A Semantics and Verification Framework for ELLA[1]

Howard Barringer   Graham Gough   Thomas Longshaw
Brian Monahan   Martin Peim   Alan Williams

Department of Computer Science
University of Manchester

March 30 1992

Abstract

We present an automata-based compositional semantics for an abstraction of the hardware design and description language ELLA [MPT85]. The semantics forms the basis for automata-theoretic verification of ELLA descriptions, or programs. We further outline the principles of the verification strategy and use a small example, a twisted-ring counter, to demonstrate both the semantics and the verification techniques.

Contents

Chapter 1

ELLA and hardware verification

1.1 Introduction

ELLA is a system pioneered by RSRE for the high-level description of complex VLSI designs [MPT85]. The system[Com90] is commercially available; it was originally marketed by Praxis Systems, and is now available through Computer General Electronic Design Ltd (CGED). The system includes the ELLA language and its compiler, an application support environment, a simulator and other design support tools. ELLA is a parallel language and is used to describe and model circuit behaviour by defining the nodes, connections and signal flows in a circuit network.

The work we present here lays part of the foundation of a larger collaborative project, namely, to develop a design support environment, including design tools based on formal verification techniques, capable of handling ELLA designs of up to 100K gates. Clearly this size presents a major challenge to the current verification technology. Sophisticated techniques need to be applied in order to extend the capabilities of the particular approach towards the desired goal. Those aspects, however, will be the subject of later reports and papers. Here we wish to concentrate on introducing the basic ideas.

ELLA is, syntactically, a large language and comprises many different language features. However, many of these features can be reformulated in terms of "more primitive" features. Naturally, one of the first steps towards our project goal has been to identify the basic language elements to which we can expect to apply formal verification. This has led to Core ELLA [HM91] and more recently to Kernel ELLA. However, before one can contemplate formal verification of program properties, a formal semantics for the programming language is a necessity. In the case of ELLA, as with other HDLs, this does not currently exist. There is obviously some, good, informal understanding of desired semantics (in plural), and indeed, there is a *de facto* standard implied by the CGED ELLA simulator (in this respect it is fortunate that there is only one marketed simulator available). However, this does not assist with formal verification.

Recently there have been three independent approaches to providing a formal semantics for ELLA. At Edinburgh operational semantics are provided for two simplified versions of ELLA, called *micro*ELLA[Goo90a] and *pico*ELLA [Goo90b]. At Cambridge, a system is being developed to translate ELLA-text into HOL [BGHT90], so that the HOL verification system and its associated tools can be employed. At Manchester, we gave a structured operational semantics (SOS) for a small parallel language with delays [BGM91]. That work was designed to capture the fundamental semantic aspects of ELLA and stood as a pilot scheme towards providing a full reference semantics for Core ELLA. The principal purpose of the formal semantics, for our project at least, is not to act as a reference in that sense; it is, rather, to act as a semantic foundation for verification. Although we could, quite inefficiently, use the SOS style semantics in that respect, it seems better to provide a more direct form. The verification technology we wish to apply is principally automata-theoretic, or state-based. It provides automated verification procedures and has certainly been shown to be successful in the area of communication protocols. It will easily lend itself to verifying equivalence, and refinement, in appropriate senses, of ELLA descriptions, as well as enabling verification against formal specifications written in more expressive formalisms, such as temporal logics. Furthermore, there is a natural correspondence between automata and ELLA components. We exploit this in the formal semantic definitions appearing in the rest of this paper. A first step in the verification of temporal properties for ELLA designs is contained in [Bec87].

We use IO automata to provide the semantic values for Core ELLA components. These are essentially a form of extended Mealy finite state machine where states carry no information, while the input and output values are all attached to the transitions, and augmented to be able to characterise infinite behaviours. They provide a very compact model for representing ELLA descriptions. Further, there are automaton composition operations which provide a natural means for constructing the semantics of an ELLA design from its constituent (sub-)designs.

This paper has the following structure: in the sub-section below a more detailed overview is presented of the state-based verification approach being adopted. Section 2.1 then defines Boolean Kernel ELLA (or BKE), the sub-language of ELLA that we are currently handling, and presents its semantics informally through a small example of an RS flip flop. Section 3.2 follows with an introduction to IO automata and concludes by demonstrating their use in the composite modelling of the RS flip flop. Section 3.3 presents, in VDM, the compositional IO automaton semantics of BKE. This is then exercised by application to a twisted ring-counter in Section 4.4.

1.2 Verification Techniques

The verification approach applied in this paper is one based on model checking. As such, the architecture of the system consists of two main components: a state machine generator and a model-checker.

The state machine generator takes as input a program and generates a finite state graph which represents all possible runs of the program. Such state graphs can, of course, only be generated for finite state programs i.e. those with bounded memory requirements. This graph is one of the inputs to the model checker, the other being a specification, or property, of the program. The task of the model checker is to determine whether the state graph, and hence the program, possesses the property. There are many approaches to this task, varying according to the characteristics of the specification language. We outline one that fits well with a range of specification formalisms. The idea is that we first construct, from the given specification, a state graph that characterises all runs of programs that *do not* possess the given property. This task is reasonably straightforward, although it can be computationally expensive. We then determine whether the intersection of the program state graph and the latter graph is empty, i.e. whether there are any runs common to both. If the intersection is empty then we can conclude that *all* runs of the program possess the desired property. If the intersection is non-empty then we have a run which does not possess the property, which can be returned to the user as a counterexample.

There are several existing systems which are based on this approach. Clarke et al [CE82, CES83, BC86] developed model checkers for a branching time logic, CTL, for verifying properties of programs written in their state machine language, SML. Sifakis et al [RRSV87] have developed Xésar, a model checking based verification system for the language ESTELLE, a specification language for describing communications protocols. In Manchester, we have developed a generic system SMG [GB88, BFG89] which allows the user to specify the input language syntax and semantics, and provides model checking support for linear time temporal logics. The use of this generic approach was not used for ELLA, since the compilation process would not have been efficient enough for use with large designs.

In the above architecture, the specification is usually given in a language, such as temporal logic, suitable for describing particular desired aspects of the programs behaviour. This language could, however, be the original programming language itself, in which case the success of the emptiness test described above would mean that all runs of the program were also runs of the “specification” program. This corresponds to one notion of program refinement. In the particular case of ELLA, however, since all ELLA programs are deterministic, a more direct and efficient approach may be taken to refinement testing.

1.3 Acknowledgements

The authors wish to thank Mike Hill and John Morison (DRA Malvern (RSRE)), for many useful discussions concerning ELLA during the development of the semantics presented here.

Chapter 2

Boolean Kernel ELLA

2.1 Syntax of BKE

We give a composite syntax for a simple ELLA-like language called BKE. It extends the LD language (i.e. Logic with Delay) from [BGM91] by including a "let" expression to allow local bindings of variables and function declarations to provide encapsulation and modularity of designs.

BKE is related to Core ELLA [HM91] through its inclusion of function declarations and delays. It differs from Core ELLA in that "let" bindings can be recursive and that it does not include the ELLA model of data types.

As usual, the convention used in the composite syntax is that non-terminals correspond to types and are indicated by names having their first letter in capitals. Instances of these types which appear on the right-hand side of productions have the same name, but are written in lower-case.

So, the composite syntax for BKE is as follows:

$$
\begin{array}{rcl}
\textit{Closure} & ::= & \textit{fndecl}_1, \cdots, \textit{fndecl}_{k+1}. \\
\textit{Fndecl} & ::= & \texttt{fn}\ \textit{fnname}(\textit{var}_1, \cdots, \textit{var}_n) = \textit{expr} \\
\textit{Binding} & ::= & \textit{bind}_1, \cdots, \textit{bind}_{k+1} \\
\textit{Bind} & ::= & \textit{var}\ =\ \textit{expr} \mid (\textit{var}_1, \cdots, \textit{var}_n) =\ \textit{expr} \\
\textit{Expr} & ::= & \textit{const} \mid \textit{var} \mid \textit{let} \mid \textit{row} \mid \textit{case} \mid \textit{apply} \mid \\
 & & \textit{delays} \mid \textit{sample} \mid \textit{bram} \mid \textit{biop} \\
\textit{Const} & ::= & 0 \mid 1 \\
\textit{Var} & ::= & \textit{id} \\
\textit{Let} & ::= & \texttt{let}\ \textit{binding}\ \texttt{in}\ \textit{expr} \\
\textit{Row} & ::= & (\textit{expr}_1,\ \cdots, \textit{expr}_{k+1}) \\
\textit{Case} & ::= & \texttt{case}\ \textit{row}\ \texttt{of}\ \textit{caseitem}_1,\ \cdots,\ \textit{caseitem}_{k+1} \\
 & & \texttt{else}\ \textit{expr}\ \texttt{end} \\
\textit{CaseItem} & ::= & \textit{chooser} \longrightarrow\ \textit{expr} \\
\textit{Chooser} & ::= & (\textit{choice}_1,\ \cdots, \textit{choice}_{k+1}) \\
\textit{Choice} & ::= & \texttt{X} \mid \textit{const} \\
\textit{Apply} & ::= & \textit{fnname}(\textit{expr}_1,\ \cdots,\ \textit{expr}_n) \\
\textit{Delays} & ::= & \textit{idelay}(\textit{expr}, \textit{const}, \textit{nat}) \mid \\
 & & \textit{delay}(\textit{expr}, \textit{const}_1, \textit{nat}_1, \textit{const}_2, \textit{nat}_2) \\
\textit{Sample} & ::= & \textit{sample}(\textit{expr}, \textit{const}, \textit{nat}_1, \textit{nat}_2) \\
\textit{BRAM} & ::= & \textit{bram}(\textit{expr}_1, \textit{nat}, \textit{const}, \textit{expr}_2, \textit{expr}_3, \textit{expr}_4) \\
\textit{Biop} & ::= & \textit{notex} \mid \textit{orex} \\
\textit{Notex} & ::= & \neg\, \textit{expr} \\
\textit{Orex} & ::= & \textit{expr}_1\ +\ \textit{expr}_2 \\
\textit{FnName} & ::= & \textit{id} \\
\textit{Nat} & ::= & \mathbb{N}
\end{array}
$$

In the production for Choice, the `X` symbol stands for “don’t care”.

2.1.1 Intuitive semantics of BKE

A BKE description, or closure, consists of a list of function declarations. Implicit in the model underlying BKE is a global clock, on the ticks of which the values of input signals are examined. Thus the model is synchronous and based upon states, rather than events. Consider a call of one of the functions in the closure. At any given clock tick, the function body, together with the given inputs and function declarations, is used to determine the output behaviour. As in ELLA, the use of function definitions and instantiations provides the main technique used to modularise descriptions.

We illustrate this briefly by giving descriptions of an RS flip flop, shown in Figure 2.1, using both ELLA and BKE. The corresponding ELLA text for the RS flip flop is as follows:

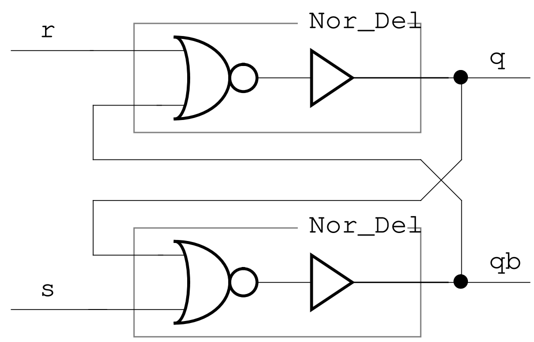

Figure 2.1: RS Flip Flop

```
TYPE bool = NEW(h | l).

FN BOOL_DELAY (bool) -> bool:
#init value l, delay 1 unit: #
DELAY(l,1).

FN OR = (bool: x y) -> bool:
CASE (x,y) OF
  (l,l)              : l,
  (bool,h) | (h,bool): h
ESAC.

FN NOT = (bool: x) -> bool:
CASE x
OF l: h,
   h: l
ESAC.
```

```
FN NOR = (bool: x y) -> bool:
#defined in terms of NOT and OR: #
NOT(x OR y).

FN NOR_DEL = (bool: x y) -> bool:
BOOL_DELAY(NOR(x,y)).

FN RSFF =(bool: r s)->(bool,bool):
BEGIN
  MAKE NOR_DEL: q qb.
  JOIN (r, qb) -> q,
       (q, s) -> qb.
  OUTPUT (q, qb)
END.
```

As an illustration of how ELLA is related to BKE, an equivalent description in BKE would be:

```
fn NOR(x, y) = ¬(x + y),

fn BOOL_DELAY(y) = idelay(y,0,1),

fn NOR_DELAY(x, y) = BOOL_DELAY(NOR(x,y)),

fn RSFF(r s) =
  let q  = NOR_DEL(r,qb),
      qb = NOR_DEL(s,q)
  in (q,qb).
```

Thus the BKE description describes the behaviour of the output signal history of the device, given the input signal history. These can be represented in a very natural way as IO automata. The automaton semantics for this example is given in section 3.2.2.1.

2.1.2 ELLA Features not covered by BKE

In this preliminary version of BKE it has not yet been possible to include the RAM primitive operation, the full typing mechanism and the unspecified value '?' available in Core ELLA. For all but the last feature, it is clear how to extend the present BKE definitions and to define special function objects to include these features (see Chapter 3).

The exact interpretation of '?' is more difficult to ascertain. In [BGHT90], '?' was interpreted as an extra value included in each data type. In [Com90], '?' may be used in the following ways:

- As an input value to a CASE expression, it acts as an unknown but legal value of the appropriate type. It will match any appropriate type name in a chooser-value during choice determination. For example, an input of '?' would match the chooser-value 'bool', but would produce a '?' when tested against the chooser-value '(0 | 1)',
- as an output from an under-determined CASE expression (i.e. if an input does not match any chooser-value, and there is no ELSE clause),
- as a 'don't care' input value for simulation,
- as a 'don't care' value for delay, and sequence variable initial values,
- as an output from ARITH (BIOP) expression when return integer value goes out of range.

If '?' were introduced into BKE then the definitions of the basic functions would be extended as follows:

$$\neg(?) \quad = ?$$

$$(1+?) = 1 \quad (0+?) \quad = ? \qquad (?+?) = ?$$

The '?' is acting here as an additional special value in the data type. Work has begun on investigating algebraic models of '?', one of which identifies the behaviour of '?' with a notion of *ambiguity*[Mon91].

Chapter 3

Automaton Semantics for Boolean Kernel ELLA

3.1 Introduction

In order to establish properties of designs given in a particular hardware description language, some formal meaning must be given to them. This is provided by giving a mathematical semantics for the design language.

Hardware systems are in principle finite state, i.e. they have bounded memory. Automated verification methods have been developed for finite state programs [GB88, BFG89], so that the properties of state machines, such as equivalence, can be determined automatically. Hence, by semantically mapping the specification and implementation languages for hardware into state machines, such automatic verification techniques can be applied to hardware systems. This indicates that existing hardware design languages need semantics from which a state machine description can be easily extracted for each design. In the following two chapters we present two different semantic presentations that enjoy this property. In this chapter a semantics for Boolean Kernel ELLA is based directly on state machine descriptions, i.e. the semantic objects *are* IO Automata — a particular form of transition-labelled finite state machine. Thus, the semantic objects themselves can be directly used for verification purposes.

In the rest of this chapter, we introduce IO automata and give motivational examples for their use in the semantics of hardware descriptions, then we present a compositional semantics for BKE using IO automata as the underlying semantic objects.

3.2 IO Automata and Hardware Components

The development of IO automata is part of the adaptation of well-known ideas from automata and language theory to the description of general system behaviour[VW86]. In some sense, they provide a generalisation of the *Mealy* finite state automaton from standard automata theory[HU79, Shi87]. They have some similarity to the structures of the same name studied by Lynch and Tuttle [LT88].

More formally, we take a (basic) IO automaton as a 7-tuple comprising a set of states, an input alphabet, an output alphabet, an input/output-labelled transition relation, a distinguished set of states — the initial states, a distinguished set of states – the terminal states, and a set of subsets of states — the acceptance subsets. The latter component is a generalised form of Büchi-style acceptance condition and allows for the definition of restricted sets of infinite runs of the automata. For the purpose of giving meaning to ELLA components, this aspect can be ignored as we require all infinite runs to be accepting. In fact, in the definition below we omit both terminal and acceptance sets.

