FORMAL VERIFICATION SUPPORT FOR ELLA

IED 4/1/1357

Towards an operational semantics of Ella*

Howard Barringer, Brian Monahan

University of Manchester†

Mike Hill

RSRE

October 1990

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

1 Introduction

The Ella™ language has been designed to model hardware circuits from high level behavioural descriptions down to the gate level[1]. At the heart of the Ella language lies a small number of primitive, syntactically orthogonal constructs[2] from which a number of higher level constructs are built. Thus, in many cases, there exist transformations from complex syntactic structures onto the lower level ones.

It is the aim of this document to start considering the operational semantics of the primitive, orthogonal constructs. The operational semantic definitions developed in this deliverable will then be used as a basis for later work to describe a core set of semantic constructs from which the whole of the Ella language is built. Thus this deliverable can safely defer consideration of many features of the full Ella language[3] to the full scale semantic study later in the project.

The various operational semantics described in this deliverable provide a first step at formalising particular aspects of signal computation thought to be relevant to an operational model of Ella's semantics. A hierarchy of languages is introduced with the purpose of examining specific semantic features, such as timing delay, for example. By this gradual introduction of concepts, each one can be discussed individually without the clutter of others that are less relevant. Each of these languages will be given an operational semantics using an inference rule notation based upon the Structural Operational Semantics[4, 5] approach.

Furthermore, this operational technique makes it possible to 'compile' language descriptions into state machines suitable for verification[6]. The operational semantics of Ella will therefore provide an essential component in the development of verification support tools during later stages of the project. In the context of Ella, previous work by Davies[7] has discussed the use of transformation based techniques to calculate equivalence of simplified Ella descriptions.

We also explain and set up our notation by giving illustrative examples of its use. A general glossary is also included in an appendix.

1.1 A Simple Beginning

To begin with, we present a short illustrative example in which an operational semantics is constructed for a very small parallel programming language. We first define the language syntax and then demonstrate how to set up an operational semantics. The semantic rules for the language expressions and structures are given. Finally, we use the semantic rules to analyse an example program.

1.1.1 Concrete and Abstract Syntax

Language syntax can be thought of as either *concrete* or *abstract*. In the case of concrete syntax the letter casing, position of language symbols and tokens are all important. This is because its main design function is to provide a basis for communication between the user and the machine – via parsing – or between users themselves.

As an example, we give a BNF-like concrete syntax for the statements of a simple parallel programming language:

$$S ::= \textbf{skip} \mid V{:=}E \mid S;S \mid \textbf{if}\ E\ \textbf{then}\ S\ \textbf{else}\ S \mid \textbf{while}\ E\ \textbf{do}\ S \mid S \parallel S$$

with the syntax for its expressions given by:

$$\begin{array}{l} E ::= \textbf{true} \mid \textbf{false} \mid E\ \textbf{and}\ E \mid \textbf{not}\ E \mid V \\ V ::= \mathrm{Id} \end{array}$$

where Id is some syntactic class of identifiers.

To motivate the idea of an abstract syntax we mention that the objective of a semantic model is to make clear the intention and significance of certain components of the syntax, which are known as *phrases*. Therefore it is useful to extract the barest structural features of the language as a first step and to then define the semantics over this somewhat reduced syntactic structure.

To do this we use abstract syntax, which for our example language, we take as having the following form:

$$\begin{array}{lcl} \mathrm{Stmt} & \stackrel{\mathrm{def}}{=} & \{\textbf{skip}\} \cup \mathrm{Asgn} \cup \mathrm{Seq} \cup \mathrm{If} \cup \mathrm{While} \cup \mathrm{Parallel} \\ \mathrm{Asgn} & \stackrel{\mathrm{def}}{=} & \textbf{assign}(\mathrm{Var} \times \mathrm{Expr}) \\ \mathrm{Seq} & \stackrel{\mathrm{def}}{=} & \textbf{seq}(\mathrm{Stmt} \times \mathrm{Stmt}) \\ \mathrm{If} & \stackrel{\mathrm{def}}{=} & \textbf{if}(\mathrm{Expr} \times \mathrm{Stmt} \times \mathrm{Stmt}) \\ \mathrm{While} & \stackrel{\mathrm{def}}{=} & \textbf{while}(\mathrm{Expr} \times \mathrm{Stmt}) \\ \mathrm{Parallel} & \stackrel{\mathrm{def}}{=} & \textbf{parallel}(\mathrm{Stmt} \times \mathrm{Stmt}) \end{array}$$

with

$$\begin{array}{lcl} \mathrm{Expr} & \stackrel{\mathrm{def}}{=} & \{\ \textbf{true}\ \} \cup \{\ \textbf{false}\ \} \cup \textbf{and}(\mathrm{Expr} \times \mathrm{Expr}) \cup \textbf{not}(\mathrm{Expr}) \cup \mathrm{Var} \\ \mathrm{Var} & \stackrel{\mathrm{def}}{=} & \textbf{var}(\mathrm{Id}) \end{array}$$

where the notation "$\textbf{label}(\ S_1 \times \cdots \times S_k)$" is a set consisting of the Cartesian product $\bigtimes_{i \in 1..k} S_i$, labelled by the tag **label**. Therefore, the above equations define sets of (labelled) tree structures. Since these definitions involve recursive self reference, we require that they define the least sets satisfying the above equations, where the sets are ordered by inclusion.

