2], ror[3], ror[4], ror[5], ror[6], ror[7],
              ror[8], ror[9], ror[10], ror[11], ror[12], ror[13],
              ror[14], ror[15], ror[16], f)

           ELSE (f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f, f)
                         # This value is thrown away by the#
                         # don't-care set#
        ESAC
END.
```

```
FN GOOD_INPUTS = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3)
               -> [5]bool:
   CASE (m,e0,e1,e2,e3) OF
       (t,f,t,t,f) : (m,e0,e1,e2,e3),
       (t,f,t,f,t) : (m,e0,e1,e2,e3),
       (t,f,t,f,f) : (m,e0,e1,e2,e3),
       (t,f,f,t,f) : (m,e0,e1,e2,e3),
       (f,f,t,t,f) : (m,e0,e1,e2,e3),
       (f,t,f,f,f) : (m,e0,e1,e2,e3),
       (f,t,t,t,f) : (m,e0,e1,e2,e3)
   ELSE (f,f,f,f,f)
   ESAC.


FN ALU16_SPEC_HARNESS = (bool: m, bool:  e0, bool:  e1, bool:  e2,
                         bool:  e3, bool:  cin, bool:  a0, bool:  b0,
                         bool:  a1, bool:  b1, bool:  a2, bool:  b2,
                         bool:  a3, bool:  b3, bool:  a4, bool:  b4,
                         bool:  a5, bool:  b5, bool:  a6, bool:  b6,
                         bool:  a7, bool:  b7, bool:  a8, bool:  b8,
                         bool:  a9, bool:  b9, bool:  a10, bool:  b10,
                         bool:  a11, bool:  b11, bool:  a12, bool:  b12,
                         bool:  a13, bool:  b13, bool:  a14, bool:  b14,
                         bool:  a15, bool:  b15)
                         -> [17]bool:
BEGIN
   MAKE GOOD_INPUTS:select.
   MAKE ALU16: alu.
   JOIN (m,e0,e1,e2,e3) -> select.
   JOIN (select[1],select[2],select[3],select[4],select[5],
         cin,a0, b0, a1, b1, a2, b2, a3, b3,
         a4, b4, a5, b5, a6, b6, a7, b7,
         a8, b8, a9, b9, a10, b10, a11, b11,
         a12, b12, a13, b13, a14, b14, a15, b15) -> alu.
   OUTPUT alu
END.
```

B.6 alu16_impl.elt

```
FN AND3 = ( bool: a, bool: b, bool: c ) -> bool:
CASE (a,b,c) OF
  (t,t,t): t
ELSE f
ESAC.

FN EOR = (bool: x, bool: y) -> bool:
        OR(AND(x, NOT(y)) , AND(NOT(x), y)).

FN ALU1 = (bool: a, bool: b, bool: c, bool: m, bool: e0,
           bool: e1, bool: e2, bool: e3)
           -> (bool,bool):
BEGIN
        LET s1 = AND(a, e1).
        LET s2 = NOT(b).
        LET s3 = NOT(AND(s2, e3)).
        LET s4 = NOT(AND(e2, b)).
        LET s5 = AND(c, m).
        LET s6 = NOT(AND3(a, b, e0)).
        LET s7 = NOT(AND(s3, s4)).
        LET s8 = EOR(s1, s7).
        LET s9 = NOT(AND(s1, s7)).
        LET s10 = NOT(AND(s8, s5)).
        LET s11 = NOT( EOR(s8, s5)).
        OUTPUT (NOT(AND(s11, s6)), NOT(AND(s9, s10)))
```

```
END.

FN ALU4 = (bool: a0, bool: a1, bool: a2, bool: a3,
           bool: b0, bool: b1, bool: b2, bool: b3, bool: cin,
           bool: m, bool: e0, bool: e1, bool: e2, bool: e3)
           -> (bool, bool, bool, bool, bool):
BEGIN
        LET r1 = ALU1(a0, b0, cin, m, e0, e1, e2, e3).
        LET r2 = ALU1(a1, b1, r1[2], m, e0, e1, e2, e3).
        LET r3 = ALU1(a2, b2, r2[2], m, e0, e1, e2, e3).
        LET r4 = ALU1(a3, b3, r3[2], m, e0, e1, e2, e3).
        OUTPUT (r1[1], r2[1], r3[1], r4[1], r4[2])
END.

