FORMAL VERIFICATION SUPPORT FOR ELLA

IED 4/1/1357

ELLA Verification and Semantics Report - Year 2*

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`

May 13, 1992

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

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

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

1 Introduction

In this report we describe the current state of the verification framework outlined in chapter 10 of the report submitted at the end of year 1 [GB91], and illustrate its use by means of a series of examples.

This approach is, of course, based on automata theoretic methods, and in particular on a formal compositional semantics for a subset of ELLA, where the semantic objects are input-output labelled finite automata. The use of such an approach, however, is well-known to raise problems of "state explosion". Interestingly, our particular use of IO automata leads more to "transition" rather than "state" explosion, but the underlying problem remains the same.

One approach to easing this problem is the development of compact representations for the state/transition space. Binary decision diagrams (BDDs) [Bry86] have been shown, in several cases, to be rather successful in providing compact representations of very large state spaces. Claims of handling systems with 10^{50} and even 10^{120} reachable states have been made [BCL91]. However, the gain does depend critically on the particular system being modelled and the way in which they are used; such statements should be read together with a sort of "government health warning". Indeed, one major difficulty with the use of BDDs is how to determine an ordering for the BDD-variables — a bad choice will lead to an explosion in the size of the BDD.

As appendices to this report we attach reports which describe the way in which we are using the BDD approach and also present an alternative form of symbolic representation which attempts to address the variable-ordering problem mentioned above.

The full list of appendices is:

- **Implementors guide to the ELLA-VERIFIER package** [MPW92]

  This appendix documents the facilities provided by the Common Lisp package `ELLA-VERIFIER`, which forms the basis for the current verification toolset.

- **Operations for BDD manipulation—algorithms and interface** [Pei92]

  This appendix presents some extensions to the algorithms given by Bryant [Bry86] for the manipulation of boolean functions represented by BDDs. Also described is the interface to an implementation (in Common Lisp) of these and other operations and some higher level set-based operations built on them.

- **BDD semantics of BKE** [Wil92]

  This appendix describes a BDD based approach for representing efficiently an IO automaton semantics for ELLA. The problem is to develop a sufficiently compact representation for the semantics which also allows efficient application of formal verification techniques.

- **Logical representation using if expressions** [Mon92]

  This appendix describes an approach to symbolic representation which tries to lessen one of the well-known drawbacks of BDDs – namely finding a suitable variable ordering upon which to build a canonical representation.

2 Present verification approach

Existing methods of comparison of ELLA designs are obtained from exhaustive simulation of the two designs; this is an extremely time-consuming process and is often not complete. The approach taken within this project is to compare the two designs via their respective formal semantic descriptions, as shown in figure 1. Since our semantic models are based on IO automata, the task of comparison becomes one of testing equivalence or containment of the automata representing the designs. By that we mean check that the languages accepted / generated by the automata are the same. Since there are well-known automata-theoretic algorithms for performing complementation, intersection and testing emptiness of automata, testing for equivalence is somewhat simplified. This approach also enables us to compare ELLA designs with specifications given using other formalisms, as long as they have semantic interpretations in terms of IO automata. Since many interesting order properties can be easily expressed either directly as IO automata or using a temporal logic, which has a natural automaton based interpretation, the flexibility of our approach is apparent. An example verification of a temporal property is given in section 3.4.

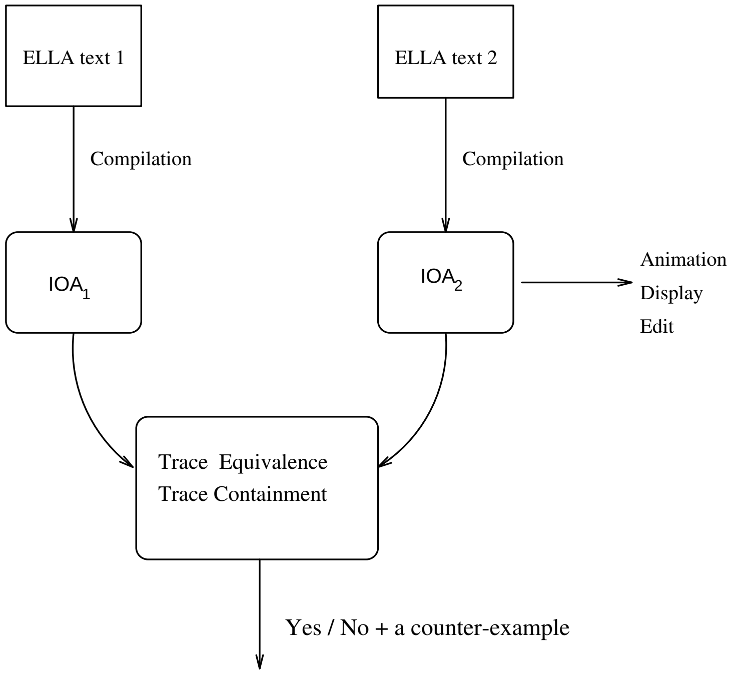

Figure 1: Verification approach

As indicated in figure 1, if two designs are found to be non-equivalent, a counter-example automaton is generated. This automaton contains all IO traces which distinguish the implementation from the specification, and can be used to direct debugging of the specification. One of the tasks for the next year of the project is to provide the extra hooks necessary to relate the counter-example traces back to the original ELLA text.

The current version of the verification system is based on a compiler for BKE in which the states of the automaton are explicitly enumerated. An appendix to this document [Wil92] contains an account of a reworking of the semantics to use an underlying representation of the IO automata symbolically, in terms of Binary Decision Diagrams (BDDs). We believe that a compiler based on such a representation will be capable of handling significantly larger examples than the current system. In fact, we already have, incorporated in the verification system, a compiler for a purely combinational subset of Kernel ELLA which is based on BDD technology. Examples of its use are described in section 5.