Using the meta language of VDM (see [Jon90] for an introduction to VDM) we have the following definitions.

$$
\begin{aligned}
\mathit{Trans} :: \ & \mathit{from} : \mathit{State} \\
& \mathit{ivals} : \mathit{Name} \xrightarrow{m} \mathit{Val} \\
& \mathit{ovals} : \mathit{Name} \xrightarrow{m} \mathit{Val} \\
& \mathit{to} : \mathit{State}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{IOA} :: \ & \mathit{states} : \mathit{State}\text{-set} \\
& \mathit{inputs} : \mathit{Name}\text{-set} \\
& \mathit{outputs} : \mathit{Name}\text{-set} \\
& \mathit{transitions} : \mathit{Trans}\text{-set} \\
& \mathit{start} : \mathit{State}\text{-set}
\end{aligned}
$$

The invariant on *IOA* states that the start states are genuine states of our machine and that outputs may not also be inputs. Furthermore, the input and output mappings for each transition have appropriate domains and the *from* and *to* states must be in the state set.

$$\mathit{inv\text{-}IOA} : \mathit{state}\text{-set} \times \mathit{Name}\text{-set} \times \mathit{Name}\text{-set} \times \mathit{Trans}\text{-set} \times \mathit{State}\text{-set} \to \mathbb{B}$$

$$
\begin{aligned}
& \mathit{inv\text{-}IOA}(ss, i, o, ts, st) \ \triangleq \\
& \quad st \subseteq ss \land i \cap o = \{\,\} \land \\
& \quad \forall t \in ts \cdot \\
& \qquad \mathit{from}(t) \in ss \land \mathit{to}(t) \in ss \land \\
& \qquad \mathrm{dom}\, \mathit{ivals}(t) \subseteq i \land \mathrm{dom}\, \mathit{ovals}(t) = o
\end{aligned}
$$

3.2.1 IO Automaton Examples

3.2.1.1 NOR Gate

Consider a NOR gate (without delay) over the two valued set $\{0, 1\}$. We model it by a single-state automaton with four transitions and is shown in Figure 3.1. For convenience, its transitions are repeated in tabular form in Table 3.1. The automaton has state set $\mathit{states} = \{n\}$, initial state set $\mathit{start} = \{n\}$

From State	Input x	y	Output o	To State
0	0	0	1	0
0	0	1	0	0
0	1	0	0	0
0	1	1	0	0

Table 3.1: Transition Table for Nor Gate

and value set $\mathit{Val} = \{0, 1\}$. The labelling on the transitions in each automaton has been simplified for presentation purposes. Each mapping from label to value has been replaced by a value only, whose position determines the label to which it is assigned. In Figure 3.1 each label has the form *ab*/*c*, which

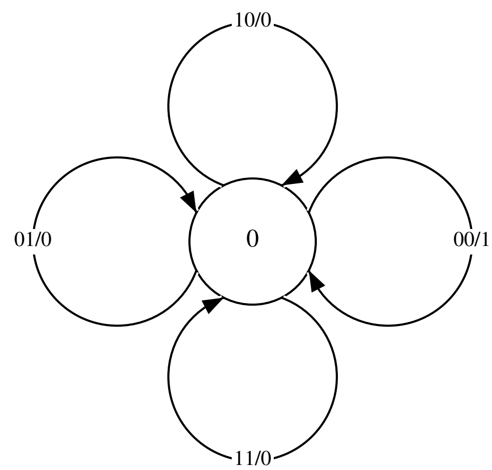

Figure 3.1: IO Automaton for Nor Gate

represents input labels *x*, *y* and output label *o*, having values *a*, *b* and *c* respectively. From the given labellings, or transition table, it is clear that the output on any transition is the "nor" of the two inputs.

3.2.1.2 Unit Delay

Consider a unit delay function which delays the given input by one (notional) step. It can be modelled by a two-state automaton as shown in Figure 3.2, with initial state 0. Consider the possible transitions. Starting in the initial state, if the input is 0, then the output is 0 and the state does not change. If, however, the input had been 1, then 0 would have again been output but the new state would have been state 1. The idea is that state 0 corresponds to having input a 0 on the previous transition, i.e. all incident transitions have input 0, and its emergent transitions have output 0, thus effecting a unit delay on 0 inputs. Furthermore, the incident transitions of state 1 all have input 1 and the emergent transitions have output 1, thus effecting unit delay for 1 inputs.

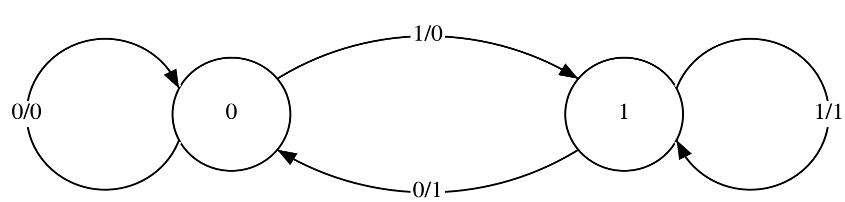

Figure 3.2: IO Automaton for Unit Delay

3.2.2 Product IO Automata

Given two, or more, automata representing the behaviour of certain hardware components, we require some form of composition on automata which will correspond to "connecting" the hardware components together. For example, we might wish to connect a Unit Delay onto the output of a NOR gate forming a unit-delay-NOR gate. The composition required is a form of *product*.

There are, in fact, many ways of forming products of automata that may be of interest in automata-based verification. In the following semantic descriptions, we use a *causal* product as principal operation for constructing product automata. It is a "synchronous" product in that the resulting automaton produces a state transition only when each of the participants (in the product) have taken a transition of their own. In particular, it takes the usage of inputs and outputs between participants into account, so involving the notion of *causality* between participants. Furthermore, any input of one that happens to be an output of the other is hidden within the resulting product automaton. Therefore all communications exchanged between the participants are hidden from third-party external observers.

A VDM presentation of causal product follows.

$causal\text{-}prod : IOA \times IOA \to IOA$

$causal\text{-}prod(a_1, a_2) \quad \triangleq$
  let $mk\text{-}IOA(ss_1, i_1, o_1, ts_1, st_1) = a_1$ in
  let $mk\text{-}IOA(ss_2, i_2, o_2, ts_2, st_2) = a_2$ in
  let $o = o_1 \cup o_2$ in
  let $i = (i_1 \cup i_2) - o$ in
  let $ts = combine\text{-}transitions(i, o, ts_1, ts_2)$ in
  $mk\text{-}IOA(ss_1 \times ss_2, i, o, ts, st_1 \times st_2)$

pre $outputs(a_1) \cap outputs(a_2) = \{\,\}$

$combine\text{-}transitions(i\text{-}sig, o\text{-}sig, ts_1, ts_2) \quad \triangleq$
  $\{prod\text{-}transition(i\text{-}sig, o\text{-}sig, tn_1, tn_2) \in Trans \mid$
    $tn_1 \in ts_1 \wedge tn_2 \in ts_2 \wedge compatible\text{-}transition(tn_1, tn_2)$
  $\}$

$compatible\text{-}transition(tn_1, tn_2) \quad \triangleq$
  let $mk\text{-}Trans(\text{-}, im_1, om_1, \text{-}) = tn_1$ in
  let $mk\text{-}Trans(\text{-}, im_2, om_2, \text{-}) = tn_2$ in
  let $iom_1 = im_1 \cup om_1$ in
  let $iom_2 = im_2 \cup om_2$ in
  let $com\text{-}nms = \mathrm{dom}\, iom_1 \cap \mathrm{dom}\, iom_2$ in
    $com\text{-}nms \lhd iom_1 = com\text{-}nms \lhd iom_2$

$prod\text{-}transition(i\text{-}sig, o\text{-}sig, tn_1, tn_2) \quad \triangleq$
  let $mk\text{-}Trans(s_1, im_1, om_1, s_1') = tn_1$ in
  let $mk\text{-}Trans(s_2, im_2, om_2, s_2') = tn_2$ in
  let $val\text{-}map = (im_1 \cup im_2) \cup (om_1 \cup om_2)$ in
  let $im = i\text{-}sig \lhd val\text{-}map$ in
  let $om = o\text{-}sig \lhd val\text{-}map$ in
    $mk\text{-}Trans((s_1, s_2), im, om, (s_1', s_2'))$

Note that the partiality of *map union* ($\_ \cup \_$) and of *map subtraction* ($\_ - \_$) is used here to constrain the set of resulting transitions in the product to those that are consistent.

3.2.2.1 Example: IO Automaton Construction for RS Flip Flop

To illustrate the automaton composition rule presented above, an IO automaton for an RS flip flop is composed from the automata representing two NOR gates, each having unit delay; the ELLA-text and BKE description for the RS flip flop appeared earlier in §2.1.1.

The undelayed NOR gate and unit-delay appeared above. To obtain a unit-delay-NOR gate automaton we construct the causal product of the unit-delay and the NOR gate automata, forming a two-state machine. Output from the NOR gate is connected to the input of the unit-delay. The IO automaton for the unit-delay-NOR gate has input labels x, y, output label o, state set $states = \{0, 1\}$, initial state set $start = \{0\}$ and value set $Val = \{0, 1\}$, and is shown in Figure 3.3.

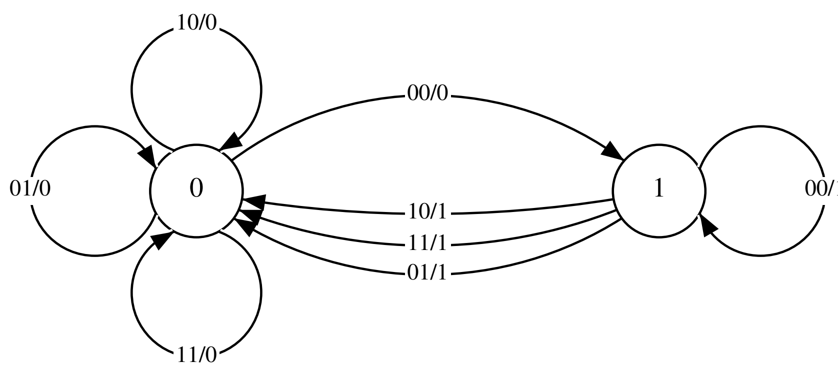

Figure 3.3: IO Automaton for Delay-Nor Gate

Now, two instances of the delay-NOR gate automaton are composed, using the causal product, to form the RS flip flop machine shown in Figure 3.4. Intuitively, this is performed by first taking all combinations of the 8 transitions from two instances of the delay-NOR gate automaton, resulting in a set of 64 pairs[1]. The first automaton instance has inputs labelled $x1$ and $y1$, and output labelled $o1$; the second instance has input labels $x2$ and $y2$ and output label $o2$. Input $y1$ is connected to output $o2$ and input $x2$ is connected to output $o1$, simply by renaming and hiding those inputs. Inputs $x1$ and $y2$ are labelled r and s respectively; the output labels remain unchanged.

Conflicting transitions can now be removed from the transition table. Table 3.2 shows two examples of transitions after the delay-NOR gates have been renamed; the first transition contains a conflict on the output labelled $o2$ (since $y1 = 1$ and $o2 = 0$) and is deleted from the final transition table. The second transition is valid since both the values for $o1$ and both for $o2$ are in agreement. The now redundant information for the renamed inputs $x2$ and $y1$ is removed, and the two component state labels i and j are paired to form the resulting single state label (i, j), with $i, j \in \{0, 1\}$. The transitions which remain in the automaton are shown in Figure 3.4. A transition *ab*/*cd* in the diagram represents inputs r and s, outputs $o1$ and $o2$, having the values a, b, c, and d respectively. The bold arrow which emerges from

[1] In an implementation of the causal product operation, the complete transition tables are not constructed in this way. This is done here purely for explanatory purposes.

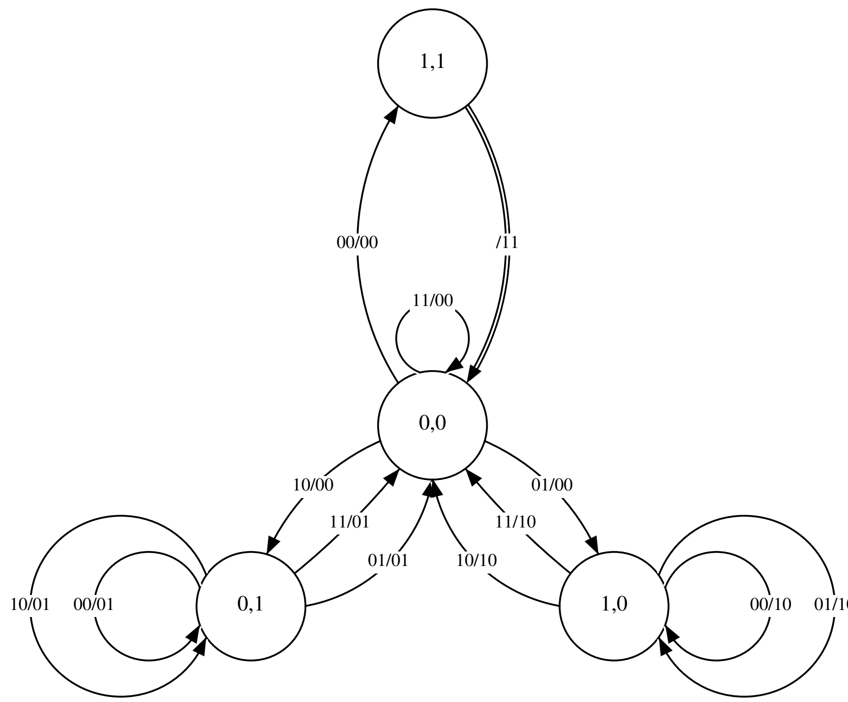

Figure 3.4: IO Automaton for RS Flip Flop

state (1, 1) is labelled with /11, which means that for any input received in state (1, 1), then the output $o1 = 1, o2 = 1$ will be produced when going to state (0, 0). Finally input labels r, s are replaced with negative integers $-1, -2$ and output labels are replaced with natural numbers $1, 2$, to form a semantic object.

3.3 Semantics of BKE

We now present a direct semantics, based on IO automata, for BKE. A related style of semantics was presented by Milner for a simple parallel programming language in [Mil90]. The notation used

First Delay-Nor Gate					Second Delay-Nor Gate					Conflict Removal
From	x x1 r	y y1 o2	o o1 o1	To	From	x x2 o1	y y2 s	o o2 o2	To	
0	0	1	0	0	0	0	0	0	1	deleted
0	0	0	0	1	0	0	0	0	1	included

Table 3.2: Example Calculation of Transitions for RS Flip Flop

follows the style of C.B. Jones' presentation of a denotational semantics for a Pascal-like programming language in [JS90]. We will present the semantics of BKE along with the abstract syntax, in a "bottom-up" fashion with the simplest constructs coming first. We ignore the static semantics of BKE in our discussion, so we simply assume that functions are defined before use and are called with the proper number of arguments, and so on.

3.3.1 Semantic Objects

BKE, like ELLA, is an expression-oriented language. Thus, the semantic objects must provide a uniform mechanism for naming the results of expressions. Furthermore, this naming mechanism must allow for the multiple-valued Boolean results of BKE, e.g. collateral of Boolean valued expressions. This is achieved by using a set of distinct internal names for outputs. For simplicity, we take this set as the positive integers. So, the IO automaton introduced and exemplified in the preceding sections is restricted as follows to give the basic semantic object *SemObj*.

$$\mathit{Name} = \mathit{Id} \cup \mathbb{Z} \backslash \{0\}$$

$$\mathit{SemObj} = \mathit{IOA}$$

where

$$\mathit{inv\text{-}SemObj} : \mathit{IOA} \to \mathbb{B}$$

$$\mathit{inv\text{-}SemObj}(\mathit{mk\text{-}IOA}(ss, i, o, ts, st)) \quad \triangleq \quad \exists n \in \mathbb{N} \cdot o = \{1, \ldots, n\}$$

3.3.1.1 Function objects

Function application will be achieved by taking products of automata whose input and/or output ports have been suitably renamed to effect connections. A "compiled" function, or *function object*, is an automaton, some of whose inputs are to be connected to the outputs of the arguments in a call of the function (see section 3.3.2.6).

We will represent a function object of arity n by a *SemObj* whose input set will include the negative integers $-1, \ldots, -n$ acting as formal argument names. To call the function, we attach the outputs of the actual arguments to the numbered inputs. Note that negative numbers are not legal program names. In this way, we can bind the actual arguments in a function application to the formal parameters without any danger of a name clash with the inputs.