FN ALU16 = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3,
            bool: cin, bool: a0, bool: b0, bool: a1, bool: b1,
            bool: a2, bool: b2, bool: a3, bool: b3, bool: a4, bool: b4,
            bool: a5, bool: b5, bool: a6, bool: b6, bool: a7, bool: b7,
            bool: a8, bool: b8, bool: a9, bool: b9, bool: a10, bool: b10,
            bool: a11, bool: b11, bool: a12, bool: b12, bool: a13, bool: b13,
            bool: a14, bool: b14, bool: a15, bool: b15)
            ->
            (bool, bool, bool, bool, bool, bool, bool, bool,
             bool, bool, bool, bool, bool, bool, bool, bool, bool):

BEGIN
    LET r1 = ALU4 (a0, a1, a2, a3,
                   b0, b1, b2, b3, cin,
                   m, e0, e1, e2, e3).
    LET r2 = ALU4 (a4, a5, a6, a7,
                   b4, b5, b6, b7, r1[5],
                   m, e0, e1, e2, e3).
    LET r3 = ALU4 (a8, a9, a10, a11,
                   b8, b9, b10, b11, r2[5],
                   m, e0, e1, e2, e3).
    LET r4 = ALU4 (a12, a13, a14, a15,
                   b12, b13, b14, b15, r3[5],
                   m, e0, e1, e2, e3).
    OUTPUT  (r1[1], r1[2], r1[3], r1[4], r2[1], r2[2], r2[3], r2[4],
             r3[1], r3[2], r3[3], r3[4], r4[1], r4[2], r4[3], r4[4],
             AND(r4[5], m))
END.

FN GOOD_INPUTS = (bool: m, bool: e0, bool: e1, bool: e2, bool: e3)
               -> [5]bool:
   CASE (m,e0,e1,e2,e3) OF
       (t,f,t,t,f) : (m,e0,e1,e2,e3),  #ADD#
       (t,f,t,f,t) : (m,e0,e1,e2,e3),  #SUB#
       (t,f,t,f,f) : (m,e0,e1,e2,e3),  #INCA#
       (t,f,f,t,f) : (m,e0,e1,e2,e3),  #INCB#
       (f,f,t,t,f) : (m,e0,e1,e2,e3),  #EXOR#
       (f,t,f,f,f) : (m,e0,e1,e2,e3),  #AND#
       (f,t,t,t,f) : (m,e0,e1,e2,e3)   #OR#
   ELSE (f,f,f,f,f)
   ESAC.


FN ALU16_IMPL_HARNESS = (bool: m, bool:  e0, bool:  e1, bool:  e2,
                         bool:  e3, bool:  cin, bool:  a0, bool:  b0,
                         bool:  a1, bool:  b1, bool:  a2, bool:  b2,
                         bool:  a3, bool:  b3, bool:  a4, bool:  b4,
                         bool:  a5, bool:  b5, bool:  a6, bool:  b6,
                         bool:  a7, bool:  b7, bool:  a8, bool:  b8,
                         bool:  a9, bool:  b9, bool:  a10, bool:  b10,
                         bool:  a11, bool:  b11, bool:  a12, bool:  b12,
                         bool:  a13, bool:  b13, bool:  a14, bool:  b14,
                         bool:  a15, bool:  b15)
```

```
                         -> [17]bool:
BEGIN
   MAKE GOOD_INPUTS:select.
   MAKE ALU16: alu.
   JOIN (m,e0,e1,e2,e3) -> select.
   JOIN (select[1],select[2],select[3],select[4],select[5],
         cin,a0, b0, a1, b1, a2, b2, a3, b3,
         a4, b4, a5, b5, a6, b6, a7, b7,
         a8, b8, a9, b9, a10, b10, a11, b11,
         a12, b12, a13, b13, a14, b14, a15, b15) -> alu.
   OUTPUT alu
END.
```

B.7 deland2.elt

```
TYPE bool = NEW (f | t).
FN DEL = (bool) -> bool: IDELAY(f,1).