1.1.2 Composite Syntax

To be completely formal, we should add that a mapping is required from concrete parse trees to abstract syntax trees. However, for the work herein, the mapping is always quite straightforward and can safely be omitted as it contributes little to the understanding of either the syntax or the semantics.

Because this mapping will often be direct, we can exploit this simplicity by giving only a single *syntactic* definition. This will combine sufficient features of both syntax definitions to enable each to be derived separately. Such a definition is called here a *composite syntax*.

For the purpose of illustration, a composite syntax is given for our example language, since the abstract and concrete syntax definitions are already similar in structure:

$(s, s_1, s_2 \in \text{Stmt})$
$s ::= \textbf{skip} \mid v_1 := e_1 \mid s_1 \; ; s_2 \mid \textbf{if } e_1 \textbf{ then } s_1 \textbf{ else } s_2 \mid \textbf{while } e_1 \textbf{ do } s_1 \mid s_1 \parallel s_2$

$(e, e_1, e_2 \in \text{Expr})$
$e ::= \textbf{true} \mid \textbf{false} \mid e_1 \textbf{ and } e_2 \mid \textbf{not } e_1 \mid v_1$

$(v \in \text{Var})$
$v ::= id$

$(id \in \text{Id})$

So, the concrete syntax can easily be obtained by replacing each occurrence of an indeterminate by the name of the syntactic class to which it belongs. As can also be seen, the abstract syntax structure could also be derived, by forming each alternative as a separately constructed tree object whose descendent components are determined by the types of the indeterminates that appear. The constructor labels for the tree objects all have to be unique but that is easily arranged, such as by concatenating all the keyword literals appearing in the relevant alternative.

Since we can be sure that there is an underlying abstract syntax representation for each legal phrase, we can allow the use of the composite syntax as a way of ‘quoting’ syntactic phrases within semantic definitions, e.g. we could write “**while** b **do** e_1” instead of the abstract syntax form: “**while**(b,e_1)”. Care is taken to ensure that there is no possibility of notational confusion when using this convention.

To summarise the distinctions again, concrete syntax is mainly concerned with defining a language as a set of *strings*, whereas an abstract syntax defines a language in terms of a set of *tree* structures. Such a distinction crucially permits the definition of a language’s semantics to be governed by the constituents of each phrase, rather than by the way that phrases just happen to be parsed.

When there are sufficient similarities between the *structure* of the abstract and concrete syntax definitions, we can exploit such similarity by giving a composite syntax, which is sufficient to define each type of structure from the same definition.

1.2 Operational Semantics

An operational semantics for a language is a mathematical model of the *process* of execution. It enables one to answer questions concerning the execution of programs in that language. So, for example, we may ask the question “what does my program do when started in a given initial state?”. Other questions may be concerned with the *form* of the execution such as “how many execution steps are involved in executing this program fragment?”.

Consider the following example:

a := **false**; b := **not** a

Intuitively, no matter what state we start from, this should set the variable denoted by a to some semantic value designated to stand for **false** e.g. f and the variable b to t, a semantic value which corresponds to **true**. An operational semantics enables us to argue about execution traces and so demonstrate this result in a precise, mathematical way.

To do this, we need to introduce a precise characterisation of *state*. A straightforward way of representing state is by identifying it with a mapping from variable identifiers to the semantic value each identifier is associated with. Thus, from any state, q_0, we would expect that our semantics would show that:

$$q_0 \xrightarrow{\text{a} := \textbf{false}} q_1 \xrightarrow{\text{b} := \textbf{not}\ \text{a}} q_2$$

with $q_1 = q_0 \dagger \{ a \mapsto f \}$, $q_2 = q_1 \dagger \{ b \mapsto t \}$, where $q_i \dagger q_j$ updates the state q_i with the state q_j. In other words, $q_2 = q_0 \dagger \{a \mapsto f , b \mapsto t \}$. If, for example, we put $q_0 = \{b \mapsto f , c \mapsto f \}$, then the above calculation shows that $q_1 = \{ a \mapsto f , b \mapsto f , c \mapsto f \}$ and that $q_2 = \{ a \mapsto f , b \mapsto t , c \mapsto f \}$.

1.2.1 Semantic Sets

In the above calculation, we simply assumed that the syntactic object **false** maps to the semantic object 'f' (and similarly **true** maps to 't'). In making the semantic model precise, we shall need to define this association of literal value with semantic value by giving a *meaning function*. For complex languages, this mapping can form a major part of a semantic definition, especially if a denotational semantics is given.

For our example language, we have already chosen above that the set of semantic values is the set $\{ t, f \}$ and that the state corresponds to a mapping from identifiers to values:

$$\text{Value} \stackrel{\text{def}}{=} \{ t, f \}$$
$$\text{State} \stackrel{\text{def}}{=} \text{Var} \longrightarrow \text{Value}$$

Throughout this deliverable, we shall use functions which are assumed to be sufficiently defined for the calculations to make sense. Clearly, a more detailed and careful treatment might require partial functions, but usually finite ones, whereas we have apparently used total functions. In all the examples that we consider, the functions in question will always be as sufficiently defined as needed.