3.3.2 Expressions

The first syntactic category we consider is the *Expression*.

$$\begin{aligned}\mathit{Expression} = {} & \mathit{Const} \cup \mathit{Var} \cup \mathit{Let} \cup \mathit{Row} \cup \mathit{Case} \cup \mathit{Funapp} \\ & \cup \mathit{Delays} \cup \mathit{Sample} \cup \mathit{BRAM} \cup \mathit{Not} \cup \mathit{Infix}\end{aligned}$$

The meaning of an expression will be given relative to an *environment*, which captures the function definitions in effect at the point where the expression occurs.

$$\mathit{Env} = \mathit{Name} \xrightarrow{m} \mathit{SemObj}$$

The denotation of an expression in a given environment will be a *SemObj*. The semantic function for an expression simply dispatches according to the expression type as follows:-

$MExpression : Expression \rightarrow Env \rightarrow SemObj$

$$
\begin{array}{lll}
MExpression[\![e]\!]\,\sigma & \triangleq & \text{cases } e \text{ of} \\
 & & \begin{array}{ll}
mk\text{-}Const(v) & \rightarrow MConst[\![e]\!] \\
mk\text{-}Var(v) & \rightarrow MVar[\![e]\!] \\
mk\text{-}Let(bd, e_1) & \rightarrow MLet[\![e]\!]\,\sigma \\
mk\text{-}Row(els) & \rightarrow MRow[\![els]\!]\,\sigma \\
mk\text{-}Case(rw, itms, else) & \rightarrow MCase[\![e]\!]\,\sigma \\
mk\text{-}Funapp(f, args) & \rightarrow MFunapp[\![e]\!]\,\sigma \\
mk\text{-}IDelay(e_1, v, n) & \rightarrow MIDelay[\![e]\!]\,\sigma \\
mk\text{-}Delay(e_1, v_1, n_1, v_2, n_2) & \rightarrow MDelay[\![e]\!]\,\sigma \\
mk\text{-}Sample(e_1, v, n_1, n_2) & \rightarrow MSample[\![e]\!]\,\sigma \\
mk\text{-}BRAM(e_1, n, v_1, e_2, e_3, e_4) & \rightarrow MBRAM[\![e]\!]\,\sigma \\
mk\text{-}Not(e_1) & \rightarrow MNot[\![e]\!]\,\sigma \\
mk\text{-}Infix(e_1, op, e_2) & \rightarrow MInfix[\![e]\!]\,\sigma
\end{array} \\
 & & \text{end}
\end{array}
$$

3.3.2.1 Constants

A constant denotes a one-state automaton whose purpose is to produce the desired constant value whenever it is required. The automaton has a single state, no inputs, one output, and one transition which outputs the constant value.

$Const :: Val$

$MConst : Const \rightarrow SemObj$

$$MConst[\![mk\text{-}Const(c)]\!] \triangleq const\text{-}machine(c)$$

$const\text{-}machine : Val \rightarrow SemObj$

$$const\text{-}machine(c) \triangleq mk\text{-}IOA(\{0\}, \{\,\}, \{1\}, \{mk\text{-}Trans(0, \{\,\}, \{1 \mapsto c\}, 0)\}, \{0\})$$

3.3.2.2 Variables

A variable reference denotes a one-state machine whose transitions simply copy the (single) input value to the output. The sole purpose of such a machine is to record the variable's name in the input set.

$Var :: Id$

$MVar : Var \rightarrow SemObj$

$$MVar[\![mk\text{-}Var(x)]\!] \triangleq var\text{-}machine(x)$$

$var\text{-}machine : Id \rightarrow SemObj$

$$
\begin{array}{l}
var\text{-}machine(x) \triangleq \\
\quad \text{let } ts = \{mk\text{-}Trans(0, \{x \mapsto v\}, \{1 \mapsto v\}, 0) \mid v \in Val\} \text{ in} \\
\quad mk\text{-}IOA(\{0\}, \{x\}, \{1\}, ts, \{0\})
\end{array}
$$

3.3.2.3 Recursive Let

The *Let* construct in BKE consists of a *Binding* and an *Expression*, which may refer to the bound variables of the *Binding*.

To evaluate a *Let*, we evaluate the *Binding* and the *Expression* independently, form the causal product of the results, then restrict to the outputs of the given expression.

$$
\begin{array}{lrl}
Let :: & eqns & : Binding \\
 & return & : Expression
\end{array}
$$

$MLet : Let \to Env \to SemObj$

$$
\begin{array}{l}
MLet [\![mk\text{-}Let(eqns, ret)]\!] \sigma \quad \triangleq \\
\quad \mathsf{let}\ expr\text{-}obj = MExpression [\![ret]\!] \sigma \ \mathsf{in} \\
\quad \mathsf{let}\ outs = outputs(expr\text{-}obj) \ \mathsf{in} \\
\quad restrict(causal\text{-}product(Mbinding [\![eqns]\!] \sigma, expr\text{-}obj), outs)
\end{array}
$$

Bindings A *Binding* in BKE consists of a non-empty sequence of equations either of the form $id = expr$, or $id_1, \ldots, id_n = expr$, where the identifiers on the left hand sides must all be distinct, though they may also appear in the expressions on the right hand sides. The meaning of a binding is a general *IOA* formed by evaluating each of the right hand sides, renaming the outputs to the left hand sides and taking the causal product of the resulting sequence of automata. We can think of this process as "solving" a set of simultaneous equations, since the only transitions permitted in the product are those whose input/output values satisfy the equations.

A *BindList* is a sequence of pairs which consist of a sequence of *Name* together with an *Expression* or an *SemObj*. The names must all be distinct.

$$
\begin{array}{lrl}
BindElem :: & lhs & : Name^{+} \\
 & rhs & : Expression \cup SemObj
\end{array}
$$

$BindList = BindElem^{+}$

$inv\text{-}BindList : BindElem^{+} \to \mathbb{B}$

$inv\text{-}BindList(bl) \quad \triangleq \quad \forall i, j \cdot i \neq j \ \Rightarrow \ (\mathsf{elems}\ lhs(bl(i)) \cap \mathsf{elems}\ lhs(bl(j))) = \{\,\}$

We represent a list of equations in BKE by a *Binding*. A *Binding* is a *BindList* whose left hand sides are all program identifiers and whose right hand sides are all expressions.

$Binding :: BindList$

$inv\text{-}Binding : BindList \to \mathbb{B}$

$inv\text{-}Binding(mk\text{-}Binding(l)) \quad \triangleq \quad \forall x \in \mathsf{elems}\ l \cdot rhs(x) \in Expression$

$MBinding : Binding \to Env \to IOA$

$MBinding [\![mk\text{-}Binding(bdl)]\!] \sigma \quad \triangleq \quad bind(eval\text{-}rhs(bdl, \sigma))$

$eval\text{-}rhs : BindList \times Env \rightarrow BindList$

$eval\text{-}rhs(bdl, \sigma) \quad \triangleq$

cases bdl of

$[mk\text{-}BindElem(x, e)] \quad \rightarrow [mk\text{-}BindElem(x, MExpression[\![e]\!]\sigma)]$

$[mk\text{-}BindElem(x, e)] \frown l \rightarrow mk\text{-}BindElem(x, MExpression[\![e]\!]\sigma) \frown eval\text{-}rhs(l, \sigma)$

end

pre $\forall x \in \mathsf{elems}\, bdl \cdot rhs(x) \in Expression$

Since the binding operation is also used in section 3.3.2.6, we need to distinguish between *Bindings* and *evaluated* binding lists, in which the expressions have already been replaced by automata.

$EvBindList = BindList$

where

$inv\text{-}EvBindList : BindElem^{+} \rightarrow \mathbb{B}$

$inv\text{-}EvBindList(l) \quad \triangleq \quad \forall x \in \mathsf{elems}\, l \cdot rhs(x) \in SemObj$

$bind : EvBindList \rightarrow IOA$

$bind(bdlist) \quad \triangleq$

cases $bdlist$ of

$[mk\text{-}BindElem(x, a)] \quad \rightarrow rename(a, x)$

$l \frown [mk\text{-}BindElem(x, a)] \rightarrow causal\text{-}prod(bind(l), rename(a, x))$

end

The function *rename* renames the outputs of an IOA a from $outputs(a)$ to the names contained in a list xl. It does this by constructing an IOA *copy* which simply copies inputs named $outputs(a)$ to outputs named $\mathsf{elems}\, xl$, and then takes the causal product of a and *copy*. Finally, the outputs of the IO automaton are restricted to those having names in $\mathsf{elems}\, xl$:

$rename : SemObj \times Name^{+} \rightarrow IOA$

$rename(a, xl) \quad \triangleq$

let $n = \mathsf{len}\, xl$ in

let $outs = [1, \ldots, n]$ in

let $new\text{-}outs = \mathsf{elems}\, xl$ in

let $copy = copy\text{-}machine(outs, xl)$ in

$restrict(causal\text{-}product(a, copy), new\text{-}outs)$

pre $outputs(a) = \mathsf{inds}\, xl \wedge \mathsf{card}\,(\mathsf{elems}\, xl) = \mathsf{len}\, xl$

When performing *rename* a copying IOA will be needed which copies inputs with name-list il to outputs with name-list ol. The resulting IOA is *not* a semantic object:

$$copy\text{-}machine : Id^* \times Id^* \rightarrow IOA$$

$$\begin{array}{l} copy\text{-}machine(il, ol) \quad \triangleq \\ \quad \mathsf{let}\ n = \mathsf{len}\ il\ \ \mathsf{in} \\ \quad \mathsf{let}\ ts = \{mk\text{-}Trans(0, \{il(1) \mapsto v_1, \ldots, il(n) \mapsto v_n\}, \\ \qquad\qquad\qquad \{ol(1) \mapsto v_1, \ldots ol(n) \mapsto v_n\}, 0) \mid \\ \qquad\qquad\qquad\quad v_i \in Val, i \in \{1, \ldots, n\}\} \\ \qquad \mathsf{in} \\ \quad mk\text{-}IOA(\{0\}, \mathsf{elems}\ il, \mathsf{elems}\ ol, ts, \{0\}) \end{array}$$

$$\mathsf{pre}\ \mathsf{len}\ il = \mathsf{len}\ ol$$

3.3.2.4 Rows

A row expression is denoted by a *SemObj* that has multiple outputs, for example $(e_1, \ldots, e_n)$ will be denoted by an automaton that produces n outputs, named 1 up to n. The semantic object is constructed by considering the row as a binding of the form $[(1, e_1), \ldots, (n, e_n)]$ and applying the meaning function for bindings. The function *number-outputs* creates the binding from the sequence of expressions representing the row.

$$Row = Expression^+$$

$$MRow : Row \rightarrow Env \rightarrow SemObj$$

$$MRow[\![els]\!]\sigma \quad \triangleq \quad MBinding(number\text{-}outputs(els))\sigma$$

$$number\text{-}outputs : Expression^+ \rightarrow Binding$$

$$\begin{array}{l} number\text{-}outputs(els) \quad \triangleq \\ \quad \mathsf{cases}\ els\ \ \mathsf{of} \\ \quad [\,] \qquad\qquad \rightarrow [\,] \\ \quad el \frown [expr] \rightarrow [mk\text{-}BindElem([\mathsf{len}\ els], expr)] \frown number\text{-}outputs(el) \\ \quad \mathsf{end} \end{array}$$

3.3.2.5 Case expressions

Before giving the formal semantics of the *case* statement, we outline the intuition behind the definition. Suppose we have a *case* statement of the form

```
case (u1,...,uk) of
  (c11,...,c1k)  : e1
  (c21,...,c2k)  : e2
    ......
  (cn1,...,cnk)  : en
    else
              e
end
```

where the selector expression is a row of width k and there are n cases. Suppose also that each of the expressions ei evaluates to a row of width s. The selector expression is represented by an automaton *sel-ioa* with k outputs, of the form shown in figure 3.5. The number of inputs of *sel-ioa* will depend on the number of free variables in the expressions ui and is not relevant to this discussion.

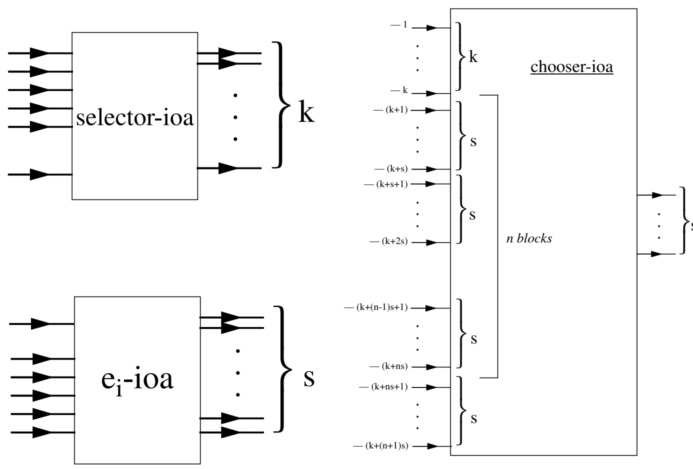

Figure 3.5: Case automaton components

Each of the eis is represented by an automaton with s outputs. The case mechanism is effected by connecting these automata to an automaton *chooser-ioa* which acts in the rôle of a multiplexor. This automaton, whose form is shown in figure 3.5, has $s - +(n+1)s$ inputs, partitioned as follows. The first k are connected to the output of *sel-ioa*, to give the value of the selector expression. The next $n * s$ inputs are connected, in blocks of s, to the outputs from the automata corresponding to the ei's. The final block of s inputs is connected to the outputs of the automaton for e. The automaton *chooser-ioa* has just one state and acts as follows: if the first k inputs have values matching (ci1,...,cik) , then the inputs from the ith block of s inputs are copied to the output.

$$ChoiceVal = Val \cup \{X\}$$

$$RowChoiceVal = ChoiceVal^{+}$$

$$\begin{aligned} Case \;::\; selector &: Row \\ items &: CaseItem^{+} \\ else\text{-}row &: Row \end{aligned}$$

$$\begin{aligned} CaseItem \;::\; chooser &: RowChoiceVal \\ result &: Row \end{aligned}$$

$MCase : Case \rightarrow Env \rightarrow SemObj$

$$
\begin{array}{l}
MCase\,[\![mk\text{-}Case(selector, itms, else)]\!]\,\sigma \quad \triangleq \\
\quad \text{let } k = \text{len}\, selector \text{ in} \\
\quad \text{let } n = \text{len}\, itms \text{ in} \\
\quad \text{let } s = \text{len}\, result(\text{hd}\, itms) \text{ in} \\
\quad \text{let } sel\text{-}ioa = rename(MRow\,[\![selector]\!]\,\sigma, outseq(0, 0, k)) \text{ in} \\
\quad \text{let } choose\text{-}ioa = MChList(get\text{-}choosers(itms)) \text{ in} \\
\quad \text{let } choices\text{-}ioa = iter\text{-}c\text{-}p(get\text{-}choices\text{-}rename(k, s, itms, \sigma)) \text{ in} \\
\quad \text{let } else\text{-}ioa = rename(MExpression\,[\![else]\!]\,\sigma, outseq(k, n, s)) \text{ in} \\
\quad \text{let } ioa\text{-}list = [sel\text{-}ioa, choose\text{-}ioa, choices\text{-}ioa, else\text{-}ioa] \text{ in} \\
\quad restrict(iter\text{-}c\text{-}p(ioa\text{-}list), \{1, \ldots, s\})
\end{array}
$$

MCase builds its result by taking the causal product of 4 automata. The first two of these are the automata *selector-ioa* and *chooser-ioa*, described above. The other two are *choices-ioa*, which is constructed by taking the causal product of the ei's with their outputs renamed to connect to the appropriate inputs of *chooser-ioa*, and *else-ioa*, the automaton corresponding to the action for the else clause, with its outputs suitably renamed.