FN AND = (bool: a, bool: b)  -> bool:
   CASE (a,b) OF
     (t,t) : t
   ELSE f
   ESAC.


FN DELAND2 = (bool: x, bool: y) -> bool:
   DEL(AND(x,y)).
```

B.8 simple-latches.elt

```
TYPE control = NEW ( ok | read ).
TYPE data = NEW i/(0..2000).

FN DEL = (data) -> data:
  IDELAY(i/0,1).

FN SUM = (data: x, data: y) -> data:
ARITH x+y.

FN MOD1 = (data: x, data: y) -> data:
ARITH x MOD y.

FN MAXB = () -> data: i/1000.

FN DIV = (data: x, data: y) -> data:
ARITH x % y.

FN MULT = (data: x, data: y) -> data:
ARITH x * y.

FN F = (data: in, control: cntrl) -> data:
BEGIN
   MAKE DEL: m.
   LET newm = CASE cntrl OF
                read : in
              ELSE m
              ESAC.
   JOIN newm -> m.
   OUTPUT CASE cntrl OF
            ok : SUM(in,m)
          ELSE in
          ESAC
END.
```

```
FN G = (data: in, control: cntrl) -> data:
BEGIN
   MAKE DEL: low.
   MAKE DEL: high.
   LET newlow = CASE cntrl OF
                  read : MOD1(in,MAXB())
                ELSE low
                ESAC.
   LET newhigh = CASE cntrl OF
                   read : MULT(MAXB(), DIV(in,MAXB()))
                 ELSE high
                 ESAC.
   JOIN newlow -> low.
   JOIN newhigh -> high.
   OUTPUT CASE cntrl OF
            ok : SUM(in, SUM(low, high))
          ELSE in
          ESAC
END.
```

B.9 div-axiom.pvs

```
  div_ax: AXIOM (FORALL (x: int), (y: int):
                  ((y /= 0) IMPLIES (y * div(x, y) + mod(x, y) = x)))
```

B.10 and.elt

```
TYPE bool = NEW (f | t).

FN NOT = (bool: x) -> bool:
   CASE x OF
     t : f
   ELSE t
   ESAC.

FN AND = (bool: a, bool: b)  -> bool:
   CASE (a,b) OF
     (t,t) : t
   ELSE f
   ESAC.

FN TEST = (bool: a, bool: b)  -> bool:
  NOT( AND(NOT(AND(a,b)),NOT(AND(a,b)))).
```

B.11 div-axiom.pvs

```
TYPE bool = NEW (f | t).

FN DEL_F = (bool)->bool: IDELAY(f,1).
FN DEL_T = (bool)->bool: IDELAY(t,1).

FN PLUS = (bool:x, bool:y) -> bool:
CASE (x,y) OF
  (t,f) : t,
  (f,t) : t
  ELSE f
ESAC.

FN PIPE_1 = () -> bool:
BEGIN
```

```
   MAKE DEL_F: a.
   MAKE DEL_T: b.
   MAKE PLUS: plus.
   JOIN a -> b.
   JOIN (a,b) -> plus.
   JOIN plus -> a.
  OUTPUT plus
END.


FN PIPE_2 = () -> bool:
   BEGIN
      MAKE DEL_F: a.
      MAKE DEL_T: b.
      MAKE DEL_T: plus.
      JOIN a -> b.
      JOIN b -> plus.
      JOIN plus -> a.
     OUTPUT plus
END.
```

B.12 pipe-ax.pvs

```
pipe_ax1: AXIOM (FORALL (x: bool), (y: bool):
              (pipe_outp_plus(x,y) = pipe_outp_plus(y,x)))

pipe_ax2: AXIOM (FORALL (x: bool), (y: bool), (z: bool):
              (pipe_outp_plus(pipe_outp_plus(x,y),z) =
               pipe_outp_plus(x, pipe_outp_plus(y,z))))

pipe_ax3: AXIOM (FORALL (x: bool):
              (pipe_outp_plus(x,f) = x))