In this case, the mapping from syntactic structure to semantic value is done through a simple functional relation, such as:

$$M \colon \{ \textbf{true}, \textbf{false} \} \longrightarrow \text{Value}$$

with:

$$M(\textbf{true}) \stackrel{\text{def}}{=} t$$
$$M(\textbf{false}) \stackrel{\text{def}}{=} f$$

1.2.2 Transition relations

Having defined our semantic space of values (or *universe of discourse*) we may proceed to introduce the following configuration structure and transition relation in accordance with Structural Operational Semantics:

$$\text{Config} \stackrel{\text{def}}{=} \text{State} \times \text{Stmt}$$
$$(\_ \Longrightarrow \_) \subseteq \text{Config} \times \text{Config}$$

For the purpose of this deliverable, we shall assume that the relation $(\_ \Longrightarrow \_)$ is constrained so that it only applies to well-formed (i.e. sensible) statements. These omitted conditions on statements to assert well-formedness may also be presented using an *inference rule* style; such collections of rules are often known as the *static semantics* (or *context conditions*) for a language.

Before defining the above semantic relation for statements, we need to introduce a similar semantic relation that defines the execution for expressions e.g. define:

$$\text{ExprConfig} \stackrel{\text{def}}{=} \text{State} \times \text{Expr}$$
$$(\_ \longrightarrow \_) \subseteq \text{ExprConfig} \times \text{Value}$$

using the following set of inference rules:

$$\boxed{\text{i.}}\ \frac{}{\text{q}, \textbf{false} \longrightarrow \text{f}} \qquad \boxed{\text{ii.}}\ \frac{}{\text{q}, \textbf{true} \longrightarrow \text{t}}$$

$$\boxed{\text{iii.}}\ \frac{}{\text{q}, \text{id} \longrightarrow \text{q(id)}}$$

$$\boxed{\text{iv.a}}\ \frac{\text{q}, \text{e} \longrightarrow \text{t}}{\text{q}, \textbf{not}\ \text{e} \longrightarrow \text{f}} \qquad \boxed{\text{iv.b}}\ \frac{\text{q}, \text{e} \longrightarrow \text{f}}{\text{q}, \textbf{not}\ \text{e} \longrightarrow \text{t}}$$

$$\boxed{\text{v.a}}\ \frac{\text{q}, \text{e}_1 \longrightarrow \text{t}\ , \quad \text{q}, \text{e}_2 \longrightarrow \text{t}}{\text{q}, \text{e}_1\ \textbf{and}\ \text{e}_2 \longrightarrow \text{t}} \qquad \boxed{\text{v.b}}\ \frac{\text{q}, \text{e}_\text{i} \longrightarrow \text{f (where i = 1 or 2)}}{\text{q}, \text{e}_1\ \textbf{and}\ \text{e}_2 \longrightarrow \text{f}}$$

This table of inference rules in effect introduces properties of the relation $(\_, \_) \longrightarrow \_$. As these properties may generally be held by many such relations, we shall take the smallest relation satisfying these rules i.e. the intersection of all relations that satisfy the rules.

Having now defined the execution behaviour for expressions, we can then give the semantic rules for the statements in the same manner. The semantic rules are chosen to reflect the behaviour that we wish to apply to the language. Note that we shall denote the empty program containing no further statements by Ø in the rules below.

$$\boxed{1}\ \frac{}{\text{q}, \textbf{skip} \Longrightarrow \text{q}, \varnothing} \qquad \boxed{2}\ \frac{\text{q}, \text{e} \longrightarrow \text{v}}{\text{q}, \text{id:=e} \Longrightarrow \text{q} \dagger \{\ \text{id} \mapsto \text{v}\ \}, \varnothing}$$

$$\boxed{3.a}\ \frac{q, S_1 \Longrightarrow q', S'_1}{q, S_1; S_2 \Longrightarrow q', S'_1; S_2} \qquad \boxed{3.b}\ \frac{}{q, \emptyset; S_1 \Longrightarrow q, S_1}$$

$$\boxed{4.a}\ \frac{q, b \longrightarrow t\,,\quad q, S_1 \Longrightarrow q', S'_1}{q, \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \Longrightarrow q', S'_1} \qquad \boxed{4.b}\ \frac{q, b \longrightarrow f\,,\quad q, S_2 \Longrightarrow q', S'_2}{q, \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \Longrightarrow q', S'_2}$$

$$\boxed{5.a}\ \frac{q, b \longrightarrow f}{q, \textbf{while } b \textbf{ do } S \Longrightarrow q, \emptyset} \qquad \boxed{5.b}\ \frac{q, b \longrightarrow t\,,\quad q, S \Longrightarrow q', S'}{q, \textbf{while } b \textbf{ do } S \Longrightarrow q', S'; \textbf{while } b \textbf{ do } S'}$$

For parallelism we need to consider whether it is (lock step) synchronous or (interleaved) asynchronous concurrency. If asynchronous then the corresponding rules would be:

$$\boxed{6.a}\ \frac{q, S_1 \Longrightarrow q', S'_1}{q, S_1 \parallel S_2 \Longrightarrow q', S'_1 \parallel S_2} \qquad \boxed{6.b}\ \frac{q, S_2 \Longrightarrow q', S'_2}{q, S_1 \parallel S_2 \Longrightarrow q', S_1 \parallel S'_2}$$

$$\boxed{7.a}\ \frac{}{q, (\emptyset \parallel S) \Longrightarrow q, S} \qquad \boxed{7.b}\ \frac{}{q, (S \parallel \emptyset) \Longrightarrow q, S.}$$

where rules 6.a and 6.b ensure that parallel components can each make a step. Note that in this asynchronous case, nothing is said about which statement gets done first. Different traces occur depending on which rule is chosen. Rules 7.a and 7.b are there to ensure termination of the parallel construct.