The remainder of this document describes in more detail the current verification approach by means of a series of examples.

3 The twisted ring counter

The twisted-ring counter was introduced as an illustrative example in Chapter 10 of [GB91]. Some of the introductory details below are reproduced from that chapter.

Informally, a twisted-ring counter cycles through a finite sequence of values. Here we restrict our attention to a 3-bit version of the counter. It has 6 valid values, corresponding to 6 valid states, and 2 invalid values (states). The normal counting cycle is through the 6 valid values, shown in table 1, Invalid

State	Outputs		
Label	a	b	c
0	0	0	0
1	0	0	1
3	0	1	1
7	1	1	1
6	1	1	0
4	1	0	0
0	0	0	0
.	.	.	.
.	.	.	.

Table 1: Sequence of Outputs

values can occur through device error, State 2: (0,1,0) and State 5: (1,0,1). However, should the counter enter an invalid state, the device will recover to a valid state within one, or two, steps. A reset is provided to force the counter to state 0.

3.1 First Description of Twisted-Ring Counter

We now give "algorithmic" descriptions for this example, first in ELLA and then in Kernel ELLA.

3.1.1 ELLA Description

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

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,f),
              (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.
```

The counter is formed by applying a function, `NEXT`, which computes the next value to be output from the value previously output by the counter, given via the `THREE_BOOL_DELAY` function.

3.1.2 Kernel ELLA Description

In the verification process, the above ELLA description is translated into Kernel ELLA, where the high-level description is written using a recursive let. The Kernel ELLA-text for the counter, produced by ELLA verification system, is included here for comparison.

```
TYPEDEC bool = NEW (f | t)

FNDEC NOT = (bool) -> (bool):
     BEGIN (LET x = input.)
     OUTPUT  CASE x OF ( (t : f))  ELSE t  ESAC
     END.
FNDEC OR = ((bool, bool)) -> (bool):
     BEGIN (LET a = input.  LET b = input.)
     OUTPUT  CASE (a,b) OF ( ((f,f) : f))  ELSE t  ESAC
     END.
FNDEC AND = ((bool, bool)) -> (bool):
     BEGIN (LET a = input.  LET b = input.)
     OUTPUT  CASE (a,b) OF ( ((t,t) : t))  ELSE f  ESAC
     END.
FNDEC NEXT = ((bool, (bool, bool, bool))) -> ((bool, bool, bool)):
     BEGIN (LET reset = input.  LET in = input.)
     OUTPUT  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,f))
                     ((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
     END.
FNDEC BOOL_DELAY = (bool) -> (bool):
     BEGIN ()
     OUTPUT IDELAY(f, 1).
     END.
FNDEC THREE_BOOL_DELAY = ((bool, bool, bool)) -> ((bool, bool, bool)):
    BEGIN (LET in = input.)
     OUTPUT (BOOL_DELAY in[1],BOOL_DELAY in[2],BOOL_DELAY in[3])
     END.
FNDEC TWIST_SPEC = (bool) -> ((bool, bool, bool)):
     BEGIN (LET reset = input.
             LET tbd = THREE_BOOL_DELAY nxt.
             LET nxt = NEXT (reset,tbd).)
     OUTPUT tbd
     END.
```

This Kernel ELLA-text is now compiled[1]; the resulting automaton is shown in figure 2. State 0 is the (normal) initial state. States 2 and 5 are unreachable from the initial state and correspond to the two invalid states described in the specification. The states have been given labels which happen to correspond to the integer values of the boolean output produced when leaving each state; the state labels do not have any further significance. The label 'r/xyz' on a transition arc means that if the input has value $reset = r$ then the transition can take place, producing an output for the counter of $a = x$, $b = y$ and $c = z$, where the outputs of the counter are labelled a, b and c. Note that it is coincidental and for convenience only that the states have labels corresponding to the integer values of the outputs on their departing arcs.

3.2 Gate-level Description of Twisted-Ring Counter

A "gate-level" design for the twisted-ring counter contains three latches with reset, one AND gate and one OR gate, as shown in figure 3. There is a single input $reset \in \{0, 1\}$ which resets the circuit to State 0 if it is raised.

3.2.1 ELLA Description

An ELLA-text description is now given for the gate-level ring counter.

[1] In normal circumstances, the Kernel ELLA-text is not actually output — the internal data structures produced by the ELLA–Kernel ELLA translator are fed directly into the Kernel ELLA compiler

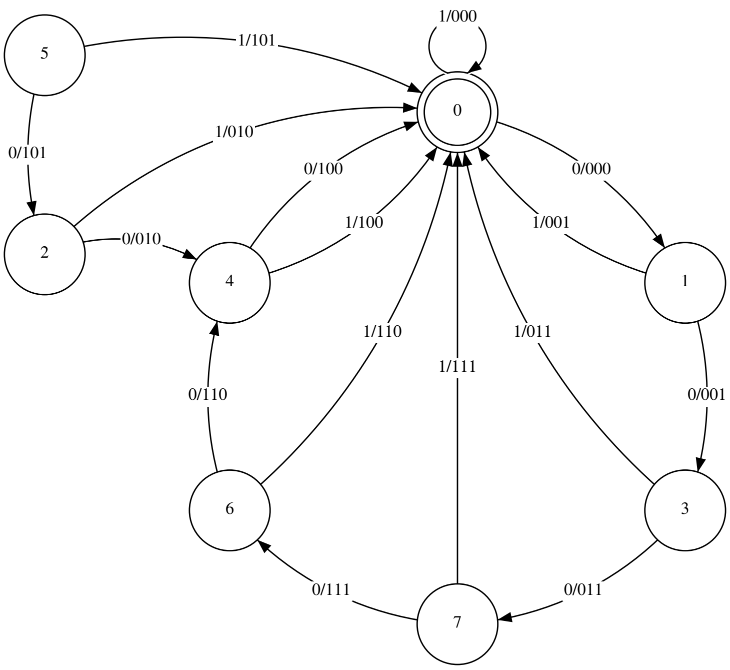

Figure 2: IO Automaton for Algorithmic Description of TRC

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

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:
   # defined in terms of NOT and OR: #
   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:
# i is input, r is reset:   #
# out(n+1) = i(n) if r(n)=f #
# out(n+1) = f    if r(n)=t #

      BOOL_DELAY(AND(i, NOT(r))).
```