pipe_ax4: AXIOM (FORALL (x: bool):
              (pipe_outp_plus(x,x) = f))
```

C A design methodology

The purpose of this appendix is to indicate how the verification tools developed on the IED project "Formal Verification Support for ELLA" may be used in practice. We discuss the basic approaches that are supported by the tools, their pros and cons in terms of design characteristics, and the basic model that should be used to achieve results in an efficient way.

C.1 Introduction to System Verification

This section sets out some of the basic modus operandi and techniques that a hardware designer needs when using the system to "verify" designs.

The whole idea of verification involves the comparison of one system description against another. Often one of these descriptions is called a *specification* and the other an *implementation*. The important point is that each of them describes semantic objects that can be compared against each other. The process of verification is then to calculate (in some sense) the behaviours of systems described and to then *compare* them against each other, in some suitable fashion.

In the current version of our ELLA verification system, this notion of specification and implementation is purely one of convenience and convention to the designer — they are *both* simply (Core) ELLA descriptions of the (mathematical) behaviour of hardware devices. From a theoretical point of view, both can be as "Abstract" or as "Concrete" as the other. Because of the symmetrical role of these designs, the comparison between designs is for (semantic) *equivalence* which we denote by "==".

As you may well appreciate, the verification of system designs is a hard and complex task for which there may be many different ways of achieving similar goals, many of which are still to be invented or discovered. To make progress with research in this area, it is necessary to make decisions regarding what will and what will not be supported.

Our prototype verification system provides the designer with essentially two ways of approaching the verification task:

IOA approach: Complete enumeration, followed by comparison of semantic objects

VCG approach: Generation of a series of logic formulae representing this comparison of semantic behaviour, which then have to be logically proven - by some external theorem-proving system.

In our case, this system is the PVS system from SRI. We chose it because the system provides reasonable support for decision procedures and rewriting, so allowing some automated theorem proving support.

We feel that it should be reasonably easy to add support for other external theorem-proving systems.

At the start of the ELLA verification project, we took soundings concerning verification support from designers at a number of leading UK semiconductor manufacturers (including GEC-Plessey Semiconductors). From these soundings it became clear that the most important requirement was to be able to check behavioural equivalence of designs. It was also clear from these discussions that any theorem-proving technology provided should be as automatic as possible to assist adoption of the overall techniques amongst hardware designers.

So a major part of the rationale for researching the IOA and VCG approaches mentioned above is that the resulting techniques held out considerable promise of being fairly automatic, so that the hardware designer has almost none of the "proof engineering" burden that would otherwise be needed when using general automated theorem-proving systems. Such "proof engineering" is often not related to the content of the original hardware design, but is usually associated with persuading theorem-proving systems to accept what the design, the specification and its proofs are in its own terms.

What follows below are some criteria for assessing which approach to use. We first discuss the "basic model" which gives the basic process that designers should use to verify their designs using this system. This is followed by discussions of the pros and cons of using the IOA and VCG approaches.

C.2 Basic model for system verification

This section describes a pragmatic approach for designers to take in order to use this system as successfully as possible.

This approach is essentially one of taking small incremental design steps from a single starting point until a satisfactory design is reached that is equivalent in functionality but perhaps much more efficient in terms of resources. The system is used to verify all the intermediate design steps, ensuring that functionality is preserved at each stage.

All designs must be eventually expressed in Core ELLA to use the system's verification tools. We discuss use of alternative design notations in §C.7 below.

The basic model is as follows:

- We shall assume that the designer has a *single* starting point - the first design, written in Core ELLA. Call this first system $D1$.
- The designer then makes some (small) changes to this description, call this description $D2$.
- The designer then wishes to use the system to compare these systems – to determine if $D1 == D2$.
- If this holds, the designer then continues to develop description $D3$ starting from $D2$. The designer then tests if $D2 == D3$ and so on.