If, instead, parallelism is taken to be synchronous in shared variables then the following rules would be used:

$$\boxed{6'}\ \frac{q, S_1 \Longrightarrow q_1, S'_1\,,\quad q, S_2 \Longrightarrow q_2, S'_2\,,\quad \mathrm{diff}(q,q_1) \cap \mathrm{diff}(q,q_2) = \{\}}{q, S_1 \parallel S_2 \Longrightarrow q_1 \dagger q_2, S'_1 \parallel S'_2}$$

$$\boxed{7'.a}\ \frac{q, S \Longrightarrow q', S'}{q, (\emptyset \parallel S) \Longrightarrow q', S'} \qquad \boxed{7'.b}\ \frac{q, S \Longrightarrow q', S'}{q, (S \parallel \emptyset) \Longrightarrow q', S'}$$

where, in rule $6'$, $\mathrm{diff}(q,q_1) \cap \mathrm{diff}(q,q_2) = \{\}$ is the non-interference condition, ensuring that simultaneous update of shared variables is not permitted.

1.3 Example

Consider the following (straight-line) program:

$$\underbrace{a := \textbf{false};\ \underbrace{b := \textbf{not}\ a\ ;\ \underbrace{c := a\ \textbf{and}\ b}_{P_3}}_{P_2}}_{P_1}$$

where, for convenience, we have introduced the variables P_1, P_2, P_3 denoting the indicated (sub) programs.

Our first observations are that, for *any* states q', q'', q''', we can show the following separate results:

H1: q', a := **false** $\Longrightarrow q' \dagger \{ a \mapsto f \}, \emptyset$
H2: q'', b := **not** a $\Longrightarrow q'' \dagger \{ b \mapsto r_1 \}, \emptyset$
H3: q''', c:= a **and** b $\Longrightarrow q''' \dagger \{ c \mapsto r_2 \}, \emptyset$

where each follows from an application of rule 2 and r_1, r_2 are determined by using the expression execution rules, i.e. q', **not** a $\longrightarrow r_1$ and also q'', a **and** b $\longrightarrow r_2$.

Let the initial state be q_0, and consider the following:

q_0, a := **false**; P_2	$\Longrightarrow q_0 \dagger \{ a \mapsto f \}, \emptyset; P_2$	by use of H1 and from 3.a
	$\Longrightarrow q_0 \dagger \{ a \mapsto f \}, P_2$	from 3b

So putting $q_1 = q_0 \dagger \{ a \mapsto f \}$, we proceed:

q_1, b := **not** a; P_3	$\Longrightarrow q_1 \dagger \{ b \mapsto t \}, \emptyset; P_3$	by reference to H2 and from 3.a
	$\Longrightarrow q_1 \dagger \{ b \mapsto t \}, P_3$	from 3.b

Similarly, putting $q_2 = q_1 \dagger \{ b \mapsto t \}$, we can show that:

q_0, P_1	$\Longrightarrow q_1, \emptyset; P_2 \Longrightarrow q_1, P_2$	by use of H1
	$\Longrightarrow q_2, \emptyset; P_3 \Longrightarrow q_2, P_3$	by reference to H2
	$\Longrightarrow q_0 \dagger \{a \mapsto f, b \mapsto t, c \mapsto f \}, \emptyset$	by reference to H3

Hence we have related an arbitrary initial state, q_0, with the values of the state variables that changed after execution of the program. Note that the result is obtained by *symbolic execution* and does not force the initial state to be any more specific than necessary.

2 Logic + Delay language

Our investigation into Ella semantics starts with a development of semantic rules for a small expression based language of Booleans, which also includes delay. Intuitively, this language contains all of the most primitive ingredients for the semantic study of Ella. It is a propositional logic language, together with a transport delay primitive thus requiring consideration of timing in the semantics.

The basic idea is that an expression in this language defines a transformation from possibly several streams of input signals into a (single) stream of output signals. No feedback from output to input is possible within expressions of this language.

Language Classes

Expressions, ($e \in$ Expr) given by:-

$$e \quad ::= \quad v \mid \mathsf{high} \mid \mathsf{low} \mid \overline{e_1} \mid e_1 + e_2 \mid \Delta(e_1, d)$$

An expression can either be a named port, the elements high or low , the negation of an expression, the logical 'or' of two expressions or the value of an expression, simply delayed by d time units. As such, an expression specifies a device for computing a single output, at time t, which is dependent upon the inputs of signals, some of which could be as defined at an earlier time (cf: stream model of computation with delays).