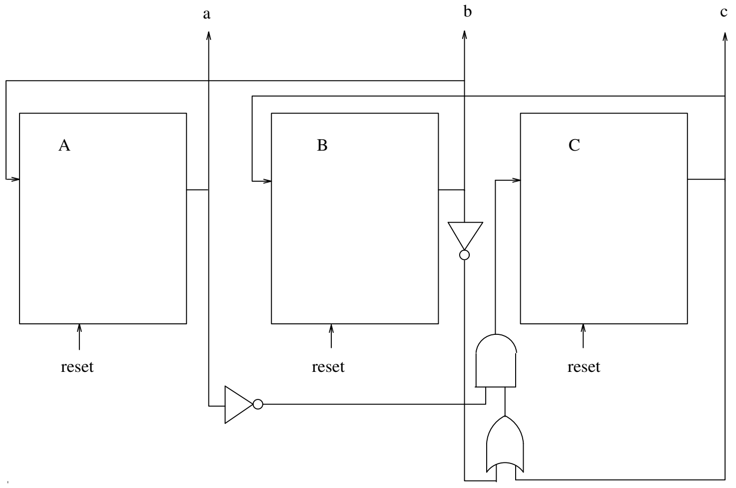

Figure 3: Twisted-Ring Counter

```
FN TWIST_IMPL = (bool: reset) -> [3]bool:
# twisted ring-counter #
  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.
```

3.2.2 Kernel ELLA Description

The ELLA-text above is then translated by the ELLA verification system into the following Kernel ELLA-text which is included again for comparison only and is not normally produced by the verification system:

```
TYPEDEC bool = NEW (f | t)

FNDEC NOT = (bool) -> (bool):
     BEGIN (LET x = input.)
     OUTPUT  CASE x OF ( (t : f))  ELSE t  ESAC
     END.
FNDEC OR = ((bool, bool)) -> (bool):
     BEGIN (LET a = input.  LET b = input.)
     OUTPUT  CASE (a,b) OF ( ((f,f) : f))  ELSE t  ESAC
     END.
FNDEC AND = ((bool, bool)) -> (bool):
     BEGIN (LET a = input.  LET b = input.)
     OUTPUT  CASE (a,b) OF ( ((t,t) : t))  ELSE f  ESAC
     END.
FNDEC BOOL_DELAY = (bool) -> (bool):
```

```
      BEGIN ()
      OUTPUT IDELAY(f, 1).
      END.
FNDEC LATCH = ((bool, bool)) -> (bool):
      BEGIN (LET i = input.  LET r = input.)
      OUTPUT BOOL_DELAY AND (i,NOT r)
      END.
FNDEC TWIST_IMPL = (bool) -> ((bool, bool, bool)):
      BEGIN (LET reset = input.
              LET a = LATCH (b,reset).
              LET b = LATCH (c,reset).
              LET c = LATCH (AND (NOT a,OR (NOT b,c)),reset).)
      OUTPUT (a,b,c)
      END.
```

3.2.3 IO Automaton Description

An IO automaton can now be constructed from the Kernel ELLA description of the counter. In fact, it is isomorphic up to state names with that of figure 2.

3.3 Twisted-Ring Counter Implementation with Redundant States

The twisted-ring counter implementation will now be modified so that one of the latches contains redundant components and operates by taking 'majority vote', a method frequently used in safety-critical applications. This will introduce additional but unreachable states into the automaton resulting from compilation, which will now not be isomorphic to the original design — it will, however, still be equivalent.

The 'safety-latch' contains three of the original latches — the output is determined by selecting the most popular signal value. The following ELLA-text produces this effect:

```
FN OR3 = (bool: a, bool: b, bool: c) -> bool:
   OR(a, OR(b,c)).

FN SAFELATCH = (bool: ip, bool: reset) -> bool:
BEGIN
    MAKE LATCH: a.
    MAKE LATCH: b.
    MAKE LATCH: c.
    JOIN (ip, reset) -> a.
    JOIN (ip, reset) -> b.
    JOIN (ip, reset) -> c.
    OUTPUT OR3(AND(a,b),AND(a,c),AND(b,c))
END.

FN TWIST_SAFE = (bool: reset) -> [3]bool:
  BEGIN
     MAKE SAFELATCH: 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.

FINISH
```

The operations `LATCH, AND, NOT` and `OR` have the same definitions as before.

When this is compiled, a 32-state automaton is produced. On examination, the automaton has a similar form to the original. However, for each input arc from a node *in-node* to a node *n* (except node 0) with label $0/out1$, there are now three additional unreachable nodes, forming a ‘composite’ node as shown in figure 4. Here the bold arcs and shaded node are the same as in the original automaton. The node *nores-node* corresponds to that reached in the original counter when $reset = 0$, with transition label $0/out2$.