$MChList : RowChoiceVal^{+} \rightarrow IOA$

$$
\begin{array}{l}
MChList(rch\text{-}list) \quad \triangleq \\
\quad \text{let } k = \text{len}\, chooser(\text{hd}\, rch\text{-}list) \text{ in} \\
\quad \text{let } n = \text{len}\, rch\text{-}list \text{ in} \\
\quad \text{let } s = \text{len}\, result(\text{hd}\, rch\text{-}list) \text{ in} \\
\quad \text{let } input\text{-}set = \{-(k + (n + 1)s), \ldots, -1\} \text{ in} \\
\quad \text{let } output\text{-}set = \{1, \ldots, s\} \text{ in} \\
\quad \text{let } all\text{-}alt\text{-}maps = \{-s, \ldots, -1\} \xrightarrow{m} Val \text{ in} \\
\quad \text{let } all\text{-}chsr\text{-}maps = \{-k, \ldots, -1\} \xrightarrow{m} Val \text{ in} \\
\quad \text{let } choices\text{-}trans = \\
\qquad \bigcup \{build\text{-}transitions(mapify(rch\text{-}list[i], -1), \\
\qquad\qquad\qquad all\text{-}alt\text{-}maps, k + (i - 1)s) \mid i \in \text{inds}\, rch\text{-}list\} \\
\qquad \text{in} \\
\quad \text{let } choice\text{-}inp\text{-}maps = \{ivals(tr) \mid tr \in choices\text{-}trans\} \text{ in} \\
\quad \text{let } else\text{-}maps = \\
\qquad \{m \in all\text{-}chsr\text{-}maps \mid \neg\, \exists i \in choice\text{-}inp\text{-}maps \cdot i \subseteq m\} \text{ in} \\
\quad \text{let } else\text{-}trans = \\
\qquad \bigcup \{build\text{-}transitions(m, all\text{-}alt\text{-}maps, k + n \times s) \mid \\
\qquad\qquad\qquad\qquad\qquad\qquad m \in else\text{-}maps\} \text{ in} \\
\quad mk\text{-}IOA(\{0\}, input\text{-}set, output\text{-}set, choices\text{-}trans \cup else\text{-}trans, \{0\})
\end{array}
$$

The function *MChList*, used to build *chooser-ioa* is the core of the construction. The rôle of *chooser-ioa* is, on the basis of the values its first k inputs, to copy the appropriate block of s inputs to its outputs. The function *build-transitions* is used to build a set of transitions corresponding to a particular choice, with inputs constructed from *chooser-in-map* and a set of possible inputs, *in-maps*, from a block of s inputs, which are just copied across to the output.

It might appear that a large number of internal signals are generated in this construction; in fact they all disappear from the final automaton, whose inputs correspond to the free variables in the expression and output width is that of one of the choice actions.

$build\text{-}transitions : (Name \xrightarrow{m} ChoiceVal) \times (\mathbb{Z} \xrightarrow{m} Val)\text{-set} \times \mathbb{N} \to Trans\text{-set}$

$build\text{-}transitions(chooser\text{-}in\text{-}map, in\text{-}maps, args\text{-}shift) \quad \triangleq$

$\quad \{mk\text{-}Trans(0, chooser\text{-}in\text{-}map \cup (m \circ shift(args\text{-}shift)), m \circ reflect, 0) \mid$

$\qquad m \in in\text{-}maps\}$

$mapify : RowChoiceVal \times \{-1, 1\} \to (\mathbb{Z} \xrightarrow{m} Val)$

$mapify(rcv, porm) \quad \triangleq \quad \{porm \times i \mapsto rcv[i] \mid i \in \text{inds}\, rcv \wedge rcv[i] \neq X\}$

$shift : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$

$shift(sh)(n) \quad \triangleq \quad n - sh$

$reflect : \mathbb{Z} \to \mathbb{Z}$

$reflect(n) \quad \triangleq \quad -n$

$iter\text{-}c\text{-}p : IOA^{+} \to IOA$

$iter\text{-}c\text{-}p(ioal) \quad \triangleq$

cases $ioal$ of

$[ioa] \to ioa$

$[ioa] \frown rest \to causal\text{-}product(ioa, iter\text{-}c\text{-}p(rest))$

end

$get\text{-}choosers : CaseItem^{*} \to RowChoiceVal^{*}$

$get\text{-}choosers(ci\text{-}list) \quad \triangleq$

cases $ci\text{-}list$ of

$[\,] \to [\,]$

$[ci] \frown rest \to [chooser(ci)] \frown get\text{-}choosers(rest)$

end

$get\text{-}choices\text{-}rename: (init\text{-}shift: \mathbb{N}, out\text{-}len: \mathbb{N}, ci\text{-}list: CaseItem^{+}, \sigma: Env) \to ioa\text{-}list: IOA^{+}$

post (len $ioa\text{-}list$ = len $ci\text{-}list$) $\wedge$

$\quad \forall i \in$ inds $ci\text{-}list \cdot$

$\qquad ioa\text{-}list[i] = rename(MExpression[\![result(ci\text{-}list[i])]\!]\,\sigma,$

$\qquad\qquad outseq(init\text{-}shift, i-1, out\text{-}len))$

$outseq: (first: \mathbb{N}, blocks: \mathbb{N}, length: \mathbb{N}1) \to os: \mathbb{Z}^{*}$

post len $os = length \wedge$

$\quad \forall i \in$ inds $os \cdot$

$\qquad os[i] = -(first + blocks \times length + i)$

An example automaton construction including a case statement is given in section 4.3.

3.3.2.6 Function application

To apply a function to a sequence of arguments, we look up the function definition in the environment. We will assume that the number of arguments in the call matches the number specified by the function object. We evaluate the arguments into a list of automata (using *eval-list*). We then apply the function object to the evaluated arguments by renaming their outputs to the names $-1, \ldots, -n$ and taking the product with the body of the function object (this is what the function *bind* does—see section 3.3.2.3). The function *number-autlist* takes a list $[a_1 \ldots a_n]$ and returns the binding list $[(-1, a_1) \ldots (-n, a_n)]$. Finally we restrict the output of our automaton to the outputs of the function.

$$\begin{array}{lll} \mathit{Funapp} :: & \mathit{func} & : \mathit{Name} \\ & \mathit{args} & : \mathit{Expression}^* \end{array}$$

$$\mathit{MFunapp} : \mathit{Funapp} \to \mathit{Env} \to \mathit{SemObj}$$

$$\mathit{MFunapp}[\![\mathit{mk\text{-}Funapp}(f, \mathit{args})]\!]\sigma \quad \triangleq \quad \mathit{apply}(\sigma(f), \mathit{eval\text{-}list}(\mathit{args}, \sigma))$$

$$\mathit{eval\text{-}list} : \mathit{Expression}^* \times \mathit{Env} \to \mathit{Env}$$

$$\begin{array}{l} \mathit{eval\text{-}list}(\mathit{exps}, \sigma) \quad \triangleq \\ \quad \textsf{cases } \mathit{exps} \textsf{ of} \\ \quad [\,] \qquad\quad \to [\,] \\ \quad [e] \frown \mathit{rest} \to [\mathit{MExpression}[\![e]\!]\sigma] \frown \mathit{eval\text{-}list}(\mathit{rest}, \sigma) \\ \quad \textsf{end} \end{array}$$

$$\mathit{apply} : \mathit{SemObj} \times \mathit{SemObj}^* \to \mathit{SemObj}$$

$$\begin{array}{l} \mathit{apply}(\mathit{fo}, \mathit{alist}) \quad \triangleq \\ \quad \textsf{let } \mathit{args} = \mathit{bind}(\mathit{number\text{-}autlist}(\mathit{alist})) \textsf{ in} \\ \quad \textsf{let } \mathit{cp} = \mathit{causal\text{-}product}(\mathit{fo}, \mathit{args}) \textsf{ in} \\ \quad \mathit{restrict}(\mathit{cp}, \mathit{outputs}(\mathit{fo})) \end{array}$$

$$\mathit{number\text{-}autlist} : \mathit{SemObj}^* \to \mathit{BindElem}^*$$

$$\begin{array}{l} \mathit{number\text{-}autlist}(l) \quad \triangleq \\ \quad \textsf{cases } l \textsf{ of} \\ \quad [\,] \qquad \to [\,] \\ \quad l_1 \frown [a] \to \mathit{number\text{-}autlist}(l_1) \frown [\mathit{mk\text{-}BindElem}([-\textsf{len}\, l], a)] \\ \quad \textsf{end} \end{array}$$

$$\mathit{restrict} : \mathit{IOA} \times \mathit{Name}\textsf{-set} \to \mathit{SemObj}$$

$$\begin{array}{l} \mathit{restrict}(a, \mathit{xs}) \quad \triangleq \\ \quad \textsf{let } \mathit{mk\text{-}IOA}(s, i, o, \mathit{ts}, \mathit{st}) = a \textsf{ in} \\ \quad \textsf{let } \mathit{new\text{-}ts} = \{\mu(t, \mathit{ovals} \mapsto \mathit{xs} \lhd \mathit{ovals}(t)) \mid t \in \mathit{ts}(a)\} \textsf{ in} \\ \quad \mu(a, \mathit{transitions} \mapsto \mathit{new\text{-}ts}, \mathit{outputs} \mapsto \mathit{xs}) \end{array}$$

$$\textsf{pre } \mathit{xs} \subseteq \mathit{outputs}(a)$$

3.3.2.7 Delays

A delay operation in BKE applies a predefined IO automaton to the result of its expression operand. BKE contains two different delay types called ambiguity delay *delay* and inertial delay *idelay*[HWM90]. A different IO automaton is constructed for each delay duration and initial output value. In each case the same basic technique will be used to construct the IO automata. Each state label consists of a string of values which encode the input history required to compute the next output and state for each input. Having constructed the IO automaton, the special string state labelling can be replaced with the usual natural number labelling, since the labelling itself no longer has further significance.

3.3.2.7.1 Inertial delay An inertial delay *idelay*(*arg, const, int*) produces an initial output value *const*, and then the input values *arg* are reproduced on the output after a delay of *int* time units. However if the input is stable for fewer than *int* time units then the input change is ignored.

For example, if *arg* = *In*, *const* = 0 and *int* = 3 then *Out* = *idelay*(*In*, 0, 3):

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
In	1	1	0	0	0	1	1	1	0	0	1	0	0	0	0
Out	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0

The semantics for inertial delay is given by:

$$Delays = idelay \mid delay$$

$$\begin{aligned} idelay ::\ \ & arg : Expression \\ & const : RowOfVal \\ & int : \mathbb{N} \end{aligned}$$

$$MIdelay : idelay \times Env \to SemObj$$

$$\begin{aligned} & MIdelay[\![mk\text{-}Idelay(e, c, i)]\!]\sigma \ \ \triangleq \\ & \quad \mathsf{let}\ exp\text{-}obj = MExpression[\![e]\!]\sigma\ \ \mathsf{in} \\ & \quad \mathsf{let}\ w = \mathsf{card}\ outputs(exp\text{-}obj)\ \ \mathsf{in} \\ & \quad apply(mk\text{-}idel(c, i, w), [exp\text{-}obj]) \end{aligned}$$

where *mk-idel*(*c, i, w*) is a *SemObj*:

$$mk\text{-}idel : RowOfVal \times \mathbb{N} \times Nat_1 \to SemObj$$

$$\begin{aligned} & mk\text{-}idel(c, i, w) \ \ \triangleq \\ & \quad \mathsf{let}\ val\text{-}tuples = Val^w\ \ \mathsf{in} \\ & \quad \mathsf{let}\ states = \{s \frown t \in val\text{-}tuples^* \mid \mathsf{len}\, s \frown t = i, \\ & \qquad\qquad \mathsf{elems}\, s = \{a_1\}, \mathsf{elems}\, t = \{a_2\}, \\ & \qquad\qquad a_1 \in val\text{-}tuples, a_2 \in val\text{-}tuples\}\ \ \mathsf{in} \\ & \quad \mathsf{let}\ inits = \{s \in states \mid \mathsf{elems}\, s = \{c\}\}\ \ \mathsf{in} \\ & \quad \mathsf{let}\ ts = \{mk\text{-}Trans(p, mapify(v, -1), mapify(\mathsf{hd}\, p, 1), q) \mid \\ & \qquad\qquad p \in states, v \in val\text{-}tuples, q = choose\text{-}state(p, v)\}\ \ \mathsf{in} \\ & \quad mk\text{-}IOA(states, \{-w, \ldots, -1\}, \{1, \ldots, w\}, ts, inits) \end{aligned}$$

$$\mathsf{pre}\ \mathsf{len}\, c = w$$

The function *choose-state* decides whether the input change should be ignored or not, by calculating the next state:

$$choose\text{-}state : RowOfVal^{+} \times RowOfVal \rightarrow RowOfVal^{+}$$

$$\begin{aligned} choose\text{-}state(s, a) \;\triangleq\;& \mathsf{if}\ s(\mathsf{len}\, s) = a \\ & \mathsf{then}\ \mathsf{tl}\, s \frown [a] \\ & \mathsf{else}\ \mathsf{let}\ n = (\mathsf{len}\, s) - 1\ \ \mathsf{in} \\ & \quad rep\text{-}seq(n, \mathsf{hd}\, s) \frown [a] \end{aligned}$$

$$\begin{aligned} rep\text{-}seq(n, x) \;\triangleq\;& \mathsf{if}\ n <= 0 \\ & \mathsf{then}\ [\,] \\ & \mathsf{else}\ [x] \frown rep\text{-}seq(n-1, x) \end{aligned}$$

For example, the IO automaton for the three-unit inertial delay $idelay(In, 0, 3)$, with initial output value $const = 0$ and $i = 3$ is shown in Figure 3.6. The initial state is indicated by a double circle.

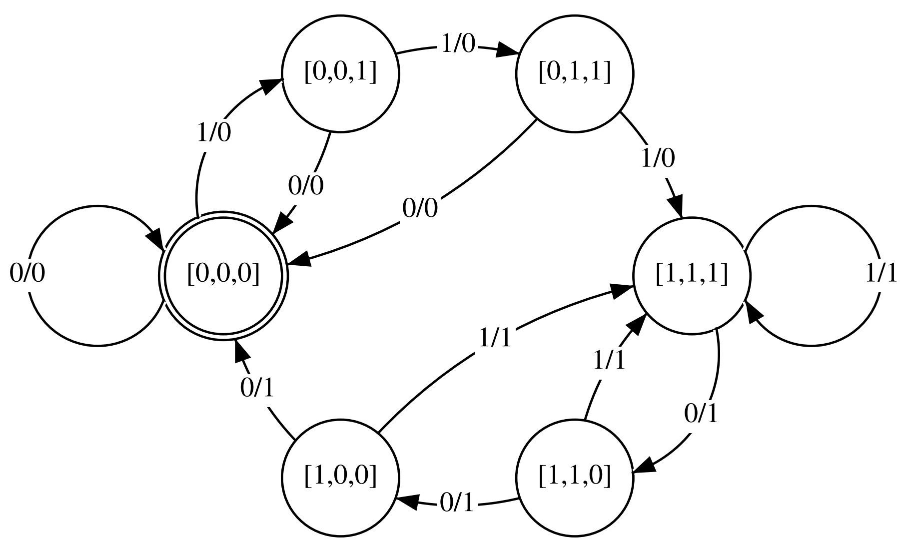

Figure 3.6: Three-Unit Inertial Delay Example IOA

An interesting characteristic of inertial delay is that even though it only requires a finite input history to be stored, in order to calculate its next output value and transition, the full trace history must be available to calculate the last output value, which is of course necessary to calculate the next output.

3.3.2.7.2 Ambiguity Delay Ambiguity delay requires four parameters to describe it: two constant values $const_1,\ const_2 \in Val$ and two delay intervals $int_1,\ int_2$.

In [HWM90], the following informal description for ambiguity delay is given, when considering the function $delay(const_1, int_1, const_2, int_2)$:

The initial value of this function is '$const_1$'. If there exists a sequence of int_2 equal input values occurring between the current time and int_1+int_2-1 units of time ago then the output of the function is the input value int_2 units of time ago, otherwise the output is '$const_2$'.

The construction of the IO automaton follows this description. Sequences of the last $int_1 + int_2 - 1$ inputs are stored as state labels; these are then searched for subsequences of int_2 equal input values to determine the next output values.