Note that we could have included the 'and' primitive, either together with or instead of, the 'or' primitive which is adopted here. In fact, it is immaterial which is used as long as at least one such binary logical connective is included. In any of these cases, the corresponding semantics is easily added, as required.

Basic syntactic classes

$$\begin{array}{lcll} \mathrm{Var} & = & \{ x_0, x_1, x_2, \cdots \} & \text{(ranged over by v)} \\ \mathbb{N} & = & \{ 0, 1, 2, 3, \cdots \} & \text{(ranged over by t, d)} \end{array}$$

Operational classes

$$\begin{array}{lcll} \mathbb{B} & = & \{ 0, 1 \} \cup \{ ? \} & \text{(ranged over by b)} \\ \mathrm{State} & = & \mathrm{Var} \longrightarrow \mathbb{B} & \text{(ranged over by s)} \\ \mathrm{SigHist} & = & \mathrm{State}^{+} & \text{(ranged over by h)} \end{array}$$

where the class $\mathbb{B}$ is comprised of a two element type and a 'dont know' element denoted by '?'. The 'dont know' quantity is necessary in order to define those cases where the choice of a particular value from the two element type is not determined. Such a quantity cannot be tested for and may only be returned as a result. The use of '?' in this class follows the format of the 'dont know' value provided in the Ella language.
The state mapping (or *environment*), State, gives an instantaneous snapshot of the stable values associated with the variables that might occur in an expression. For this language of

$$\boxed{1a}\ \frac{}{h \vdash_t \text{high} \Rightarrow 1} \qquad \boxed{1b}\ \frac{}{h \vdash_t \text{low} \Rightarrow 0}$$

$$\boxed{2}\ \frac{0 \leq t < |\,h\,|\ ,\quad h_t(v) = b}{h \vdash_t v \Rightarrow b}$$

$$\boxed{3a}\ \frac{t < d}{h \vdash_t \Delta(e,d) \Rightarrow\ ?} \qquad \boxed{3b}\ \frac{d \leq t\ ,\quad h \vdash_{(t\text{-}d)} e \Rightarrow b}{h \vdash_t \Delta(e,d) \Rightarrow b}$$

$$\boxed{4a}\ \frac{h \vdash_t e \Rightarrow 0}{h \vdash_t \overline{e} \Rightarrow 1} \qquad \boxed{4b}\ \frac{h \vdash_t e \Rightarrow 1}{h \vdash_t \overline{e} \Rightarrow 0}$$

$$\boxed{5a}\ \frac{h \vdash_t e_1 \Rightarrow 1}{h \vdash_t e_1 + e_2 \Rightarrow 1} \qquad \boxed{5b}\ \frac{h \vdash_t e_2 \Rightarrow 1}{h \vdash_t e_1 + e_2 \Rightarrow 1}$$

$$\boxed{5c}\ \frac{h \vdash_t e_1 \Rightarrow 0\ ,\quad h \vdash_t e_2 \Rightarrow 0}{h \vdash_t e_1 + e_2 \Rightarrow 0}$$

$$\boxed{6}\ \frac{h \nvdash_t e \Rightarrow 0\ ,\quad h \nvdash_t e \Rightarrow 1}{h \vdash_t e \Rightarrow\ ?}$$

Figure 1: Semantic rules for the logic + delay language

'Logic + Delay', these variables are in effect standing for the input ports of the device represented by the expression they occur in.
The *signal history*, SigHist, records the history of all states for a particular execution. This may be adequately modelled as a (non empty) sequence of state mappings.
The operational semantics is expressed as a relation which maps configurations to results. So, the collection of *configurations* is defined to be:-

$$\text{Config} = \text{SigHist} \times \mathbb{N} \times \text{Expr}$$

and the relation defining execution takes the form:

$$(\_ \vdash_{\_} \_ \Rightarrow \_) \subseteq \text{Config} \times \mathbb{B}$$

where ($h \vdash_t e \Rightarrow b$) means that, at time t, the expression e is executed to produce value b, based upon the signal history h.

The rules in Figure 1 describe the execution relation for the Logic + Delay expression language. Note that rule 6 is a *meta level rule* because it refers to the possibility of **not** deriving some result, no matter how. In fact, it *forces* the completion of this set of rules defining the execution relation.

2.1 Stable delay primitives

We now change the rules concerning the delay (Δ) defined in the previous section to give it the semantics of a simplified form of basic delay element as provided in Ella. This is a form of *stable* delay whose output is a delayed version of the input, so long as it is stable for the delay period. If the signal varies during the delay period then the output should produce the 'don't know' value, ?.