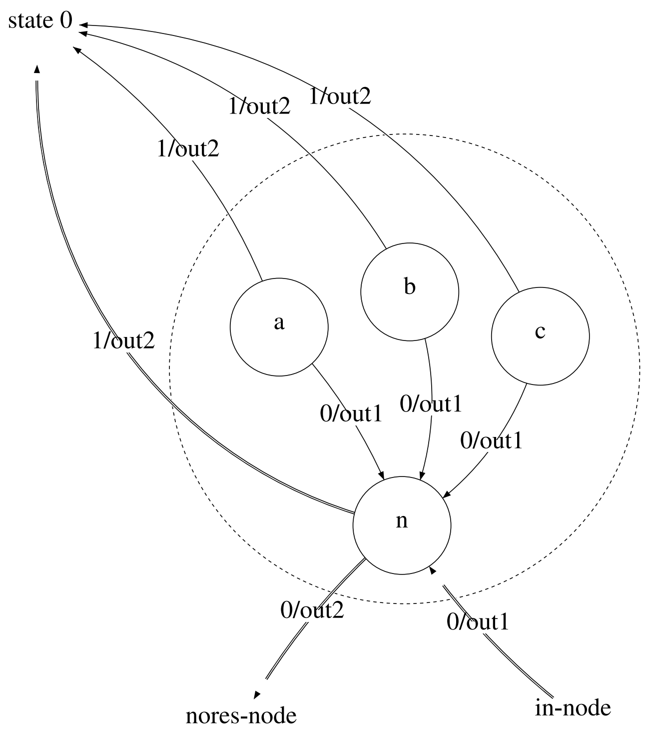

Figure 4: ‘Composite’ Node in TRC with Redundancy

This design is found to be equivalent to the original, even though the two automata involved are not isomorphic, because the equivalence-checker at present only considers reachable states. However, under certain circumstances it may be desirable to consider these unreachable states in verification checks, since they represent possible behaviour after a hardware fault or power-up has occurred.

3.4 Verifying a Temporal Property of the TRC

So far, two different ELLA design descriptions have been checked for equivalence. In Chapter 12 of [GB91] attention is given to verifying temporal properties of the twisted-ring counter. One property is that, no matter how *reset* behaves, the counter will output $(0,0,0)$ infinitely often. The property is established manually in [GB91]. Using the ELLA compiler, with a tool which examines state-machine paths, developed in an associated research project here at Manchester, it is possible to consider automatically the temporal properties of paths in the automaton for the twisted-ring counter.

In order to establish the above property, then the twisted-ring counter automaton must *not* have any paths with the negation of the property, which can be written as:

$$\text{*Not *Sometime *And } O1 \rightarrow 0. \text{ *And } O2 \rightarrow 0. \; O3 \rightarrow 0.$$

$O1, O2, O3$ are the output names for the counter. As, required, no paths are found when the automaton for the twisted-ring counter is searched.

An appropriate form of the specification language for use with the ELLA verification system is currently under consideration.

3.5 An incorrect implementation

In this section we replace, in the gate level ELLA-text description, the clause

```
JOIN (b, reset) -> a.
```

by the (incorrect)

```
JOIN (a, reset) -> a.
```

giving the new gate-level description

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

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:
   # defined in terms of NOT and OR: #
   #NOT( OR( NOT(x), NOT(y))).#
   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:
# i is input, r is reset:   #
# out(n+1) = i(n) if r(n)=f #
# out(n+1) = f    if r(n)=t #

      BOOL_DELAY(AND(i, NOT(r))).