Assume for the purposes of illustration only, that two distinct constants (x, y) are used to represent the values in the value set *Val* assigned to the constants $const_1$ and $const_2$. For example, if $int_1 = 2$ and $int_2 = 5$ then $Out = delay(In, x, 2, y, 5)$ will have the following effect:

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
In	0	0	0	1	0	0	0	0	0	1	1	1	1	1	1	1
Out	x	x	y	y	y	y	y	y	y	0	0	y	y	y	1	1

The semantics for ambiguity delay is given by:

$$\begin{aligned} delay :: \quad arg &: Expression \\ const_1 &: RowOfVal \\ int_1 &: \mathbb{N} \\ const_2 &: RowOfVal \\ int_2 &: \mathbb{N} \end{aligned}$$

$$MDelay : delay \times Env \to SemObj$$

$$\begin{aligned} & MDelay[\![mk\text{-}Delay(e, c_1, i_1, c_2, i_2)]\!]\sigma \quad \triangleq \\ & \quad \text{let } exp\text{-}obj = MExpression[\![e]\!]\sigma \text{ in} \\ & \quad \text{let } w = \text{card } outputs(exp\text{-}obj) \text{ in} \\ & \quad apply(mk\text{-}del(c_1, i_1, c_2, i_2, w), exp\text{-}obj) \end{aligned}$$

where $mk\text{-}del(c_1, i_1, c_2, i_2, w)$ is a *SemObj*:

$$mk\text{-}del : RowOfVal \times \mathbb{N} \times RowOfVal \times \mathbb{N} \times Nat_1 \to SemObj$$

$$\begin{aligned} & mk\text{-}del(c_1, i_1, c_2, i_2, w) \quad \triangleq \\ & \quad \text{let } states = \{s \in RowOfVal^* \mid \text{len } s = i_1 + i_2 - 1\} \text{ in} \\ & \quad \text{let } inits = \{s \in states \mid \text{elems } s = \{c_1\}\} \text{ in} \\ & \quad \text{let } ts = \{mk\text{-}Trans(p, mapify(v, -1), mapify(q, 1), \text{tl } p \frown [v]) \mid \\ & \qquad\qquad p \in states, q = next\text{-}output(p, i_1, i_2, c_2), v \in RowOfVal\} \text{ in} \\ & \quad mk\text{-}IOA(states, \{-w, \ldots, -1\}, \{1, \ldots, w\}, ts, inits) \end{aligned}$$

$$\text{pre len } c_1 = w \wedge \text{len } c_2 = w$$

The function *next-output* determines the next output value:

$$next\text{-}output : RowOfVal^* \times \mathbb{N} \times \mathbb{N} \times RowOfVal \to RowOfVal$$

$$next\text{-}output(s, m, n, c) \quad \triangleq \quad \text{if } subsequence(s, n) \text{ then } s(m) \text{ else } c$$

The function *subsequence*(s, n) checks whether the sequence s contains a sub-sequence of identical values of length $\geq n$:

$$subsequence : RowOfVal^* \times \mathbb{N} \rightarrow \mathbb{B}$$

$$subsequence(s, n) \quad \triangleq \quad \exists s_1, s_2 \in RowOfVal^*, v \in RowOfVal \cdot s = s_1 \frown rep\text{-}seq(n, v) \frown s_2$$

As before, the string labelling for each state records the finite input history so that the next transition can be determined, depending on the current input value. For example, the IO automaton for the two-unit ambiguity delay *delay*$(In, 0, 2, 0, 2)$, with constant output values $const_1 = 0$, $const_2 = 0$, and delay constants $i_1 = 2$, $i_2 = 2$, is shown in Figure 3.7. Again the initial state is indicated by a double circle.

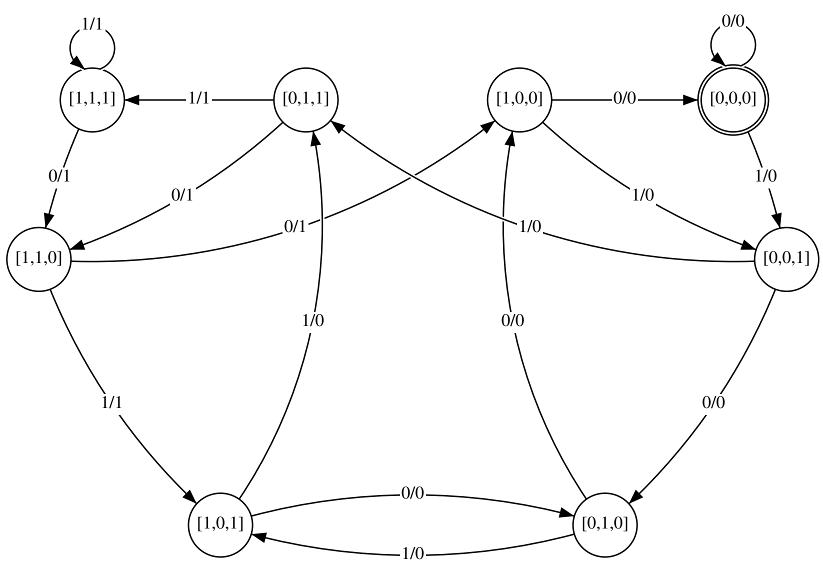

Figure 3.7: Two-Unit Ambiguity Delay Example IOA

3.3.2.8 Sample

A sample operation is included in Core ELLA, and will be part of future versions of ELLA, so that elementary retiming operations can be applied to design modules. One part of the design can then be considered to be running at a faster or slower clock rate, in terms of the ELLA time-unit, than other parts in the same interacting environment. The operation *Sample*(*expr, const, skew, int*) selects the input value given by *expr* at regular intervals *int* and outputs this value for the duration of the interval. The start of the sampling process can also be skewed by an amount *skew*, where an initial value *const* is produced until the sampling process begins at time *skew*. Fresh input samples will be taken at times $(skew + int)$, $(skew + 2 \times int)$, and so on. For example, if $expr = In$, $const = x$, $skew = 2$ and $int = 4$, then $Out = Sample(In, x, 2, 4)$ is given by:

Time	0	1	2	3	4	5	6	7	8	9	10	…
In	1	0	0	0	1	0	1	0	0	0	0	…
Out	x	x	0	0	0	0	1	1	1	1	0	…

The semantics for *Sample* is derived in a similar way to that of delays, by applying a semantic object, *Sampler* to the argument *expr*. The states are constructed as follows: there is an initial state, $-skew$, followed by a sequence of states $-skew+1, -skew+2, \ldots, -1$ where the output is *const* regardless of the value on the input. Then for each possible input vector v, a cycle of states and transitions is formed beginning and ending in state 0. The cycle has n transitions for a sampling interval n, The intermediate states are labelled (for construction purposes only) by the pair (v, i) where $i \in \{1, \ldots, n-1\}$ is the index position on the cycle. For each state (v, j) there is a single transition to state $(v, j+1)$, $j \in \{1, \ldots, n-2\}$. The transitions exiting from state 0 are both input- and output-labelled (v/v), whereas all other transitions in the cycle are just output-labelled ($/v$), because the input value does not affect the output value.

$$\begin{aligned} \mathit{Sample} :: \quad \mathit{expr} &: \mathit{Expression} \\ \mathit{const} &: \mathit{Val}^{+} \\ \mathit{skew} &: \mathbb{N} \\ \mathit{int} &: \mathbb{N}_1 \end{aligned}$$

$\mathit{MSample} : \mathit{Expression} \times \mathit{Val}^{+} \times \mathbb{N} \times \mathbb{N}_1 \rightarrow \mathit{SemObj}$

$\mathit{MSample}[\![\mathit{expr}, \mathit{const}, \mathit{skew}, \mathit{int}]\!]\sigma \quad \triangleq$

$\quad$ let $\mathit{exp\text{-}obj} = \mathit{MExpression}[\![\mathit{expr}]\!]\sigma$ in

$\quad$ let $\mathit{width} = \mathrm{card}\ \mathit{outputs}(\mathit{exp\text{-}obj})$ in

$\quad$ $\mathit{apply}(\mathit{Sampler}(\mathit{const}, \mathit{skew}, \mathit{int}, \mathit{width}), \mathit{exp\text{-}obj})$

$\mathit{Sampler} : \mathit{Val}^{+} \times \mathbb{N} \times \mathbb{N}_1 \times \mathbb{N}_1 \rightarrow \mathit{SemObj}$

$\mathit{Sampler}(c, s, n, w) \quad \triangleq$

$\quad$ let $\mathit{inits} = \{-s\}$ in

$\quad$ let $\mathit{skews} = \{-s+i \mid i \in \{1, \ldots, s\}\}$ in

$\quad$ let $\mathit{cyclers} = \{(v, i) \mid v \in \mathit{Val}^{w} \wedge i \in \{2, \ldots, n-1\}\}$ in

$\quad$ let $\mathit{states} = \mathit{inits} \cup \mathit{skews} \cup \mathit{cyclers}$ in

$\quad$ let $\mathit{skew\text{-}ts} = \{\mathit{mk\text{-}Trans}(-s+j, \{\,\}, \mathit{mapify}(c, 1), -s+j+1) \mid j \in \{0, \ldots, s-1\}\}$ in

$\quad$ let $\mathit{cycle\text{-}one\text{-}ts} = \{\mathit{mk\text{-}Trans}(0, \mathit{mapify}(v, -1), \mathit{mapify}(v, 1), 0) \mid v \in \mathit{Val}^{w} \wedge n = 1\}$ in

$\quad$ let $\mathit{cycle\text{-}first\text{-}ts} = \{\mathit{mk\text{-}Trans}(0, \mathit{mapify}(v, -1), \mathit{mapify}(v, 1), (v, 1)), \mid v \in \mathit{Val}^{w} \wedge n > 1\}$ in

$\quad$ let $\mathit{cycle\text{-}last\text{-}ts} = \{\mathit{mk\text{-}Trans}((v, n-1), \{\,\}, \mathit{mapify}(v, 1), 0) \mid v \in \mathit{Val}^{w} \wedge n > 1\}$ in

$\quad$ let $\mathit{cycle\text{-}ts} = \{\mathit{mk\text{-}Trans}((v, i), \{\,\}, \mathit{mapify}(v, 1), (v, i+1)) \mid v \in \mathit{Val}^{w} \wedge i \in \{1, \ldots, n-2\}\}$ in

$\quad$ let $\mathit{ts} = \mathit{skew\text{-}ts} \cup \mathit{cycle\text{-}one\text{-}ts} \cup \mathit{cycle\text{-}first\text{-}ts} \cup \mathit{cycle\text{-}last\text{-}ts} \cup \mathit{cycle\text{-}ts}$ in

$\quad$ $\mathit{mk\text{-}IOA}(\mathit{states}, \{-w, \ldots, -1\}, \{1, \ldots, w\}, \mathit{ts}, \mathit{inits})$

pre $\mathrm{len}\ c = w$

For example, the function object for the sample operation $\mathit{Sample}(\mathit{In}, 0, 2, 4)$, with $\mathit{Val} = \{0, 1\}$, is shown in Figure 3.8.

Having constructed the automaton, the special state labelling can be replaced with the usual natural number labelling, since the labelling itself no longer has further significance.

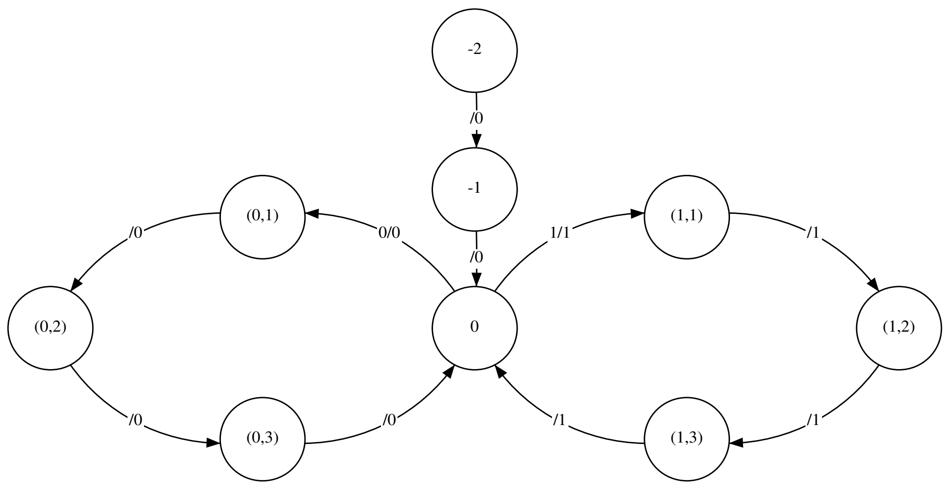

Figure 3.8: IO Automaton for Sampling Every 4 Steps with 2-unit Skew

3.3.2.9 RAM with Boolean vector addresses

A RAM operation in ELLA provides an efficient method for storing and retrieving data-values, behaving in a manner very similar to usual random access memory. In BKE, it will be implemented in a similar fashion to the delay and sample operations, by applying a predefined IO automaton to the result of its expression operand, a write and read address and a write enable. In this version, each address value will be given by a row of booleans, which will be converted to an integer representation before storing or accessing values in the RAM. Note that address values begin at 0.

The Boolean RAM operation *BRAM*(*arg, size, const, write, read, enable*) creates a storage for *size* values, which are each initialised to *const*. When *enable* = 1 then writing is enabled and the RAM is modified at the position given by *write*, changing its contents to *arg*; all other values remain the same. If *enable* = 0 then the RAM contents cannot be altered. The output from RAM is always the contents held at the read address given by *read*, after any writes have been performed. For example, for a RAM storing two one-bit vectors, *Out* = *BRAM*(*In,* 2*,* 0*, write, read, enable*) is given by:

Time	In	Write	Read	Enable	Out	Resulting RAM Contents
0	0	[1]	[0]	0	0	[0,0]
1	1	[1]	[1]	1	1	[0,1]
2	0	[0]	[0]	1	0	[0,1]
3	1	[0]	[0]	1	1	[1,1]
4	0	[1]	[0]	0	1	[1,1]
5	0	[1]	[0]	1	1	[1,0]
6	1	[0]	[1]	1	0	[1,0]
7	0	[0]	[1]	1	0	[0,0]

While the IO automaton is being constructed, each state label consists of a sequence of values, which represents the current contents of the RAM. As before, the special string state labelling can be replaced with the usual natural number labelling after construction, since the labelling itself no longer has further significance.

The semantics for BRAM is given by:

$$\begin{array}{lrl} BRAM :: & arg & : Expression \\ & size & : \mathbb{N}_1 \\ & const & : RowOfVal \\ & write & : Expression \\ & read & : Expression \\ & enable & : Expression \end{array}$$

$$MBRAM : BRAM \times Env \rightarrow SemObj$$

$$\begin{array}{l} MBRAM[\![mk\text{-}BRam(e, sz, c, w, r, en)]\!]\sigma \quad \triangleq \\ \quad \mathsf{let}\ exp\text{-}obj = MExpression[\![e]\!]\sigma\ \mathsf{in} \\ \quad \mathsf{let}\ write\text{-}obj = MExpression[\![w]\!]\sigma\ \mathsf{in} \\ \quad \mathsf{let}\ read\text{-}obj = MExpression[\![r]\!]\sigma\ \mathsf{in} \\ \quad \mathsf{let}\ enable\text{-}obj = MExpression[\![en]\!]\sigma\ \mathsf{in} \\ \quad \mathsf{let}\ wd = \mathsf{card}\ outputs(exp\text{-}obj)\ \mathsf{in} \\ \quad apply(make\text{-}bram(sz, c, wd), exp\text{-}obj, write\text{-}obj, read\text{-}obj, enable\text{-}obj) \end{array}$$

$$\mathsf{pre}\ \mathsf{card}\ outputs(write\text{-}obj) = \mathsf{card}\ outputs(read\text{-}obj)$$

where *make-bram*(sz, c, wd) is a *SemObj*:

$$make\text{-}bram : \mathbb{N}_1 \times RowOfVal \times \mathbb{N}_1 \rightarrow SemObj$$

$$\begin{array}{l} make\text{-}bram(sz, c, wd) \quad \triangleq \\ \quad \mathsf{let}\ k = \lceil \log_2(sz) \rceil\ \mathsf{in} \\ \quad \mathsf{let}\ Baddr = \{bl \in \mathbb{B}^+ \mid \mathsf{len}\ bl = k\}\ \mathsf{in} \\ \quad \mathsf{let}\ Iaddr = \{1, \ldots, sz\}\ \mathsf{in} \\ \quad \mathsf{let}\ states = \{s \mid \mathsf{elems}\ s = RowOfVal, \mathsf{len}\ s = sz\}\ \mathsf{in} \\ \quad \mathsf{let}\ inits = \{s \mid s \in states, \mathsf{elems}\ s = \{c\}\}\ \mathsf{in} \\ \quad \mathsf{let}\ ts = \{mk\text{-}Trans(p, mapify([v, wt, rd, en], -1), \\ \qquad\qquad\qquad mapify(p(radd), 1), \\ \qquad\qquad\qquad next\text{-}bram\text{-}state(p, v, wadd, enable)) \mid \\ \qquad\quad p \in states \wedge v \in RowOfVal \wedge \\ \qquad\quad wt, rd \in Baddr \wedge en \in \{0, 1\} \wedge \\ \qquad\quad wadd = row\text{-}to\text{-}int(wt) \wedge \\ \qquad\quad radd = row\text{-}to\text{-}int(rd) \wedge \\ \qquad\quad enable = row\text{-}to\text{-}int(en) \wedge \\ \qquad\quad wadd, radd \in Iaddr\}\ \mathsf{in} \\ \quad mk\text{-}IOA(states, \{-(wd + 2k + 1), \ldots, -1\}, \{1, \ldots, wd\}, ts, inits) \end{array}$$

$$\mathsf{pre}\ \mathsf{len}\ c = wd$$

where *next-bram-state* selects the next state:

$$next\text{-}bram\text{-}state : RowOfVal^+ \times RowOfVal \times \mathbb{N}_1 \times \mathbb{N} \rightarrow RowOfVal^+$$

$$next\text{-}bram\text{-}state(s, v, i, e) \quad \triangleq \quad \mathsf{if}\ e = 1\ \mathsf{then}\ s \dagger \{i \mapsto v\}\ \mathsf{else}\ s$$

The function *row-to-int* converts a row of booleans into its equivalent integer value in the obvious way. Note that for a row $x(1, \ldots, k)$, then the most significant bit is $x(1)$.