The new rules representing the stable delay replace rules 3a and 3b by the pair of rules 3a' and 3b' given in Figure 2. Rule 3a' performs the same rôle as rule 3a. In the case that the expression e in $\Delta_s(e, d)$ is unstable during a period d, rule 6 forces the output to be ?. Furthermore, it can be noted that $\Delta(e,1) = \Delta_s(e,1)$ for arbitrary expressions e. This may be seen by observing that the conditions in rule 3b' directly reduce, when d = 1, to rule 3b.

$$\boxed{\text{3a'}}\ \frac{t < d}{h \vdash_t \Delta_s(e,d) \Rightarrow ?} \qquad \boxed{\text{3b'}}\ \frac{d \leq t\ ,\quad \bigwedge_{t\text{-}d \leq\ i < t} h \vdash_i e \Rightarrow b}{h \vdash_t \Delta_s(e,d) \Rightarrow b}$$

Figure 2: Ella "stable" delay rules

The full range of Ella delays have previously been investigated[8] and their semantics defined axiomatically in terms of predicate calculus. These predicates can be compared with the operational semantics as given above. The stable delay defined here represents an Ella ambiguity delay which only 'looks back' over the last 'd' units of time. If the delay predicates apply this restriction then they reduce to the same functionality as the stable delay. Thus although a stable delay, as described above, is not provided in the Ella language it does provide a basis from which the operational semantics of Ella delays can be constructed.

2.2 Simplified Case Statement

We now extend our language further by including a basic form of the Ella 'case' statement. The **case** statement we have chosen to start with only applies to two valued enumerated types and takes the form:

$$\textbf{case}\ \mathrm{e}_0\ \textbf{of}\ 1 : \mathrm{e}_1, 0 : \mathrm{e}_2$$

This **case** statement will form the basic building block for more advanced forms of Case statements found in the full Ella language. The expression occurring between the symbols **case** and **of** is known as the *chooser* expression. The syntax of expressions in the language now becomes:

Expressions, ($\mathrm{e} \in \mathrm{Expr}$) given by:-

$$\mathrm{e} ::= \mathrm{v} \mid \mathsf{high} \mid \mathsf{low} \mid \overline{\mathrm{e}_1} \mid \mathrm{e}_1 + \mathrm{e}_2 \mid \Delta_s(\mathrm{e}_1, \mathrm{d}) \mid \textbf{case}\ \mathrm{e}_0\ \textbf{of}\ 1 : \mathrm{e}_1, 0 : \mathrm{e}_2$$

The basic Syntactic and Operational classes remain unaltered but the Semantic rules have the additional rule for the **case** statement

$$\boxed{\text{7a}}\ \frac{\mathrm{h} \vdash_t \mathrm{e}_0 \Rightarrow 1\ ,\quad \mathrm{h} \vdash_t \mathrm{e}_1 \Rightarrow \mathrm{b}}{\mathrm{h} \vdash_t \textbf{case}\ \mathrm{e}_0\ \textbf{of}\ 1 : \mathrm{e}_1, 0 : \mathrm{e}_2\ \Rightarrow \mathrm{b}}$$

$$\boxed{\text{7b}}\ \frac{\mathrm{h} \vdash_t \mathrm{e}_0 \Rightarrow 0\ ,\quad \mathrm{h} \vdash_t \mathrm{e}_2 \Rightarrow \mathrm{b}}{\mathrm{h} \vdash_t \textbf{case}\ \mathrm{e}_0\ \textbf{of}\ 1 : \mathrm{e}_1, 0 : \mathrm{e}_2\ \Rightarrow \mathrm{b}}$$

$$\boxed{\text{7c}}\ \frac{\mathrm{h} \nvdash_t \mathrm{e}_0 \Rightarrow 0\ ,\quad \mathrm{h} \nvdash_t \mathrm{e}_0 \Rightarrow 1}{\mathrm{h} \vdash_t \textbf{case}\ \mathrm{e}_0\ \textbf{of}\ 1 : \mathrm{e}_1, 0 : \mathrm{e}_2\ \Rightarrow\ ?}$$

The rule 7c is necessary for completeness and describes what happens when the chooser signal is not determined. It should be noted that, of course, "?" cannot be tested for equality and hence no extra test condition is possible within the **case** statement.

3 Example calculation

An example calculation of the operational reduction relationship for the "Logic + Delay" language defined above, is given as follows.

Suppose that the signal history, h, has only one input signal, called a, which is to be defined over an interval of at least 5 time units in length (starting at time 0). So, the signal history, h, could be defined by the following table:

h =

a	0	1	0	1	1	⋯
	0	1	2	3	4	⋯

From this table, we can see that the signal a takes value 1 at time 3 and value 0 at time 0. Additional rows may be added if more signals are to be considered and more columns may be added if further time periods are needed.

Suppose that we now consider the following simple expression:

$$\Delta_s(\overline{\Delta_s(a,1)},1)$$

and that we wish to calculate its value at time 3, given signal history h. In terms of our operational semantics, this question corresponds to the following formal query:

$$\text{Find a value B such that } h \vdash_3 \Delta_s(\overline{\Delta_s(a,1)},1) \Rightarrow B$$

We now proceed to *calculate* an answer for B, by constructing a formal deduction using the rules that we have given for our Logic + Delay language, but working *backwards* from the query given above.

Because of the form of the query, the first (backwards) step has to be either rule 3a' or rule 3b'. But clearly an application of rule 3a' would reduce the resulting antecedent to false and so we should try rule 3b' instead. This gives:

$$\boxed{\text{3b'}}\ \frac{1 \leq 3\ ,\quad h \vdash_{3\text{-}1} \overline{\Delta_s(a,1)} \Rightarrow B}{h \vdash_3 \Delta_s(\overline{\Delta_s(a,1)},1) \Rightarrow B}$$

Now we have two sub-goals to solve, replacing our original goal:

$$1 \leq 3\ ,\quad h \vdash_{3\text{-}1} \overline{\Delta_s(a,1)} \Rightarrow B$$

However, the first sub-goal (i.e. $1 \leq 3$) reduces to true immediately (as it is a true proposition about arithmetic), leaving the second sub-goal to be solved.