The important point is that if we know $D1 == D2$ and that $D2 == D3$, then we know that $D1 == D3$ — even though we did not have to test it explicitly!

This allows the designer to make small incremental steps in the design, each step of which is verified:

$$D1 == D2 == D3 == \ldots == Dn$$

By making the steps between the designs small enough, the systems should be sufficiently similar for structural correspondence to cancel out a lot of the verification overhead.

C.3 Advanced models of system verification

The basic model given above assumes that the designer has a single design to start with and that the designer simply wants to check that all further design steps maintain semantic equivalence.

We now extend this approach a little to capture more complex ways of using the verification tools.

C.4 Showing equivalence of two arbitrary designs

Suppose that the designer has *two* given designs that he wishes to know are equivalent. We can now adapt the basic model given above to this situation.

Let A and B be the two given designs. We shall show that they are equivalent if we can construct a sequence of design steps that links A to B; that is, construct a sequence of designs $D1, D2, \ldots, Dk$ such that:

$$A == D1 == D2 == \ldots == Dk == B$$

Of course, we can choose each design step to be as large or as small as necessary - as long as the tools can positively check the equivalence between each step.

A standard approach would be to develop a design chain like the above from one of the given designs - A or B and continue until the current design is shown to be positively equivalent to the other end. A more general approach would be to start from *both* ends and try to “meet in the middle” i.e eventually show that the two current end-points are equivalent. Note that the first approach is just a simpler version of the second “meet in the middle” approach, where we simply don’t develop one of the given end-points and just keep it fixed.

Clearly this can be extended to show that a collection of designs:

$$A1, A2, A3, \ldots, An$$

to be equivalent, in the obvious way.

C.5 Pros and Cons: the IOA Approach

In this approach, we take the two designs to be compared, $D1$ and $D2$, and compute a semantic object that represents the semantic behaviour of each design. This is done by an essentially enumerative technique.

Once we have the two semantic objects, we can compare the behaviours that they represent and see if there is any mismatch. If there is no mismatch of behaviour, then the original designs are semantically equivalent. On the other hand, if mismatches are found, then the original designs are not equivalent. It turns out that in this situation, the tools are able to determine an input trace which drives each system to the point at which their behaviour diverges.

C.5.1 Limitations and Envelope of Operation

The IOA approach does *not* depend very strongly upon structural similarity between the two given designs - especially if the systems do not have much state. The reason for this is that the semantic objects are “flattened” and are “canonical”.

What is important is that the state machines produced by compilation need to be as similar in structure as possible – in design terms this means that the designs being compared in this way should make use of "memory" elements like latches and delays in similar ways.

One of the great strengths and benefits of our verification approaches is that the pattern of "memory use" in each design does not have to be identical. However, making them closer will generally make the verification simpler and easier to ascertain.

The other limiting factors are the size of the data types involved - each variable used in the top-level design will contribute extra complexity in a multiplicative way to the overall compilation and the verification of the system.

For example, suppose that we have[11]

- type A of size 4
- type B of size 2
- type C of size 3

and we have a system S with:

- 3 input parameters of type A
- 2 state variables of type B and
- 2 outputs of type C

Then the overall dimension of this system is the number of variables $3+2+2 = 7$, and the overall size is $(4^3)*(2^2)*(3^2) = 64*4*9 = 2304$. Since the system is *deterministic* (i.e. output and next state are determined by the input and the current state), the transition table will have $(State * Input = 256)$ entries. Note that the number of entries is independent of the size of outputs - all that matters is the information needed to specify the current configuration of the system.

It turns out that it is possible for semantic objects with very different transition topology (i.e. shape of table) to actually be equivalent despite their superficial differences. So, in the worst case, the comparison algorithm will need to match every transition of one system against every transition in the other. This tends to happen when comparing state machines with wildly different transition topologies. Accordingly, the worst case is quadratic in the sizes of the transition tables involved.