For example, the IO automaton for the RAM storing two single-bit boolean rows, *BRAM*(*In,* 2, 0*, write, read, enable*), with *size* = 2 and *const* = 0, is shown in Figure 3.9; for readability, 'don't care' values are indicated by '?'. The special state labels used only during construction of the automaton have also been shown, and the initial state is indicated by a double circle.

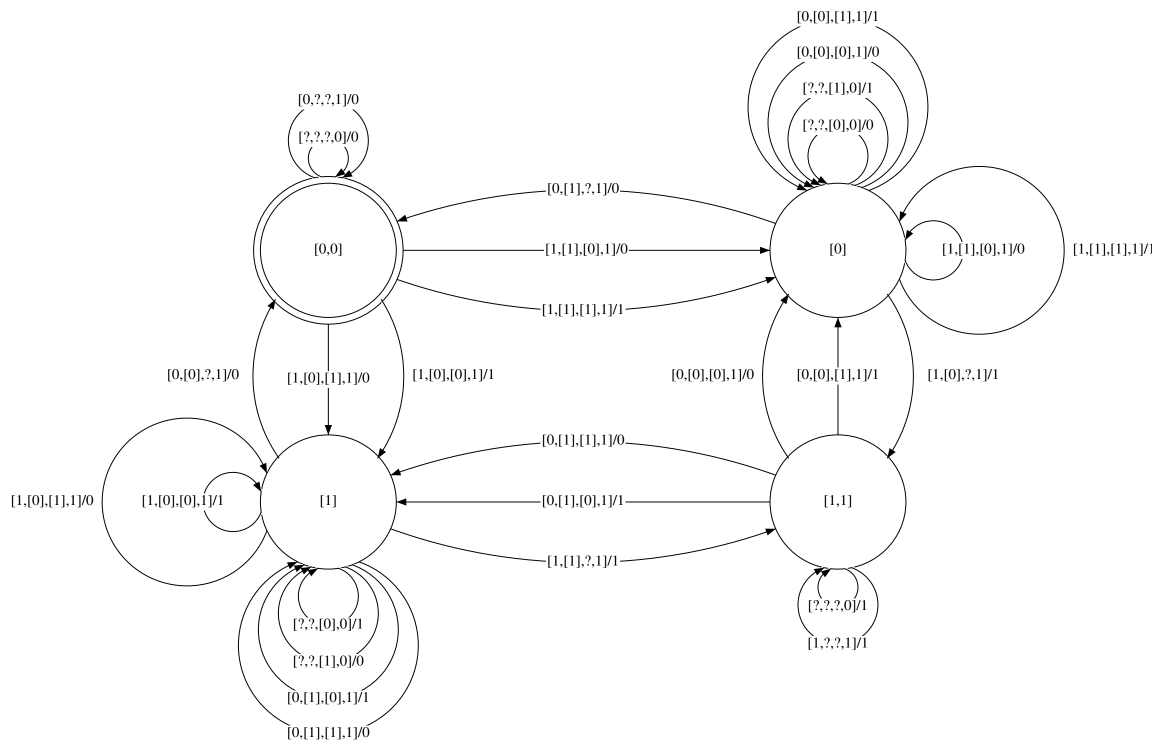

Figure 3.9: Two-Bit RAM Example IOA

3.3.2.10 BIOPS: Built-in Operators

Unary Operators BKE has one built-in unary operator, the negation operator. It is simply applied to the result of its operand expression.

$$Not \;::\; negated \;:\; Expression$$

$$MNot : Not \to Env \to SemObj$$

$$MNot[\![mk\text{-}Not(n)]\!]\,\sigma \quad \triangleq \quad apply(not\text{-}obj, MExpression[\![n]\!]\,\sigma)$$

where *not-obj* is a predefined *SemObj* given by

$$not\text{-}obj = mk\text{-}IOA(\{0\}, \{-1\}, \{1\}, ts, \{0\})$$

and

$$ts = \{mk\text{-}Trans(0, \{-1 \mapsto v\}, \{1 \mapsto \neg v\}, 0) \mid v \in Val\}$$

Binary operators Binary operations such as OR are treated in a similar way to negation, by applying predefined function objects.

$Binop$ = OR | AND

$$\begin{aligned} Infix \;::\; arg1 &: Expression \\ op &: Binop \\ arg2 &: Expression \end{aligned}$$

$MInfix : Infix \rightarrow Env \rightarrow SemObj$

$MInfix [\![mk\text{-}InFix(e_1, o, e_2)]\!] \sigma \quad \triangleq$
$\quad apply(MBinop [\![o]\!], [MExpression [\![e_1]\!] \sigma, MExpression [\![e_2]\!] \sigma])$

The function *MBinop* can be defined on a case-by-case basis for as many binary operators as we wish to include as primitives in the language. For example:

$$MBinop(\mathrm{OR}) = mk\text{-}IOA(\{0\}, \{-1, -2\}, \{1\}, ts, \{0\})$$

where

$$ts = \{mk\text{-}Trans(0, \{-1 \mapsto v_1, -2 \mapsto v_2\}, \{1 \mapsto v_1 \vee v_2\}, 0) \mid v_1, v_2 \in Val\}$$

and $\vee$ is the disjunction operator for the underlying data type *Val*.

3.3.3 Function declarations

To evaluate a function definition we first evaluate the function body in the given environment. We then rename the inputs corresponding to the formal parameters to the appropriate internal names, i.e. negative numbers. Finally, we bind the function name to a *SemObj* consisting of a count of the parameters and the automaton we have just constructed.

Note that if the expression defining the function body does not contain a free reference to some formal parameter, the resulting automaton will not have the corresponding name in its input set. When we apply the function, the result is correct since the missing inputs do not impose any conditions on the transition set.

The renaming mapping for the automaton is produced by the function *make-subst*, which has to take account of the fact that the formal parameters may not all appear in the input set. Note the precondition on the arguments to *number-inputs*, which ensures that the new numerical names do not conflict with existing input names.

$$\begin{aligned} FnDec \;::\; name &: Id \\ params &: Id^* \\ body &: Expression \end{aligned}$$

$MFnDec : FnDec \rightarrow Env \rightarrow Env$

$MFnDec [\![mk\text{-}FnDec(f, ps, b)]\!] \sigma \quad \triangleq$
$\quad \mathsf{let}\ obj\text{-}body = number\text{-}inputs(MExpression [\![b]\!] \sigma, ps)\ \mathsf{in}$
$\quad \sigma \dagger \{f \mapsto obj\text{-}body\}$

$number\text{-}inputs : IOA \times Id^* \rightarrow IOA$

$number\text{-}inputs(a, ids) \quad \triangleq$
  $\text{let } mk\text{-}IOA(ss, i, o, ts, start) = a \text{ in}$
  $\text{let } subst = make\text{-}subst(i, ids) \text{ in}$
  $\text{let } new\text{-}ts = \{\mu(t, ivals \mapsto ivals(t) \circ subst) \mid t \in ts\} \text{ in}$
  $mk\text{-}IOA(ss, \text{dom } subst, o, new\text{-}ts, start)$

$\text{pre } inputs(a) \subseteq Id$

$make\text{-}subst : Id\text{-set} \times Id^* \rightarrow (Name \xleftrightarrow{m} Name)$

$make\text{-}subst(iset, ilist) \quad \triangleq$
  $\text{let } inv = \{x \mapsto x \mid x \in iset\} \dagger \{ilist(k) \mapsto -k \mid ilist(k) \in iset \wedge k \in \text{inds } ilist\} \text{ in}$
  inv^{-1}

3.3.4 Closures

A program in BKE, or *Closure*, consists of a sequence of function declarations. The meaning of a *Closure* is thus taken as the function environment.

$Closure = FnDec^+$

$MClosure : Closure \rightarrow Env$

$MClosure[\![ds]\!] \quad \triangleq \quad MFnDecList[\![ds]\!]$

$MFnDecList : FnDec^* \rightarrow Env$

$$MFnDecList[\![ds]\!] \quad \triangleq \quad \begin{array}{l} \text{cases } ds \text{ of} \\ [\,] \quad\quad \rightarrow \{\,\} \\ l \frown [dec] \rightarrow MFnDec[\![dec]\!](MFnDecList[\![l]\!]) \\ \text{end} \end{array}$$

Chapter 4

ELLA to ELLA Verification

4.1 The Approach

Discussions with ELLA users have lead us to believe that support is required for equivalence checking between ELLA designs. Current practice provides for comparison of results obtained from exhaustive simulation of the two designs. This is an extremely time-consuming process and is often not complete. However, the particular approach does lend itself to comparing non-ELLA and ELLA designs — say, gate-level ELLA with a high-level design in, for example, C. In the case of comparing two ELLA designs, formal support is readily available. The approach is to compare the two designs via their respective formal semantic descriptions. However, in the case of the automaton based semantics, 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. Now as there are well-known automata-theoretic algorithms for performing complementation, intersection and testing emptiness of automata, testing for equivalence is somewhat simplified. We outline the algorithms in the next section.

4.2 Algorithm for Trace Equivalence and Containment

In this section, we shall describe functional algorithms for trace equivalence and containment of IO automata. We shall base these upon early work by E.F. Moore [Moo56] to obtain efficient algorithms in the case where there is no eventuality constraints upon the *infinite* traces that are accepted (i.e. every infinite looping run is accepted). As such restrictions do not naturally arise within ELLA itself, we can use these algorithms to check for trace equivalence and containment between ELLA descriptions.

To maintain clarity and a direct mathematical interpretation, we shall write our algorithms in functional terms, and make brief observations regarding their implementation.

4.2.1 Trace containment and equivalence

Essentially, the notions of trace containment and equivalence are defined by means of the following definitions:

$Env = Name \xrightarrow{m} Val$

$ILabel = Env$

$OLabel = Env$

$IOLabel = (ILabel \times OLabel)$

$Runs : IOA \rightarrow \mathbb{P}(Trans^*)$

$$Runs(mk\text{-}IOA(sts, im, om, trns, st)) \quad \triangleq$$
$$\{rn \in Trans^* \mid start(rn[1]) \in st \wedge$$
$$\forall i \in \mathsf{inds}\, rn \cdot i < \mathsf{len}\, rn \;\Rightarrow\; finish(rn[i]) = start(rn[i+1])\}$$

$trace : Trans^* \rightarrow IOLabel^*$

$$trace(rn) \quad \triangleq$$

$\mathsf{cases}\ rn\ \mathsf{of}$
$\mathsf{nil} \rightarrow \mathsf{nil}$
$[tn] \frown rn' \rightarrow [(ivals(tn), ovals(tn))] \frown trace(rn')$
end

$IOLang : IOA \rightarrow \mathbb{P}(IOLabel^*)$

$$IOLang(ioa) \quad \triangleq \quad \{trace(rn) \mid rn \in Runs(ioa)\}$$

The above definitions define what a legal run of an IO automaton is, from which we can extract the IO language expressed in terms of traces. This allows us to define the notions of trace containment and equivalence between IO automata:

$is\text{-}trace\text{-}contained : IOA^2 \rightarrow \mathbb{B}$

$$is\text{-}trace\text{-}contained(ioa_1, ioa_2) \quad \triangleq \quad IOLang(ioa_1) \subseteq IOLang(ioa_2)$$

$is\text{-}trace\text{-}equivalent : IOA^2 \rightarrow \mathbb{B}$

$$is\text{-}trace\text{-}equivalent(ioa_1, ioa_2) \quad \triangleq \quad IOLang(ioa_1) = IOLang(ioa_2)$$

4.2.2 Implementation relation

We may also define another useful comparison between IO automata - that is, when one automaton *is implemented by* another automaton. To do so, we need to define the notion of the IO language accepted by an automaton relative to some given input language:

$ILang : IOA \rightarrow \mathbb{P}(ILabel^*)$

$$ILang(ioa) \quad \triangleq \quad \{fst(unzip(tr)) \mid tr \in IOLang(ioa)\}$$

$IOLangI : IOA \times \mathbb{P}(ILabel^*) \rightarrow \mathbb{P}(IOLabel^*)$

$$IOLangI(ioa, in\text{-}lang) \quad \triangleq \quad \{tr \in IOLang(ioa) \mid fst(unzip(tr)) \in in\text{-}lang\}$$

where *fst* is the projection on the first argument and *unzip* simply maps a sequence of paired elements into a pair of sequences.

The *implementation relation* may now be defined:

$$\textit{is-implemented-by} : \textit{IOA}^2 \rightarrow \mathbb{B}$$

$$\textit{is-implemented-by}(\textit{ioa}_1, \textit{ioa}_2) \quad \triangleq \quad \textit{IOLang}(\textit{ioa}_1) = \textit{IOLangI}(\textit{ioa}_2, \textit{ILang}(\textit{ioa}_1))$$

In essence, this says that ioa_2 implements ioa_1 whenever ioa_2, restricted to the input language for ioa_1, could do precisely the same input/output actions as can ioa_1. Therefore, ioa_2 is permitted to have additional behaviours that are unconstrained by the automaton ioa_1.

We observe that if ioa_2 is *deterministic*, then the implementation relationship degenerates to trace containment:

$$\begin{array}{l} \textit{ioa}_2 \text{ is deterministic } \Rightarrow \\ \quad \textit{is-implemented-by}(\textit{ioa}_1, \textit{ioa}_2) \Leftrightarrow \textit{is-trace-contained}(\textit{ioa}_1, \textit{ioa}_2) \end{array}$$

This is easily seen by noting that, since ioa_2 is deterministic, there would be at most one input/output trace of ioa_2, for any input trace of ioa_1.

4.2.3 Functional description of the algorithm

The basic idea is to trace through each of the machines and to match corresponding states, starting from the initial states. Essentially, we have to generate a certain kind of matching relation showing how the states of one machine correspond to the states of the other. This algorithm is essentially due to E.F.Moore [Moo56], at least for the case of comparing deterministic automata. However, our contribution here is to adapt this algorithm to the non-deterministic case. This is done, in effect, by merging Moore's algorithm with the standard *subset construction* due to Rabin and Scott [RS59] for "determinising" non-deterministic automata. This results in an algorithm that does 'on the fly' determinisation of the automata, during the process of matching states.