To simplify our presentation (and the resulting proof trees!) we shall omit the simple arithmetical sub-goals from the proof trees and instead we shall say in the text whether they are true or not.

To solve the remaining subgoal, we can see from the rules that we will need to match either rule 4a or rule 4b. Unfortunately, we cannot determine which of these rules to apply since it involves knowing the value of B, which is the thing that we are trying to compute!

To get around this, we shall simply try out the rules in turn to find out if either of them works. So trying out rule 4a, we can see that this "assigns" value 1 to (meta) variable B:

$$
\begin{array}{rc}
 & h \vdash_2 \Delta_s(a,1) \Rightarrow 0 \\ \cline{2-2}
\boxed{4a} & h \vdash_2 \overline{\Delta_s(a,1)} \Rightarrow 1 \text{ (setting B = 1)} \\ \cline{2-2}
\boxed{3b'} & h \vdash_3 \Delta_s(\overline{\Delta_s(a,1)},1) \Rightarrow 1
\end{array}
$$

So we have now reduced our problem to checking another delay formula, which by another application of rule 3b', we obtain:

$$
\begin{array}{rc}
 & h \vdash_{2\text{-}1} a \Rightarrow 0 \\ \cline{2-2}
\boxed{3b'} & h \vdash_2 \Delta_s(a,1) \Rightarrow 0 \\ \cline{2-2}
\boxed{4a} & h \vdash_2 \overline{\Delta_s(a,1)} \Rightarrow 1 \\ \cline{2-2}
\boxed{3b'} & h \vdash_3 \Delta_s(\overline{\Delta_s(a,1)},1) \Rightarrow 1
\end{array}
$$

As before, we can reduce the arithmetic property $(1 \leq 2)$ to true, leaving us to evaluate the input signal a at time 1. This involves an attempt to apply rule 2 as follows:

$$
\begin{array}{rc}
 & 0 \leq 1 < |h| \ , \ \ h_1(a) = 0 \\ \cline{2-2}
\boxed{2} & h \vdash_1 a \Rightarrow 0 \\ \cline{2-2}
\boxed{3b'} & h \vdash_2 \Delta_s(a,1) \Rightarrow 0 \\ \cline{2-2}
\boxed{4a} & h \vdash_2 \overline{\Delta_s(a,1)} \Rightarrow 1 \\ \cline{2-2}
\boxed{3b'} & h \vdash_3 \Delta_s(\overline{\Delta_s(a,1)},1) \Rightarrow 1
\end{array}
$$

However, for the second arithmetical sub-goal, the definition of h shows that h satisfies $h_1(a) = 1$, which is not 0. Hence this sub-goal reduces to false. So we *backtrack* to the last place where we made a choice (i.e. selecting rule 4a) and then see if there is anything else that can be done. Clearly, we could have selected rule 4b instead and so trying that we get:

$$
\begin{array}{rc}
 & h \vdash_2 \Delta_s(a,1) \Rightarrow 1 \\ \cline{2-2}
\boxed{4b} & h \vdash_2 \overline{\Delta_s(a,1)} \Rightarrow 0 \text{ (setting B = 0)} \\ \cline{2-2}
\boxed{3b'} & h \vdash_3 \Delta_s(\overline{\Delta_s(a,1)},1) \Rightarrow 0
\end{array}
$$

and then proceeding much as we did before to get:

$$
\begin{array}{rc}
 & 0 \le 1 < |\,h\,| \;,\quad h_1(a) = 1 \\ \cline{2-2}
\boxed{2} & h \vdash_1 a \Rightarrow 1 \\ \cline{2-2}
\boxed{3b'} & h \vdash_2 \Delta_s(a,1) \Rightarrow 1 \\ \cline{2-2}
\boxed{4b} & h \vdash_2 \overline{\Delta_s(a,1)} \Rightarrow 0 \\ \cline{2-2}
\boxed{3b'} & h \vdash_3 \Delta_s(\overline{\Delta_s(a,1)},1) \Rightarrow 0
\end{array}
$$

After reducing the outstanding arithmetic facts, we obtain:

$$
\begin{array}{rc}
 & \\ \cline{2-2}
\boxed{2} & h \vdash_1 a \Rightarrow 1 \\ \cline{2-2}
\boxed{3b'} & h \vdash_2 \Delta_s(a,1) \Rightarrow 1 \\ \cline{2-2}
\boxed{4b} & h \vdash_2 \overline{\Delta_s(a,1)} \Rightarrow 0 \\ \cline{2-2}
\boxed{3b'} & h \vdash_3 \Delta_s(\overline{\Delta_s(a,1)},1) \Rightarrow 0
\end{array}
$$

so completing our example derivation.

4 Adding declarations

The next step is to enable internal points in the expression to be named and to allow the use of these names elsewhere. The extended language uses the expression syntax, Expr, that we have already built up and adds in two new syntax classes: Binding and Declaration set (or Decls).