However, in the case that the transition tables are similar in structure, then the comparison algorithm is more efficient and this is almost linear in the sizes of the transition table entries.

This leads to an important observation: comparing inequivalent designs generally requires more resources to produce a result than the comparison of equivalent designs will. The reason for this is that inequivalent designs are quite *unlikely* to compile down to transition systems of similar topology and this generally produces behaviour closer to the worst case.

[11] In actual fact the IOA approach is presently restricted to handling Boolean types (i.e. size 2). But the above discussion is still correct and relevant.

C.5.2 Guidelines for selecting/rejecting IOA approach

Let the systems be $D1$ and $D2$ with compiled semantic objects IOA1 and IOA2 respectively.

Selection

For each system, the size of the generated state machines should be "small", say, less than 2^{16}. The size of a state machine is essentially the product of the sizes of the transition table and the output types.

This means that the IOA approach is only applicable to systems involving relatively small numbers of variables ranging over fairly simple data types (i.e. Booleans).

If both designs are essentially stateless (i.e. pure combinational logic) then the functional BDD checker can be used. This tool can essentially take much larger designs – perhaps as large as 2^{185}. However, the verification of such systems is still rather subtle - if the output values are dependent upon a large number of input values then the BDD representation may not be compact enough for verification purposes.

Rejection

The data types used in either of the designs includes non-Boolean types. The present IOA analysis tools are restricted to using Boolean (i.e. two-valued) types.

Remedy: try reformulating the designs just using Boolean types. The generated transition tables have widely differing structure. This is difficult to make quantitative and precise, but essentially if the variables of each design are difficult to associate with each other, then the designs are probably too far apart to analyse using the tools.

Remedy: try extending the design chain to make some extra design steps between the two designs.

C.6 Pros and Cons: the VCG Approach

In the VCG approach, the basic idea is to avoid the computationally expensive calculation of a canonical semantic object by generating logic formulae that characterise the semantic comparison of the behaviour of two given designs. The method used for this is called the State Evolution technique (a form of automated symbolic execution). The mathematical basis for this is described elsewhere in the documentation.

So, given two designs, $D1$ and $D2$, the VCG is applied to their definitions. We also need to give a number, Step, which says how many steps we should expand the behavioural comparison. What is then produced as output is (hopefully) a small set of logical clauses which, if these can be shown to be valid, mean that the two designs are equivalent. It is possible that the set of clauses may indeed be empty — this means that there are no obligations to be proven and that therefore the designs are equivalent.

So, if all the clauses produced can be established as being valid, then this means that the designs are equivalent. However, it may turn out that at least one clause is unprovable and cannot be shown to be valid. In this situation, we need to be careful to distinguish why a clause is unprovable - this analysis will tell us if the designs are actually inequivalent or if we simply need to increase the value of Step.

We need to check if the unprovable clause is:

Satisfiable (but also falsifiable) : - This means that the designs could still be equivalent – their logical status is contingent.

This satisfiability can be witnessed by finding inputs and states which make the formula hold. Usually these will be reasonably easy to find.

Unsatisfiable - This means that the designs are inequivalent - as we have reached a point at which each design specifies distinct behaviours.

C.6.1 Limitations and Envelope of Operation

Unlike the IOA approach, the VCG approach critically exploits the similarity of structure within definitions in order to generate compact verification clauses. This method utilises the algebraic structure of known operators in order to simplify the clauses that are generated. The VCG attempts to:

a) Minimise the number of clauses generated by avoiding case analysis as much as possible.

b) Use rewriting and equational reasoning about operators and predicates to simplify and in many cases eliminate the clauses after they have been generated.

In other words, this method is mostly applicable when the designs are structurally similar and when the operators used to express the design have well-developed and extensive libraries of properties.

The lazy expansion strategy underlying the VCG involves characterising the output response and the next-state functions in terms of the input and the current state. This information is generated for all operators defined in each design. The VCG should also be able to exploit any sharing of definitions between the designs, so avoiding the generation of clauses for obviously equivalent sub-systems.