FN TWIST_IMPL_ERR = (bool: reset) -> [3]bool:
# twisted ring-counter #
```

```
BEGIN
   MAKE LATCH: a.
   MAKE LATCH: b.
   MAKE LATCH: c.
   #replace b by a in next line:#
   JOIN (a, reset) -> a.
   JOIN (c, reset) -> b.
   JOIN (AND(NOT(a), OR(NOT(b), c)), reset) -> c.
   OUTPUT (a, b, c)
END.
```

The IO automaton corresponding to the compiled ELLA-text code is shown in figure 5.

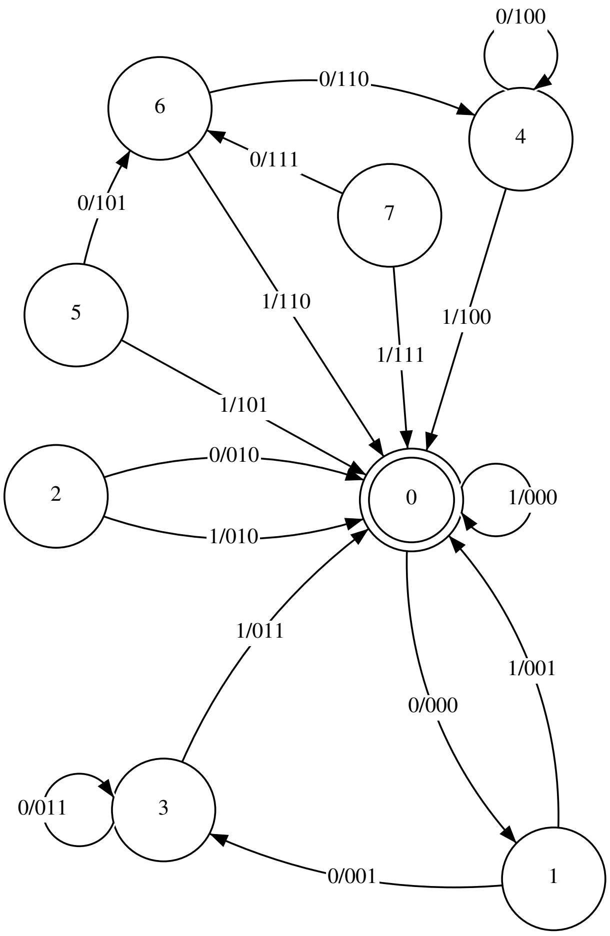

Figure 5: IO Automaton for incorrect gate-level implementation

The two automata for the implementation and specification are compared by the ELLA verification system. The differences can be observed in the resulting counter-example automaton, shown in figure 6. In this automaton, the accepting states are shown as shaded and the transitions at which the specification is violated are shown as dotted arcs. Notice that the node labels will not correspond to those appearing in figure 2, because they are created during the counter-example automaton generation. Only those transitions which may precede the occurrence of an error have been generated by the verification system, since these form the interesting and useful part of the counter-example path. In this example the only (minimum-length) path prefixes of interest are the two with I/O traces

0/000, 0/001, 0/011, 0/011 ...
0/000, 0/001, 0/011, 1/011 ...

These counter-example traces are automatically generated by the verification system. They are useful in debugging a design, especially if they are used to animate a schematic, with erroneous output signals highlighted. Subsequent examination of internal signals can be achieved by monitoring.

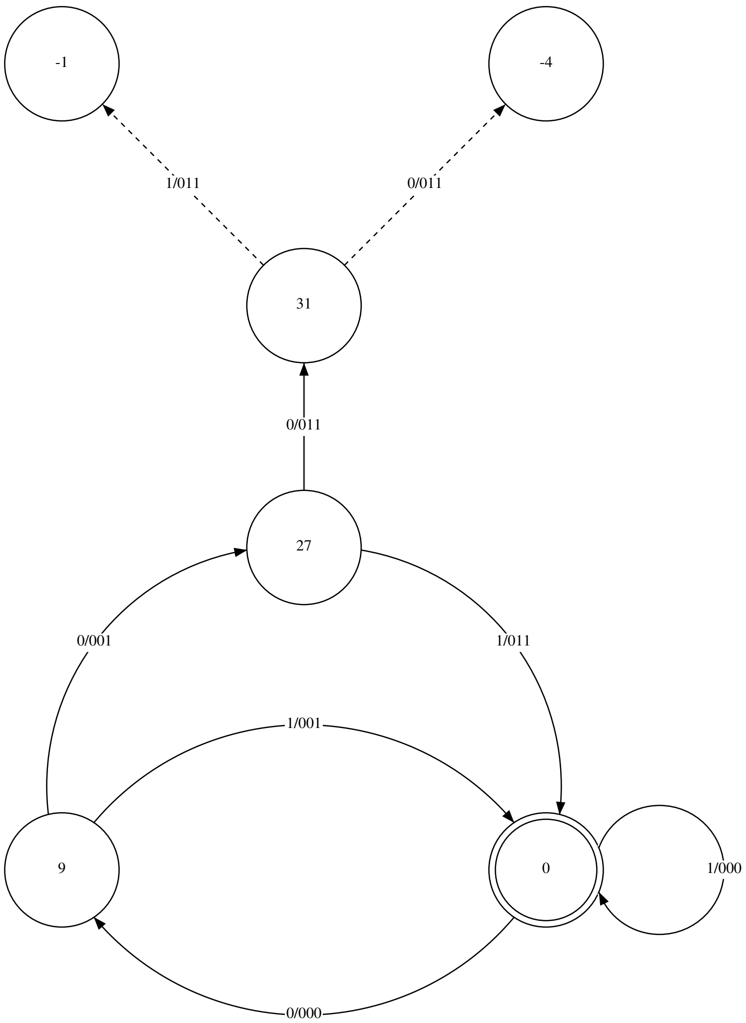

Figure 6: Counterexample IO Automaton for incorrect gate-level implementation

3.6 Monitoring signals

The IO-automata generated as semantic models have as outputs only those signals which are returned as part of function values. It will probably be useful to have the ability to *expose* internal signals as part of the feedback from verification. This can be done at the moment by slight modification of the original ELLA-text. As an example of this, suppose we wish to monitor the output of the *and* gate in the latch labelled *A* in the original twisted-ring counter circuit (see figure 3). A view of this circuit with the internals of the relevant latch exposed is shown in figure 7. Using the current semantic models we are

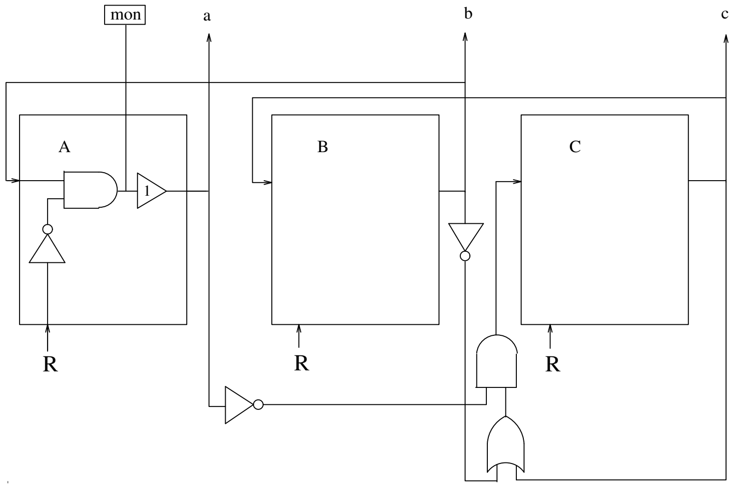

Figure 7: Twisted-Ring Counter with exposed latch

able to generate an automaton which exposes this signal by modifying the gate-level description in the following way.

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

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:
   # defined in terms of NOT and OR: #
   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:
# i is input, r is reset:   #
# out(n+1) = i(n) if r(n)=f #
# out(n+1) = f    if r(n)=t #

      BOOL_DELAY(AND(i, NOT(r))).

FN MLATCH = (bool: i, bool: r) -> [2]bool:
```

```
 BEGIN
    LET mon = AND(i, NOT(r)).
    MAKE BOOL_DELAY: del.
    JOIN mon -> del.
    OUTPUT(del, mon)
 END.

FN TWIST_IMPL_MON = (bool: reset) -> [4]bool:
# twisted ring-counter with monitor#
  BEGIN
     MAKE MLATCH: a.
     MAKE LATCH: b.
     MAKE LATCH: c.
     JOIN (b, reset) -> a.
     JOIN (c, reset) -> b.
     JOIN (AND(NOT(a[1]), OR(NOT(b), c)), reset) -> c.
     OUTPUT (a[1], b, c, a[2])
  END.

FINISH
```

The IO automaton corresponding to this ELLA-text is shown in figure 8. Alternative approaches for handling exposure of internal signals are currently under consideration.