The heart of this algorithm revolves around setting up a binary matching relation between sets of corresponding states. Such a matching relation will be of the following type:

$$\textit{MatchReln} = (\textit{State}\text{-set} \times \textit{State}\text{-set})\text{-set}$$

This relation is defined so that matching pairs of sets share a common trace prefix and should also have corresponding transitions. The main functions for computing the relation are as follows

$$\textit{trans-maps} : \textit{Trans}\text{-set} \times \textit{State}\text{-set} \rightarrow \textit{IOLabel}\text{-set}$$

$$\textit{trans-maps}(\textit{ts}, \textit{sts}) \quad \triangleq \quad \{(\textit{ivals}(\textit{tn}), \textit{ovals}(\textit{tn})) \in \textit{IOLabel} \mid \exists \textit{tn} \in \textit{ts} \cdot \textit{from}(\textit{tn}) \in \textit{sts}\}$$

$$\textit{extend-trans} : (\textit{Trans}\text{-set} \times \textit{Trans}\text{-set}) \rightarrow$$
$$(\textit{ILabel} \times \textit{OLabel} \times \textit{State}\text{-set} \times \textit{State}\text{-set}) \rightarrow \textit{MatchReln}$$

$$\begin{array}{l} \textit{extend-trans}(\textit{ts}_1, \textit{ts}_2)(\textit{im}, \textit{om}, \textit{ms}_1, \textit{ms}_2) \quad \triangleq \\ \quad \{(\textit{rs}_1, \textit{rs}_2) \in (\textit{State}\text{-set} \times \textit{State}\text{-set}) \\ \quad\quad \mid \textit{rs}_1 = \textit{image-trans}(\textit{im}, \textit{om}, \textit{ts}_1, \textit{ms}_1) \wedge \\ \quad\quad\;\; \textit{rs}_2 = \textit{image-trans}(\textit{im}, \textit{om}, \textit{ts}_2, \textit{ms}_2)\} \end{array}$$

$$image\text{-}trans : ILabel \times OLabel \times Trans\text{-set} \times State\text{-set} \rightarrow State\text{-set}$$

$$\begin{array}{l} image\text{-}trans(im, om, ts, ms) \quad \triangleq \\ \quad \{to(tn) \in State \mid \\ \quad\quad tn \in ts \ \wedge \ from(tn) \in ms \wedge \ ivals(tn) \ \subseteq im \wedge \ ovals(tn) \ \subseteq om \\ \quad \} \end{array}$$

$$ExpandMatchReln : (Trans\text{-set} \times Trans\text{-set}) \rightarrow MatchReln \rightarrow MatchReln$$

$$\begin{array}{l} ExpandMatchReln(ts_1, ts_2)(m\text{-}rel) \quad \triangleq \\ \quad m\text{-}rel \ \cup \\ \quad\quad \bigcup \{extend\text{-}trans(ts_1, ts_2)(im, om, ms_1, ms_2) \mid \\ \quad\quad\quad (im, om) \in \ (trans\text{-}maps(ts_1, ms_1) \ \cup \ trans\text{-}maps(ts_2, ms_2)) \\ \quad\quad\quad \wedge (ms_1, ms_2) \in \ m\text{-}rel \\ \quad\quad \} \end{array}$$

$$StepMatchReln : (Trans\text{-set} \times Trans\text{-set}) \rightarrow MatchReln \rightarrow MatchReln$$

$$\begin{array}{l} StepMatchReln(ts_1, ts_2)(m\text{-}rel) \quad \triangleq \\ \quad \text{let } new\text{-}m\text{-}rel = ExpandMatchReln(ts_1, ts_2)(m\text{-}rel) \\ \quad \text{in} \\ \quad\quad \text{if } (new\text{-}m\text{-}rel \subseteq m\text{-}rel) \\ \quad\quad \text{then } new\text{–}m\text{-}rel \\ \quad\quad \text{else } StepMatchReln(ts_1, ts_2)(new\text{-}m\text{-}rel) \end{array}$$

$$ComputeMatchReln : IOA^2 \rightarrow MatchRel$$

$$\begin{array}{l} ComputeMatchReln(m_1, m_2) \quad \triangleq \\ \quad \text{let } mk\text{-}IOA(\_, \_, \_, ts_1, sts_1) = m_1, \\ \quad\quad mk\text{-}IOA(\_, \_, \_, ts_2, sts_2) = m_2 \\ \quad \text{in} \\ \quad\quad\quad StepMatchReln(ts_1, ts_2)(\{(sts_1, sts_2)\}) \end{array}$$

Note that the function *StepMatchReln* is sufficiently well-defined insofaras that $m\text{-}rel \subseteq new\text{-}m\text{-}rel$ by definition of *ExtendMatchReln*. Since the number of possible mappings fulfilling this condition is finite, we can conclude that *StepMatchReln* will eventually reach a fixpoint.

Moreover, note that the use of *trans-maps* will ensure that *im* and *om* are selected so that

$$(\emptyset, \emptyset) \notin \ expand\text{-}trans(ts_1, ts_2)(im, om, ms_1, ms_2)$$

and assuming that $(\emptyset, \emptyset) \notin \ m\text{-}rel$, we have that

$$(\emptyset, \emptyset) \notin \ ExpandMatchReln(ts_1, ts_2)(m\text{-}rel)$$

We may now use the above functions and predicates to define predicates corresponding to trace containment and equivalence:

$$Contained\text{-}By : IOA^2 \rightarrow \mathbb{B}$$

$$Contained\text{-}By(m_1, m_2) \quad \triangleq$$
$$\quad \text{let } m\text{-}rel = ComputeMatchReln(m_1, m_2) \text{ in}$$
$$\quad\quad \neg \exists\ (ms_1, ms_2) \in m\text{-}rel \ \cdot\ ms_1 \neq \emptyset \ \wedge\ ms_2 = \emptyset$$

$$Equiv : IOA^2 \rightarrow \mathbb{B}$$

$$Equiv(m_1, m_2) \quad \triangleq$$
$$\quad \text{let } m\text{-}rel = ComputeMatchReln(m_1, m_2) \text{ in}$$
$$\quad\quad \forall\ (ms_1, ms_2) \in m\text{-}rel \ \cdot\ ms_1 \neq \emptyset \ \wedge\ ms_2 \neq \emptyset$$

The correctness of the above may be seen by observing the following:

- By construction, sets of states are related to those corresponding sets of states in the other machine which share a common prefix trace.
- Hence, the matching relation only contains Ø in either its domain or range when there is some trace contained in one machine but not the other.
- Accordingly, the presence or absence of Ø in the matching relation can indicate equivalence, containment in one or other direction, or inequivalence.

4.2.3.1 Implementation aspects

Any efficiency gains from using a concrete implementation of this algorithm are likely to involve the following factors. First of all, the comparison and matching process can be arranged to follow the transition structure of one of the machines, so reducing the number of comparisons between machine states that could never be related on structural grounds. This is an immeadiate gain over constructions that use Cartesian product of machines and complementation. Secondly, we can also ensure that few comparisons are repeated by only expanding the match relation for those state set pairs that were *freshly* generated during the current iteration. This further implies that the test to check if a given set pair belongs to the matching relation will need to be efficient.

Thirdly, by keeping track of which machine fails to have transitions that the other machine has required, it is possible to say, **at the end of constructing the matching relation**, whether the machines are equivalent, one is contained in the other or if they are otherwise inequivalent. Therefore, this avoids duplication of matching effort when checking for equivalence – with other algorithms, such duplication is unavoidable. Overall, therefore, we expect to be able to check for conflicts and mismatches during the matching process. In addition, this will allow for the timely ‘on the fly’ generation of error reports or, alternatively, it allows for the checking process to halt as soon as the outcome is determined. Finally, we expect that there should be little additional overhead when either of the automata are already deterministic, as is to be expected when checking correct ELLA descriptions.

4.3 Example: SR Latch

In this section we give two BKE descriptions of the SR-latch. The first is a algorithmic description and the second a gate-level description in terms of Nor-gates. We describe the construction of the automaton for each of these descriptions and show that the semantic objects generated from the two descriptions are trace equivalent; in fact, they are isomorphic.

4.3.1 Algorithmic description

The SR-latch is described by the following BKE text

```
fn SR-FUN(r, s) =
 let q = case (r,s) of
          (1,0): 0,
          (0,1): 1,
          (1,1): 0
        else dq
        esac,
     dq = IDelay(q,0,1)
 in (q,¬ q).
```

To construct the automaton for this function, we first construct that for the case statement

```
case (r,s) of
  (1,0): 0,
  (0,1): 1,
  (1,1): 0
else dq
esac
```

Following the construction described in section 3.3, we construct the automaton for *chooser-ioa*. This is a single-state automaton, so all transitions are from state 0 to itself.[1]

In						Out
−1	−2	−3	−4	−5	−6	1
1	0	0				0
1	0	1				1
0	1		0			0
0	1		1			1
1	1			0		0
1	1			1		1
0	0				0	0
0	0				1	1

Inputs −1 and −2 will be ‘connected’, using the causal product, to the outputs of the selector expression ((r,s)), and the other inputs to the outputs of the choice actions.

The automaton for the selector is the single-state automaton

In		Out	
r	s	1	2
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

[1] In the following, blank entries for inputs denote that these inputs can take any value, and states will be omitted from the tables of single state automata.

The three automata for the choice actions are single-state automata with no inputs.

Out
1
0

Out
1
1

Out
1
0

and the else action has single state-automaton, which copies input to output.

In	Out
dq	1
0	0
1	1

Combining these using the causal product (after renaming) gives the single-state automaton

In			Out
r	s	dq	1
1	0		0
0	1		1
1	1		0
0	0	0	0
0	0	1	1

IDelay(q,0,1) has the two-state automaton (with initial state 0)

From	In	Out	To
	q	1	
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

We now construct the automaton for the let construction. The automaton for the returned value is the single-state automaton

In	Out	
q	1	2
0	0	1
1	1	0

The two automata whose causal product give the binding of the let clause are

In			Out
r	s	dq	q
1	0		0
0	1		1
1	1		0
0	0	0	0
0	0	1	1

From	In	Out	To
	q	dq	
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

The causal product of these two automata is

From	In		Out		To
	r	s	q	dq	
0	1	0	0	0	0
0	0	1	1	0	1
0	1	1	0	0	0
0	0	0	0	0	0
1	1	0	0	1	0
1	0	1	1	1	1
1	1	1	0	1	0
1	0	0	1	1	1

Since this automaton is the product of an automaton with states $\{0\}$ and another with states $\{0, 1\}$, its states should have names $(0, 0)$ and $(0, 1)$. In order to simplify notation we have renamed them as 0 and 1; we adopt this approach consistently throughout this section. A simple renaming of inputs and outputs, and some merging of transitions, gives the following automaton for the function SR-ALG, shown in figure 4.1.

From	In		Out		To
	–1	–2	1	2	
0		0	0	1	0
0	0	1	1	0	1
0	1	1	0	1	0
1	1		0	1	0
1	0		1	0	1

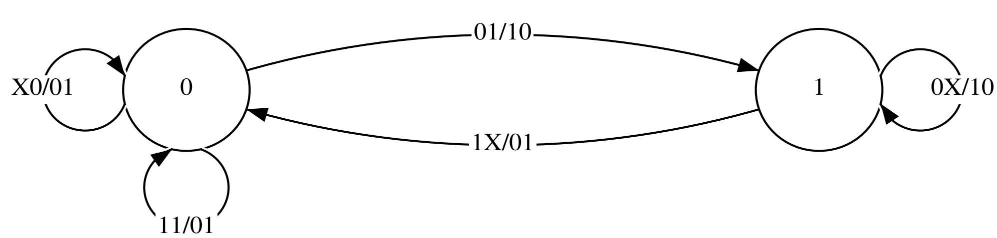

Figure 4.1: SR-latch

4.3.2 Gate-level description

We now give a gate-level description of the SR-latch, whose circuit is shown in figure 4.2

```
fn NOR(x, y) = ¬(x + y),
```

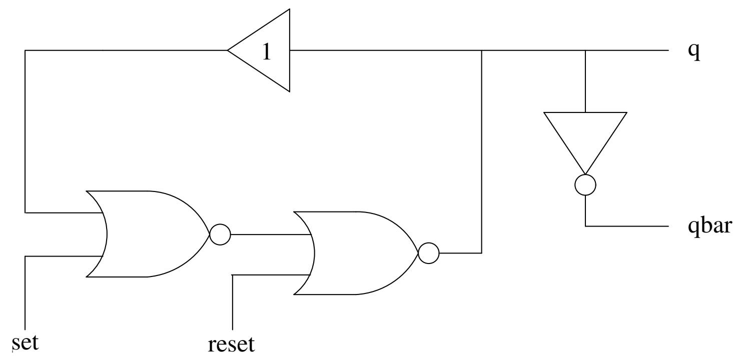

Figure 4.2: SR-latch

```
fn BDEL(y) = IDelay(y,0,1),

fn SR-GATE(reset, set) =
  let nor  = NOR(set,BDEL(q)),
      q    = NOR(reset,nor),
      qbar = ¬(q)
  in (q,qbar).
```

The functions NOR and DEL have automata

NOR		
In		Out
−1	−2	1
0	0	1
0	1	0
1	0	0
1	1	0

BDEL			
From	In	Out	To
	−1	1	
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

so that NOR(set,BDEL(q)) is the product of the three automata

In		Out
−1	−2	1
0	0	1
0	1	0
1	0	0
1	1	0

In	Out
set	−1
0	0
1	1

From	In	Out	To
	q	−2	
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

which results in the table

From	In		Out	To
	set	q	1	
0	0	0	1	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	0
1	1	1	0	1

We now construct the automaton for the let construction. The automaton for the returned value is the single-state automaton

In		Out	
q	qbar	1	2
0	0	0	0
0	1	0	1
1	0	1	0
1	1	1	1

The three other automata required to give the binding of the let clause are

From	In		Out	To
	set	q	nor	
0	0	0	1	0
0	0	1	1	1
0	1	0	0	0
0	1	1	0	1
1	0	0	0	0
1	0	1	0	1
1	1	0	0	0
1	1	1	0	1

In		Out
reset	nor	q
0	0	1
0	1	0
1	0	0
1	1	0

In	Out
q	qbar
0	1
1	0

This gives the following automaton for the let expression

From	In		Out		To
	reset	set	q	qbar	
0	0	0	0	1	0
0	0	1	1	0	1
0	1	0	0	1	0
0	1	1	0	1	0
1	0	0	1	0	1
1	0	1	1	0	1
1	1	0	0	1	0
1	1	1	0	1	0

which finally gives the following for the function SR-GATE

From	In		Out		To
	−1	−2	1	2	
0		0	0	1	0
0	0	1	1	0	1
0	1	1	0	1	0
1	1		0	1	0
1	0		1	0	1

This is identical to the automaton derived for the function SR-ALG.

4.4 Example: Twisted Ring-Counter

Our second example to illustrate ELLA to ELLA verification is a three-bit twisted-ring counter[2]

Informally, a twisted ring counter, as in [BEGS89], 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 4.1, Invalid values can occur through device error, State 2: (0,1,0) and State 5: (1,0,1). However,

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 4.1: Sequence of Outputs

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 0.

4.4.1 First Description of Twisted Ring-Counter

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

[2]In [BEGS89] a three-bit twisted ring-counter is used as the basis of a case-study of formal specification simulation and verification. The purpose of this study was to compare a selection of specification and theorem-proving systems, namely Boyer-Moore, HOL, Temporal Logic and OBJ3. The ring-counter therefore serves as an ideal candidate to illustrate the language BKE, its formal IO automaton semantics and verification process, because the example design and sample verifications are small enough to describe completely within this paper, and results can be compared with those given in [BEGS89], which may appear in a more familiar form.

4.4.1.1 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:
   DELAY(f,1).



FN DELAY_3 = ([3]bool: in) -> [3]bool:
 ( BOOL_DELAY(in[1],1),
   BOOL_DELAY(in[2],1),
   BOOL_DELAY(in[3],1)
 ).

FN COUNTER = (bool: reset) -> [3]bool:
BEGIN
  MAKE DELAY_3: tbd.
  JOIN NEXT (reset, tbd) -> tbd.
  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 DELAY_3 function.

4.4.1.2 BKE Description

In BKE the high-level description is written using a recursive let.

```
fn NEXT(reset,p,q,r) =
case reset of
```

```
    0 : case (p,q,r) of
            (0,0,0): (0,0,1),
            (0,0,1): (0,1,1),
            (0,1,1): (1,1,1),
            (1,1,1): (1,1,0),
            (1,1,0): (1,0,0),
            (1,0,0): (0,0,0),
            (0,1,0): (1,0,0),
            (1,0,1): (0,1,0)
        else (0,0,0)
        end,
     1 : (0,0,0)
  else (0,0,0)
  end.

  fn DELAY_3(p,q,r) =
  ( Idelay(p,0,1), Idelay(q,0,1), Idelay(r,0,1) ).

  fn COUNTER(reset) =
  let (dp,dq,dr) = DELAY_3(p,q,r),
      (p,q,r) = NEXT(reset,dp,dq,dr)
  in  (dp,dq,dr).
```

4.4.1.3 IO Automaton

Here we outline the construction of the IO automaton semantic object for the above BKE description.

The semantic object denoting the NEXT function is clearly a single-state automaton (it has no delays) with 16 transitions, each corresponding to the various choices of the case construct.