Finally, we should point out that since the VCG approach is applicable to infinitary systems containing unbounded data types such as the integers, it is possible to encode the behaviour of an arbitrary Turing machine into the definitions. This points to the undecidability of the method underlying the VCG approach.

In particular, it is possible to find two designs which are actually equivalent but for which no finite Step value can be found to show their equivalence using this technique. This means that for some pairs of designs, it would in effect be necessary to "unroll" the comparison tree infinitely.

Fortunately, this can generally be circumvented by modifying both designs to expose more of their state variables for comparison purposes (c.f. ghost variables). Indeed this technique of adding more ways of observing the state usually reduces the number of Steps needed to obtain a definite result.

C.6.2 Guidelines for selecting/rejecting VCG approach

The criteria below are given purely in terms of rejection - in other words, the VCG approach may be applicable if none of the rejection criteria holds.

Rejection

If there is little similarity of structure between the two designs then the VCG will produce either:

- too many clauses - essentially an enumeration of all intermediate operators(!) This could be far worse than a pure enumeration since the internal structure will also have been expanded.

  Remedy: Try "Wrapping" some of the functions, so preventing their expansion inline.

- too few clauses - essentially the original goal is returned in clause form, with no expansion at all. This happens if there are no definitions that can be further expended.

  Remedy: Try "Unwrapping" some operators if that is possible. Alternatively, see if this clause can be established using theorem proving techniques (i.e. treat it as an "atomic" indivisible clause).

You will have no doubt noticed that the above Remedies given above are in conflict with each other. Unfortunately, there seems to be no "golden rule" which guarantees that the clauses produced will be *sufficient* to establish the result one way or the other.

If the designs are both "flat" and have no algebraic operator structure, then the VCG will have little option but to attempt to "unwind" the behaviour of each design to the depth specified by the Step parameter.

This means that the VCG approach can do no better than enumeration - and may in fact do a great deal worse, since any explicit case analysis will be expanded for each function call. We thus run the risk of generating too many clauses for similar reasons to those mentioned earlier.

Remedy: Reformulate the designs to contain some algebraic structure which is, as far as possible, shared by both designs.

C.7 Using alternative design notations

Not every designer uses ELLA - they may have another in-house design notation that is in widespread use and enjoys widespread investment in tools, training and expertise.

Assume that we have a design notation (called here HDL) and a semantics-preserving translation from HDL into (Core) ELLA. This is conceptually a functional mapping:

$$HDLtoELLA : HDL \longrightarrow ELLA$$

For our purposes, it is useful to ensure that this relationship between HDL and ELLA is tight enough to allow verification results to be 'pulled across' from ELLA back to the original designs expressed in HDL. To do this, we define the *induced equivalence* on HDL:

	let $N1$ and $N2$ be any designs in HDL,
and	let $D1$ and $D2$ be designs in ELLA such that:

	$D1 = HDLtoELLA(N1)$
and	$D2 = HDLtoELLA(N2)$

i.e.	ELLA design $D1$ is the translation of HDL design $N1$ and
	ELLA design $D2$ is the translation of HDL design $N2$.

We now define the *induced equivalence* on HDL designs as follows:

$$(N1 == N2) \text{ Iff } (D1 == D2)$$

i.e. Two HDL designs, $N1$ and $N2$, are semantically equivalent if and only if their translations into ELLA, $D1$ and $D2$, are.

(Caveat: We should also check that this is compatible with the existing semantics on HDL i.e. behaviour is mapped correctly from each notation to the other. However, it is likely that most sensible choices of mapping HDLtoELLA will satisfy this point.)

The important fact about this is that if $D1$ were NOT equivalent to $D2$ then our original designs, $N1$ and $N2$, could not have been equivalent either. In other words the translation preserves both equivalence AND inequivalence. This allows us to reason about our original designs by analysing their translations into ELLA.

C.7.1 Basic model of system verification

The basic model may be carried over to this situation using the above ideas. We assume that our designer is working in HDL and then translating the designs to ELLA. In other words the designer is working according to the following (modified) scheme:

- As before, assume that the designer has a *single* starting point - the first design, written in HDL. Call this first system $N1$.
- Now translate $N1$ into a corresponding ELLA design, $D1$ say, by using the translation mapping HDLtoELLA.
- The designer then makes some (small) changes to the HDL description, $N1$. Now call this new description $N2$.