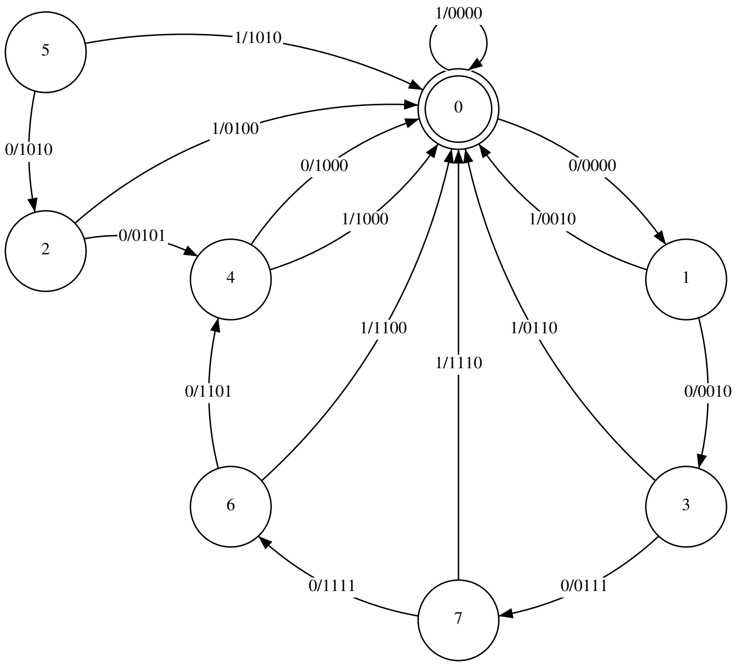

Figure 8: IO Automaton for gate-level implementation, with extra monitored signal

4 An extended twisted ring counter

The twisted-ring counter example will now be expanded so that the ELLA verification system can be exercised on slightly larger designs. The idea is to modify the original design so that basic counter units can be stacked together and operate as a single large counter, just as standard binary counters would be joined. The counter can still be reset as before. The new basic unit will then be shown to be identical to the original twisted-ring counter, when the input is suitably constrained.

For example, table 2 shows the output from a two-unit counter. To achieve this, the implementation

Time	Counter2			Counter1		
Step	a2	b2	c2	a1	b1	c1
0	0	0	0	0	0	0
1	0	0	0	0	0	1
2	0	0	0	0	1	1
3	0	0	0	1	1	1
4	0	0	0	1	1	0
5	0	0	0	1	0	0
6	0	0	1	0	0	0
7	0	0	1	0	0	1
.	.	.	.	.	.	.
.	.	.	.	.	.	.
34	1	0	0	1	1	0
35	1	0	0	1	0	0
36	0	0	0	0	0	0
37	0	0	0	0	0	1
.	.	.	.	.	.	.
.	.	.	.	.	.	.

Table 2: Sequence of Outputs for Two-Unit Counter

of the basic twisted-ring counter is modified to provide each latch in the original design with an *enable* line — if *enable* $= 1$ then the output can be updated, otherwise it is held at its previous value (the reset of course overrides this). An ELLA-text description is given for the modified gate-level design:

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

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:
   # defined in terms of NOT and OR: #
   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:
# i is input, r is reset:   #
# out(n+1) = i(n) if r(n)=f #
# out(n+1) = f    if r(n)=t #

      BOOL_DELAY(AND(i, NOT(r))).

FN ENABLE = (bool: e, bool: i, bool: last) -> bool:
   CASE e OF
     t: i
   ELSE last
   ESAC.

FN ENABLE_LATCH = (bool: i, bool: e, bool: r) -> bool:
# i is input, e is enable, r is reset:   #
# out(n+1) = i(n) if e(n) = t, r(n)=f    #
# out(n+1) = out(n) if e(n) = f, r(n)=f  #
# out(n+1) = f if r(n)=t                 #
BEGIN
   MAKE LATCH rdel.
   MAKE ENABLE: en.
   JOIN (e, i, rdel) -> en.
   JOIN (en, r)      -> rdel.
   OUTPUT rdel
END.

FN TWIST_ENABLE = (bool: enable, bool: reset) -> [3]bool:
# twisted ring-counter with enable and reset#
  BEGIN
     MAKE ENABLE_LATCH: a.
     MAKE ENABLE_LATCH: b.
     MAKE ENABLE_LATCH: c.
     JOIN (b, enable, reset) -> a.
     JOIN (c, enable, reset) -> b.
     JOIN (AND(NOT(a), OR(NOT(b), c)), enable, reset) -> c.
     OUTPUT (a, b, c)
  END.
```

When compiled, the IO automaton shown in figure 9 is produced. This is different from the automaton for the original twisted-ring counter, shown in figure 2, in only two respects: it has an extra *enable* input, and there are now self-loops on every node, traversed when $enable = 0, reset = 0$ and the current state is maintained.

If *enable* is now set permanently high, and removed from the input set, then the function *TWIST_ENABLE*

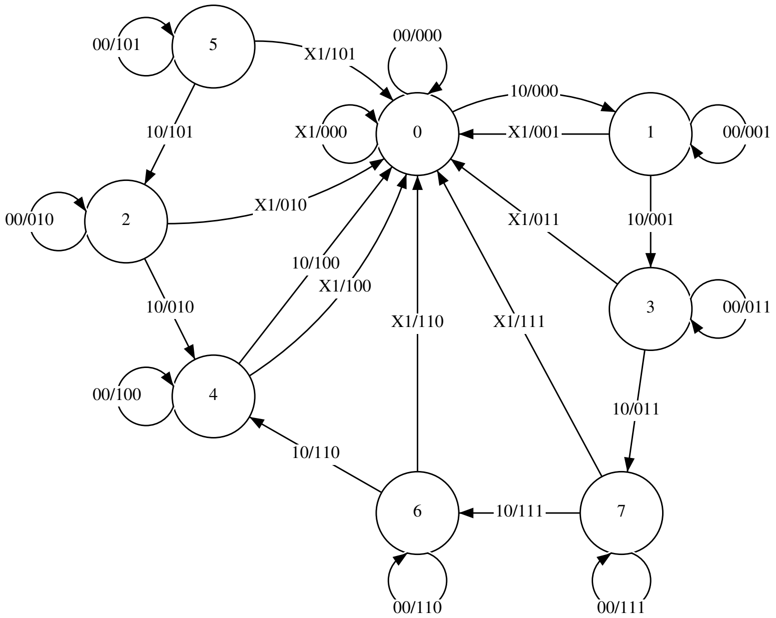

Figure 9: Twisted-Ring Counter with Enable

will have identical behaviour to the original twisted-ring counter:

```
FN TEST_TWIST_ENABLE = (bool: reset) -> [3]bool:
   TWIST_ENABLE(t, reset).
```

When the above ELLA-text is compiled then the resulting automaton is equivalent to the automaton for the original counter, as required. The comparison is performed automatically using the ELLA verification system.

4.1 Forming Larger Counters

The basic modified counter unit can now be joined, allowing larger design examples to be formed easily, with which to exercise the verification system.

As with standard binary counters, the basic twisted-ring counter units are linked by connecting the *enable* of one unit to combinational logic which recognises the required 'trigger' output pattern from the previous unit(s). The *enable* line is set high only when the trigger pattern is detected, thus incrementing the next counter unit in the sequence. The first unit receives an enable which is permanently high. Figure 10 shows how two counter units are joined together.

The following ELLA-text produces this two-unit counter:

```
FN TRIGGER = (bool: a, bool: b, bool: c) -> bool:
# a * ~b * ~c #
   AND(AND(a, NOT(b)), NOT(c)).

FN TWO_TWIST = (bool: reset) -> [6]bool:
BEGIN
   MAKE TWIST_ENABLE: twist2.
   MAKE TWIST_ENABLE: twist1.
```

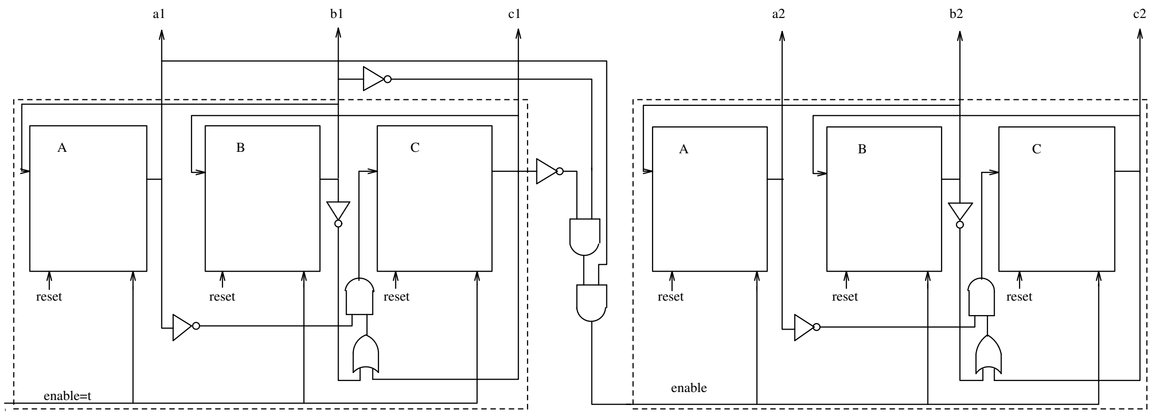

Figure 10: Two-Unit Counter Design