Next consider the semantic object that represents the rowed expression

```
( Idelay(p,0,1), Idelay(q,0,1), Idelay(r,0,1) ).
```

It is constructed by forming the causal product of the individual row-elements. Each element is in fact denoted by a two state automaton, similar to that given in figure 3.2 of section 3.2.1.2, except that the input to each is named by p, q, r and d respectively and the outputs are named 1, 2 and 3 respectively. The resulting product can have at most eight states. As the alphabets of each element are independent, and each state has a transition to every state (fully connected), the resulting automaton is also fully connected (the causal product degenerates to the usual cross product) and has 8 states and 64 transitions. This automaton is then transformed into an object denoting the function DELAY_3, essentially by renaming the inputs p, q and r by the internal names −1, −2 and −3.

The semantic object for the COUNTER is obtained by "solving" the mutually recursive equations of the let clause. This effectively renames the outputs of the eight state delay object (DELAY_3) to be the appropriate inputs of the next state automaton (NEXT) and then forming the causal product. This has the effect of restricting the transition of the 8 state automaton to be those allowed by the single-state NEXT automaton.

A picture of the resulting IO automaton is given in Figure 4.4

4.4.2 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 4.3. There is a single input *reset* $\in \{0, 1\}$ which resets the circuit to State 0 if it is raised. 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.

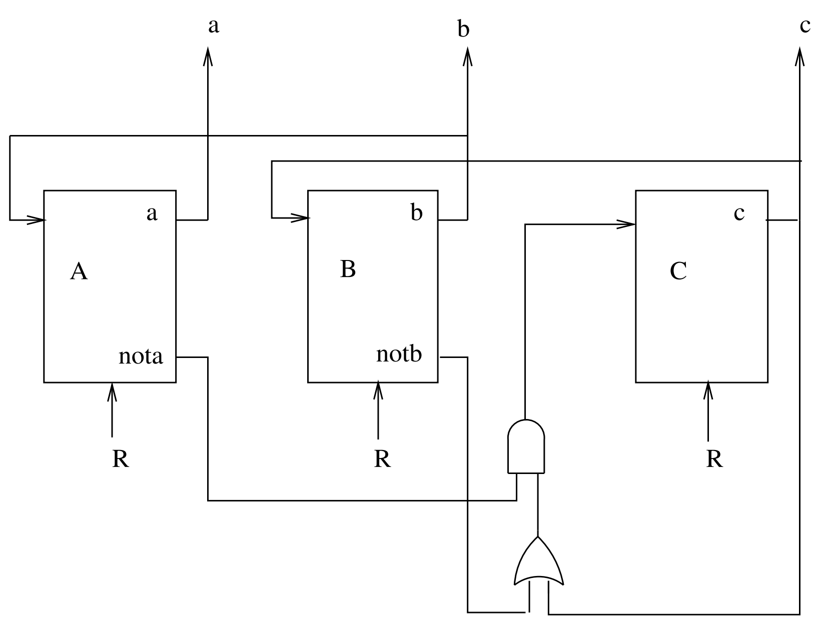

Figure 4.3: Twisted Ring-Counter

4.4.2.1 ELLA Description

An ELLA-text description is now given for the ring-counter, using a two-valued Boolean data type. The initial state is assumed here to be State 0. A "*MAKE … JOIN … OUTPUT*" description style has been adopted, and the description has been written to resemble as far as possible the BKE description which follows, so that equivalent parts in each text can be compared readily.

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

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

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

FN AND = (bool: x y) -> bool:
   # defined in terms of NOT and OR: #
   NOT( NOT(x) OR NOT(y)).

FN LATCH = (bool: i 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( i AND (NOT r)).

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

4.4.2.2 BKE Description

The ELLA-text would then be translated into BKE:

```
fn BOOL_DELAY(x) =
  Idelay(x, 0, 1),

fn AND(x, y) =
 ¬(¬ x  +  ¬ y),

fn LATCH(i, r) =
  BOOL_DELAY(AND(i, ¬ r)),

fn TWIST(reset) =
  let  a = LATCH(b, reset),
       b = LATCH(c, reset),
       c = LATCH(AND(¬ a, ¬ b + c), reset)
  in (a, b, c).
```

Note that if *reset* = 0, then the counter will cycle through the six valid states; if *reset* = 1 then state 0 is entered and (0,0,0) will be output at the next time unit. If the reset is always raised, then the counter always outputs (0,0,0), after the first time step, and remains in state 0. Whatever the initial state, or value on *reset*, the counter will output (0,0,0) an arbitrary number of times. However, it is not the case that the counter will output any other value an arbitrary number of times; for example it will only output (1,1,1) at most once if *reset* is always held high. This is the case for any output value except (0,0,0).

4.4.2.3 IO Automaton Description

An IO automaton is now constructed from the BKE description of the counter. In fact, it is isomorphic up to state names with that of Figure 4.4. As may be expected, the automaton has eight states; as before, they have been given labels which happen to correspond to the integer values of the boolean output produced when leaving each state. Again, the state labels do not have any further significance.

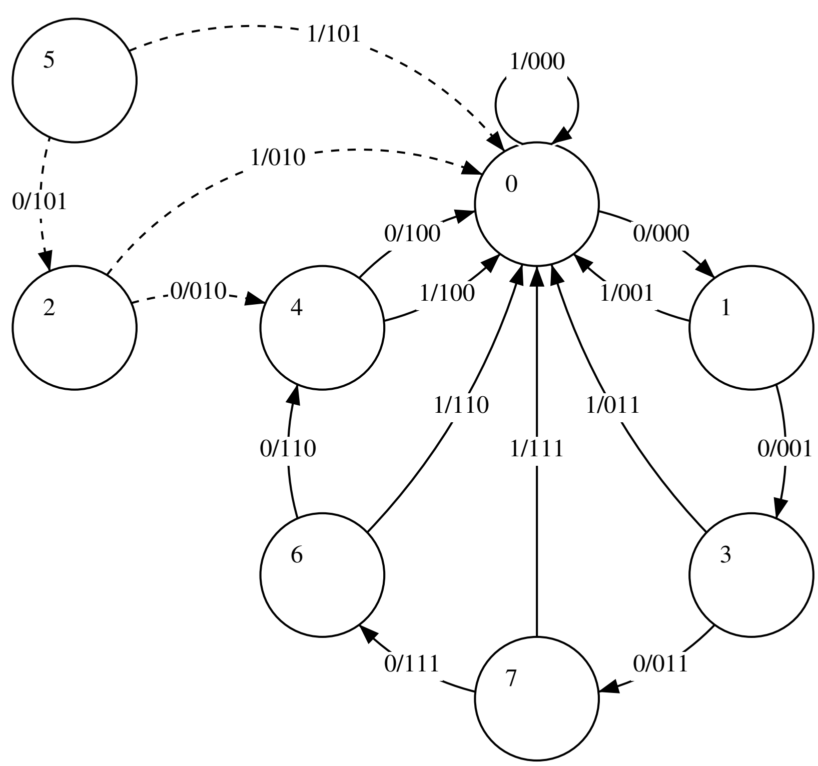

Figure 4.4: IO Automaton for Twisted Ring-Counter

The automaton shown in Figure 4.4 is constructed according to the semantic rules for BKE described in section 3.3, by composition of the automata representing the sub-components of each unit. 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*.

This process is now described, and the transition tables for these intermediate automata are given.

The function object for *LATCH* has an argument count of 2, and a two-state automaton whose transition set is given by the following table. We have labelled the states 0 and 1. The initial state is state 0. Note that the output value for each transition is the same as the label on the *from* state. This functional dependence of the next output on the state is a consequence of the fact that the outermost operation is a *DELAY*.

From	Input		Output	To
	1	2	1	
0		1	0	0
1		1	1	0
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	1	0	1	1

The three equations in the final binding for the ring counter give rise to three automata, whose causal product is the automaton for the ring counter. We will construct the product in a single step, using the fact that the product operation is associative up to isomorphism of automata.

The automaton for the equation a = *LATCH*(b, *reset*) is obtained from the *LATCH* automaton by simply renaming the inputs and outputs. The initial state is still state 0.

From	Input		Output	To
	b	reset	a	
0		1	0	0
1		1	1	0
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	1	0	1	1

The automaton for the equation b = *LATCH*(c, *reset*) is obtained similarly to the above.

From	Input		Output	To
	c	reset	b	
0		1	0	0
1		1	1	0
0	0	0	0	0
0	1	0	0	1
1	0	0	1	0
1	1	0	1	1

The equation for c is more complicated, since c occurs on both left and right hand sides. We proceed in three stages.

First, the expression *AND*($\neg a, \neg b + c$) contains no *DELAY* operators, and so gives a one-state automaton. The transition table is given below.

Input			Output
a	b	c	1
1			0
0	0		1
0	1	0	0
0	1	1	1

Substituting the above table into the table for the *LATCH* automaton and renaming input 2 to *reset* (which amounts to evaluating the function object with the correct arguments) gives the following two-state automaton for the expression $LATCH(AND(\neg a, \neg b + c), reset)$.

From	Input				Output	To
	a	b	c	reset	1	
0				1	0	0
1				1	1	0
0	1			0	0	0
0	0	0		0	0	1
0	0	1	0	0	0	0
0	0	1	1	0	0	1
1	1			0	1	0
1	0	0		0	1	1
1	0	1	0	0	1	0
1	0	1	1	0	1	1

Next, we must rename the output of this automaton to c. In doing so, we must suppress the input c and exclude all transitions in which the values of c and 1 disagree. We then obtain the following automaton for the equation $c = LATCH(AND(\neg a, \neg b + c), reset)$. Once again, the initial state is 0.

From	Input			Output	To
	a	b	reset	c	
0			1	0	0
1			1	1	0
0	1		0	0	0
0	0	0	0	0	1
0	0	1	0	0	0
1	1		0	1	0
1	0	0	0	1	1
1	0	1	0	1	1

The automaton for the ring counter is obtained by taking the causal product of the automata for a, b and c. The calculation is greatly simplified by the functional dependence of the output values on the *from* state, which carries over into the product. The state set is now $\{0, 1\}^3$, with the order on the factors given by (a, b, c). The initial state is $(0, 0, 0)$.

From	Input	Output			To
	reset	a	b	c	
(0, 0, 0)	0	0	0	0	(0, 0, 1)
(0, 0, 0)	1	0	0	0	(0, 0, 0)
(0, 0, 1)	0	0	0	1	(0, 1, 1)
(0, 0, 1)	1	0	0	1	(0, 0, 0)
(0, 1, 0)	0	0	1	0	(1, 0, 0)
(0, 1, 0)	1	0	1	0	(0, 0, 0)
(0, 1, 1)	0	0	1	1	(1, 1, 1)
(0, 1, 1)	1	0	1	1	(0, 0, 0)
(1, 0, 0)	0	1	0	0	(0, 0, 0)
(1, 0, 0)	1	1	0	0	(0, 0, 0)
(1, 0, 1)	0	1	0	1	(0, 1, 0)
(1, 0, 1)	1	1	0	1	(0, 0, 0)
(1, 1, 0)	0	1	1	0	(1, 0, 0)
(1, 1, 0)	1	1	1	0	(0, 0, 0)
(1, 1, 1)	0	1	1	1	(1, 1, 0)
(1, 1, 1)	1	1	1	1	(0, 0, 0)

The last table matches the transition set for the IO automaton representing the twisted ring-counter shown in Figure 4.4.

4.4.3 Equivalence of semantic objects

One can see by inspection that the semantic objects denoting the high and low (gate) level BKE descriptions of the twisted ring counter are isomorphic, indeed, we have only given one picture of the structure. The isomorphism implies trace equivalence of the two models and hence that the gate level circuit faithfully models the first description.

In practice, it is necessary to apply the algorithm given in section 4.2. This will establish the trace equivalence and containment relationship that holds between the models. Of importance is the fact that the models are isomorphic up to state names, then the algorithm is of linear time complexity.

Chapter 5

Conclusions and future work

We have presented a direct IO automaton based compositional semantics for Boolean Kernel ELLA, BKE, a functional language designed as an abstraction of Core ELLA, a covering subset of the hardware description language ELLA. The main purpose of this formal semantics is to enable application of automata-based formal verification to hardware designs written in ELLA. An IO automaton semantics was chosen because it has an associated automated verification system which, furthermore, can be simplified by exploiting determinism in the ELLA language. Our use of IO automata also leads to semantic structures that are state compact, although there may be some combinatorial explosion on transitions. Finally, we demonstrated our semantics and the style of verification on a small example.

There are several directions of future work. We are extending BKE so that it coincides with Kernel ELLA, an external representation of the internal data structures. This will then be used to build an automaton compilation system for Core ELLA. Essentially, this requires incorporation of the full typing system of ELLA. This will then provide a state-based verification system for ELLA. There are, however, several problems to be overcome before such a system is practicable, for example, a more compact representation must be chosen for the transition labelling. Prototype tools to perform ELLA-to-BKE translation have been prototyped, as have tools for building IO automaton models of BKE descriptions and for automation of the verification process, including automaton tools to produce complementation, intersection and composition. This will enable us, and others, to gain practical experience with undertaking larger scale verification of ELLA designs.

Future reports will address the following:-

- the problems of efficient automaton represention, first within the Boolean context and also when further data types are introduced,

- the problems associated with retiming of circuits and their specifications.

References

[BC86] M.C. Browne and E.M. Clarke.
SML - a high level language for the design and verification of finite state machines.
In *From H.D.L. descriptions to guaranteed correct circuit designs*. IFIP, September 1986.

[Bec87] A. Beck.
Using temporal logic to verify properties of circuit designs described in ELLA.
Master's thesis, St. Cross College, Oxford University, September 1987.

[BEGS89] H. Barringer, D. Edwards, J. Gurd, and V. Stavridou.
The HAVE project final report.
University of Manchester Computer Science Department, 1989.

[BFG89] H. Barringer, M. Fisher, and G.D. Gough.
Fair SMG and Linear Time Model Checking.
In *Proceedings of Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, June 1989.
(Published in *Lecture Notes in Computer Science* volume 407).

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

[BGM91] H. Barringer, G.D. Gough, and B.Q. Monahan.
Operational Semantics of Hardware Design Languages.
In *Advanced Research Workshop on Correct Hardware Design Methodologies (CHARME'91)*, Turin, Italy, May 1991. Elsevier Science Publishers (IFIP) North-Holland.

[CE82] E.M. Clarke and E.A. Emerson.
Synthesis of synchronization skeletons from branching time temporal logic.
In *Proceedings of the Workshop on Logics of Programs*, pages 52–71, Berlin, 1982. Volume 131, LNCS, Springer-Verlag.

[CES83] E.M. Clarke, E.A. Emerson, and A.P. Sistla.
Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications: A Practical Approach.
In *Proceedings of the Tenth ACM Symposium on the Principles of Programming Languages*, 1983.

[Com90] Computer General Electronic Design, The New Church, Henry Street, Bath, Avon, BA1 1JR, United Kingdom.
The ELLA Language Reference Manual, 4.0th edition, 1990.

[GB88] G. D. Gough and H. Barringer.
A Semantics Driven Temporal Verification System.
In *Proceedings of ESOP '88*. LNCS 300, Springer-Verlag, March 1988.

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

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

[HM91] M.G. Hill and J.D. Morision.
Preliminary Core ELLA Definition.
Internal Memorandum 1.1c, Formal Verification Support for Ella, IED project 4/1/1357, February 1991.

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

[HWM90] M.G. Hill, E.V. Whiting, and J.D. Morison.
Formal Semantic Definition of ELLA Timing.
Internal Memorandum Memo 4436, Royal Signals and Radar Establishment, November 1990.

[Jon90] Cliff B. Jones.
Systematic Software Development using VDM.
Prentice Hall, second edition, 1990.

[JS90] Cliff B. Jones and R. C. Shaw.
Case Studies in Systematic Software Development.
Prentice Hall, 1990.

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

[Mil90] R. Milner.
Concurrency and Communication.
Prentice Hall, 1990.

[Mon91] B.Q. Monahan.
A note on the semantics of ambiguity.
Project working paper, University of Manchester Computer Science Department, January 1991.

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

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

[RRSV87] J.L. Richier, C. Rodriguez, J. Sifakis, and J. Voiron.
Xesar a tool for protocol verification: User's guide.
Technical report, Laboratoire de Génie Informatique de Grenoble, IMAG-Campus, B.P.53H, 38041 Grenoble Cedex, France, September 1987.

[RS59] M.O. Rabin and D. Scott.

Finite Automata and their Decision Problems.
IBM Journal of Research and Development, 3(2):114–125, April 1959.

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

[VW86] M.Y. Vardi and P. Wolper.
Automata Theoretic techniques for Modal Logics of Programs.
Journal of Computer and System Sciences, 32(2):183–221, April 1986.