- Now translate $N2$ into a corresponding ELLA design, $D2$ say, by using the translation mapping HDLtoELLA.
- The designer then wishes to use the tools to compare these HDL designs - to determine if $N1 == N2$. We may do this by making a suitable comparison on the corresponding ELLA designs, $D1$ and $D2$.
- If this holds, the designer then continues to develop HDL description $N3$ starting from $N2$. The designer then tests if $N2 == N3$ and so on in the same fashion as above, by translating to ELLA.

Figure 10 neatly summarises this relationship.

C.8 Further constraints upon HDLtoELLA

It is probably wise to try to ensure that the translation HDLtoELLA is, in some sense, structure preserving. This helps to ensure that if designs $N1$ and $N2$ only differ slightly in a number of places then so too do the corresponding translations $D1$ and $D2$. This property of "preserving small changes" will also make the verification process more predictable and understandable.

$$\begin{tikzcd}
\text{HDL:} \arrow[d, phantom, "\mathit{HDLtoELLA}"] & N1 \arrow[d] & == & N2 \arrow[d] & == & N3 \arrow[d] & == & \ldots & == & Nk \arrow[d] \\
\text{ELLA:} & D1 & == & D2 & == & D3 & == & \ldots & == & Dk
\end{tikzcd}$$

Figure 10: Relationship Between Design Descriptions

D Background to the symbolic verification approach

This section gives a brief introduction to the *state evolution* method for establishing the equivalence of two design using a symbolic verification approach. Please refer to [BGMW94b] for a detailed account of the method.

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 4.1 : (Strong) Bisimilarity

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

- S is symmetric.
- $\langle P, Q \rangle \in S \;\Rightarrow$
  $\forall\, \alpha \in IOAct \bullet (P \xrightarrow{\alpha} P') \;\Rightarrow$
  $\exists\, Q' \bullet (Q \xrightarrow{\alpha} Q') \wedge \langle P', Q' \rangle \in S$

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 of State Bisimulation which relates the state spaces of P and Q (as opposed to being a relation between process terms):

Definition 4.2 : 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) $\langle e_P(x, \sigma_P),\ e_Q(x, \sigma_Q) \rangle \in SB$

We now state the desired association between state and process:

Theorem 4.3 : Correctness of state bisimulation
Let $P,\ Q$ be deterministic systems in normal form.

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

D.1 Symbolic State Evolution method

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 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:

1. we show that, for some input, the responses clearly disagree at some stage (i.e. P and Q are different), *or*

2. 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:

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

where sp_N and sq_N are expressions, dependent upon inputs $x_{N\text{-}1}, \; \cdots \; x_1$ and initial product state $\langle \sigma_P, \sigma_Q \rangle$, describing in symbolic terms the state that is reached after N-1 input steps.

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 4.4 : State Evolution rule

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

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

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

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

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

then we can conclude that the two deterministic systems P and Q are strongly bisimilar.

The VCG generates clauses which are logically equivalent to (a) and (b) in the state evolution rule.

For example, for (b) the VCG generates and expands the following expressions:

$$\begin{array}{l}
(r_P(x_1, sp_1) = r_Q(x_1, sq_1)) \wedge \\
sp_2 = e_P(x_1, sp_1) \wedge \\
sq_2 = e_Q(x_1, sq_1) \wedge \\
\qquad . \\
\qquad . \\
\qquad . \\
\wedge\ (r_P(x_N, sp_N) = r_Q(x_N, sq_N)) \wedge \\
sp_{N+1} = e_P(x_N, sp_N) \wedge \\
sq_{N+1} = e_Q(x_N, sq_N) \\
\quad \Rightarrow\ (r_P(x_{N+1}, sp_{N+1}) = r_Q(x_{N+1}, sq_{N+1}))
\end{array}$$

E References

[BGMW94a] H. Barringer, G. Gough, B. Monahan, and A. Williams. A Process Algebra Foundation for Reasoning about Core ELLA. Technical Report UMCS-94-12-1, Department of Computer Science, University of Manchester, December 1994. Available on the WWW, URL: `http://www.cs.man.ac.uk/csonly/cstechrep/Abstracts/UMCS-94-12-1.html`.

[BGMW94b] H. Barringer, G. Gough, B. Monahan, and A. Williams. A State Evolution Method for Verifying Hardware Systems. Forthcoming technical report, Department of Computer Science, University of Manchester, October 1994. Available on the WWW, URL: `http://www.cs.man.ac.uk/fmethods/projects/ELLA-PROJECT/state-evol-abstract.html`.

[BGMW94c] H. Barringer, G.D. Gough, B.Q. Monahan, and A. Williams. A Process Algebraic Semantics for Core ELLA. Technical Report UMCS-93-2-1, Department of Computer Science, University of Manchester, November 1994. Available on the WWW, URL: `http://www.cs.man.ac.uk/csonly/cstechrep/Abstracts/UMCS-93-2-1.html`.

[LW94] Harlequin Group Ltd., Cambridge. *The LispWorks Tutorial*, 1994. Edition 3.2.

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