```
    MAKE TRIGGER: trigger.
    JOIN (t, reset) -> twist1.
    JOIN (twist1[1],twist1[2],twist1[3]) -> trigger.
    JOIN (trigger, reset) -> twist2.
    OUTPUT (twist2[1],twist2[2],twist2[3],twist1[1],twist1[2],twist1[3])
END.
```

An ‘algorithmic’ description can also be given for the two-unit design using *case* statements in exactly the same way as before, although the ELLA-text will not be presented here. After compilation the two IO automata are found to be equivalent, as required.

5 Verifying ALU’s

The twisted-ring counter examples demonstrate the first prototype implementation of the Kernel ELLA compiler, based directly on the semantics given in [GB91]. This uses an enumeration of the state set and transition relation for each IO automaton, and therefore suffers from both state- and transition-explosion.

At present a new compiler is being developed, based on a BDD semantics for Kernel ELLA [Wil92], using BDD manipulation tools developed in an associated SERC-funded project here at Manchester. Initially a partial compiler has been implemented for the purely combinational subset of Kernel ELLA restricted to the boolean dataset, called Combinational Boolean Kernel ELLA (CBKE), where all resulting automata have a single state. Constructs for delay, sample and RAM are therefore excluded.

The compiler, together with associated BDD tautology checker and counter-example generator, have been exercised on n-bit ALUs which perform seven operations — *ADD*, *SUB*, *INCA*, *INCB*, *EXOR*, *AND*, *OR* — determined by a 5-bit control code. The specification is partitioned functionally — n-bit units for each operation are constructed, and a selection unit passes the required result to the ALU outputs, determined by the control code. The implementation is described bit-wise, and constructed from n single-bit ALUs, shown in figure 11. A test harness is added to each design to filter out unused control codes, for which an ALU operation has not been defined.

The original 4-bit ALU is based on a design in [MM91], and due to the transition explosion (2^{14} transitions in the final automaton), it takes over two hours to compile one design using the ‘enumerated’ compiler. With this BDD-based compiler, two 4-bit ALUs (i.e., specification and implementation) are compiled and checked for equivalence in under 9 seconds. The compiler builds one BDD for each output

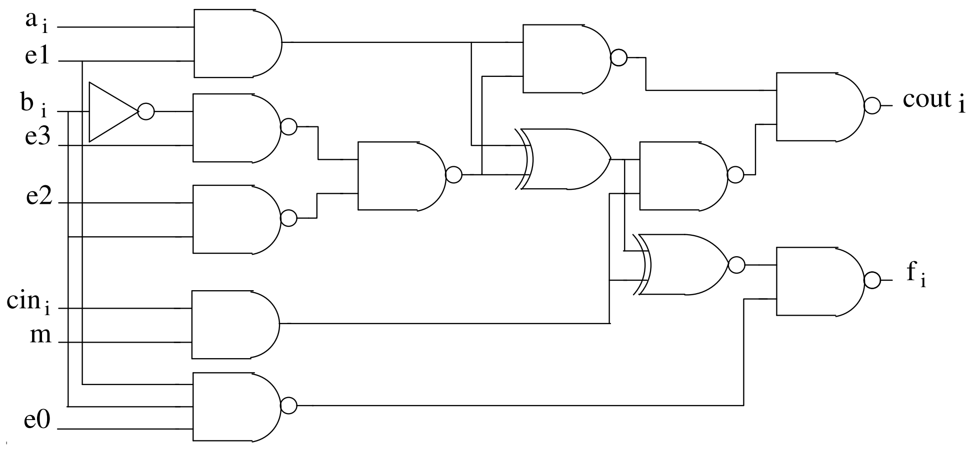

Figure 11: Single-Bit ALU Implementation

bit of the function from the CBKE-text. Equivalence-checking between two designs simply involves checking for equivalence between corresponding members of the two lists of output-BDDS.

Table 5 gives results obtained on a Sparc IPC for compilation and equivalence-checking, with $n \leq 192$ (this appears to be close to the limit in terms of size for this BDD implementation and particular machine). As usual, the variable order is vital in determining the size of resulting BDDs; the variable order used here

Size	Time (sec) (Loading)	Time (sec) (Equivalence)	Max. Nodes in output-BDDs
1	1.7	2.2	31
4	1.78	7.2	54
16	4.97	42.0	150
24	7.2	80.4	214
32	8.9	131.5	278
64	19.2	490	534
128	41.2	2056	1046
192	64	4735	1558

is $m < e0 < e1 < e2 < e3 < cin < a_0 < b_0 < \ldots < a_{n\text{-}1} < b_{n\text{-}1}$, where $m, \ldots, e3$ are the control variables, a_i, b_i, $i = 1, \ldots, n$ are the two input words and cin is the carry-in. This results in a linear growth rate of output-BDDs with n.

The BDD-based representation approach for IO automata is therefore very promising, and the BDD compiler will be extended to cater for the full Kernel ELLA language, as described in the appendix. Alternative representations, similar to BDDs but attempting to avoid the main problem of determining the best variable-ordering, are also considered in the appendix.

Although the work on BDDs is part of another research project, an appendix has been included for completeness, containing details on BDDs and various implementation techniques developed here at Manchester.

6 Conclusions and future work

In conclusion, we summarise the progress made during this year:

- A prototype implementation of a compiler taking a natural subset of Core ELLA as input and producing representations of I/O automata as semantic results has been implemented. Various comparison techniques have also been developed. See [MPW92] for further details on using the utilities.
- Progress has been made towards the development of a full semantics of Kernel ELLA where the semantics I/O automata are described in terms of characteristic functions. This is reported in [Wil92].
- Development of advanced representation techniques that make use of Binary Decision Diagram technology to represent logical relations - this is reported in [Pei92, Mon92].

This now leads on to the work programme for the coming project year:

Semantics of Kernel ELLA This completes the work started in year 2 extending the IO automaton semantic framework to include full Kernel ELLA. In particular, it will extend the BDD representations with ELLA types and will form the basis for the next version of the compiler.

Dataset reduction and its implementation This work reports on approaches to reducing the global state space of an ELLA design, in order to extend the scope of application of automata-theoretic verification methods. Strategies to be investigated include control and data space factorisation via static analysis and non-standard interpretation of data types.

Finally, we hope to integrate the outcome of this work into the Kernel ELLA verification system.

Modularisation and retiming Modular and compositional proof methods provide an approach to handling global state explosion which complements those reported in D2.3b. This work addresses aspects relevant to modular verification of ELLA designs, together with an investigation of retiming issues resulting from such abstraction.

Finally, we hope to incorporate these techniques by making them ready for inclusion in the next release of the ELLA verification environment.

References

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

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[GB91] Semantics and Verification Foundations for ELLA. Project report, University of Manchester and DRA Electronics Division, RSRE Malvern, May 1991.

[MM91] D J Mallon and W Mao. Circal analysis example set. Technical report, University of Strathclyde, September 1991.

[Mon92] Brian Monahan. Logical representation using if expressions. Project report, Department of Computer Science, University of Manchester, May 1992.

[MPW92] Brian Monahan, Martin Peim, and Alan Williams. Implementors guide to the ella-verifier package. Project report, Department of Computer Science, University of Manchester, May 1992.

[Pei92] Martin Peim. Operations for BDD manipulation — algorithms and interface. Project report, Department of Computer Science, University of Manchester, May 1992.

[Wil92] Alan Williams. A BDD representation of the IO Automaton Semantics for Boolean Kernel ELLA. Project report, Department of Computer Science, University of Manchester, May 1992.