A Binding sets a name to the value of an expression and a Declaration set is a complete set of Binding items that comprises an entire system. Since Decls is intended to be a “set” of items, we do not impose conditions such as “define before use”. Moreover, this also means that we can form loops, so providing systems with feedback. The full syntax of the extended language is given below:

Language Classes

Expressions, ($e \in \mathrm{Expr}$) given by:-

$$e ::= v \mid \mathsf{high} \mid \mathsf{low} \mid \overline{e_1} \mid e_1 + e_2 \mid \Delta_s(e_1, d) \mid \textbf{case}\ e_0\ \textbf{of}\ 1 : e_1, 0 : e_2$$

Bindings, ($bd \in \mathrm{Bind}$) given by:-

$$bd ::= v = e$$

Declaration set, ($ds \in \mathrm{Decls}$) given by:-

$$ds ::= \{\ bd_1, bd_2, \cdots, bd_n\ \}$$

So, a system is represented by a collection of Bindings called a Declaration set. Each Binding labels a particular expression with a (unique) name. References to this name then allow the value of the associated expression to be directly used elsewhere, so introducing the “fan out” of signals. Input ports correspond to names that are used in an expression but which are not defined in the set of declarations; output ports correspond to those names that are given bindings but which are left unused elsewhere.

The semantics of our extended language will involve the elaboration of a set of declarations. So, during a single time step, the signal environment is extended by the computed values for each of the declared identifiers. This process is repeated until stable values are obtained. This phase of calculation is then completed by overwriting the resulting signal environment with the new input traces.

As this language is a direct extension of the earlier “Logic + Delay” language, we shall accordingly omit those definitions already given and just give the additional clauses and definitions.

Additional operational classes

$$\begin{array}{lcll} \mathrm{Inputs} & = & \mathrm{Var} \longrightarrow \mathbb{B} & \text{(ranged over by I)} \\ \mathrm{Outputs} & = & \mathrm{Var} \longrightarrow \mathbb{B} & \text{(ranged over by O)} \end{array}$$

The collection of configurations, Config, remains as before. However, each state mapping in State records values associated with the identifiers which may occur in a given expression or which have been declared within a given declaration set. As such, identifiers stand for nodes within a circuit and they will label the inputs, outputs and internal nodes involving *fan-out*. As before, the signal histories, SigHist, record the sequence of state mappings arising during execution.

We shall require some way of elaborating declarations and so we introduce the following transition relation:

$$(\_ \xrightarrow{\_} \_) \subseteq (\mathrm{SigHist} \times \mathrm{Decls} \times \mathrm{SigHist})$$

where $(h \xrightarrow{ds} h')$ states that history h can be mapped to h' according to the declarations in the set ds.

Finally, we need a transition relation for extending a signal history by the next state, upon the application of the next inputs:

$$(\_ \underset{\_}{\overset{\_}{\leadsto}} \_) \subseteq (\mathrm{SigHist} \times \mathrm{Decls} \times \mathrm{Inputs} \times \mathrm{SigHist})$$

where $(h \underset{I}{\overset{ds}{\leadsto}} h')$ shows how, for the next inputs I, the history h maps to h', under the given declaration set ds. Assumptions concerning the treatment of the input are discussed below.

The inference rules for defining the above relations are given in Figure 3. The definitions of auxiliary functions such as the *history update* function, $(\_ \dagger\dagger \_) : ((A \xrightarrow{\sim} B)^* \times (A \xrightarrow{\sim} B)) \longrightarrow (A \xrightarrow{\sim} B)$ are given in the Glossary.

$$\boxed{\mathrm{d1}}\quad \frac{\bigwedge_{i \in 1..k} h \vdash_{|h|} e_i \Rightarrow b_i}{h \xrightarrow{\{ v_i = e_i \mid i \in 1..k \}} h \dagger\dagger \{ v_i \mapsto b_i \mid i \in 1..k \}}$$

$$\boxed{\mathrm{d2}}\quad \frac{\begin{array}{l} 1.\ h' = (h \frown \mathrm{last}(h)) \dagger\dagger I \\ 2.\ h' \xrightarrow{ds} \cdots \xrightarrow{ds} h'' \\ 3.\ h'' \xrightarrow{ds} h'' \end{array}}{h \underset{I}{\overset{ds}{\leadsto}} h''}$$

Figure 3: Rules for elaborating declarations

4.1 Explanation

We now discuss how the rules given in Figure 3 are designed to work. A basic intuition in operational semantics is to describe precisely how a computing system proceeds on a step-by-step basis. This means that we need to determine what the essential components that characterise the system behaviour in terms of transition relations. Clearly, an operational semantics could easily involve very complex transition relations but, in most cases, the complexity of these definitions can be mastered by using a combination of abstraction, structural design and modularity techniques.

We first of all note that the state of the system over time is defined by the values associated with certain identifiers standing for signal ports, as mentioned by the declaration set or by their use as input signals. Consequently, it is the signal history that characterises the behaviour of the specified device and accordingly, we identify signal history with the device's behaviour. The other determinants of step-by-step behaviour are clearly the set of declarations and the next vector of input signals. The transition relation ($\mathrm{h} \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\leadsto}} \mathrm{h}'$) describes this step-by-step evolution of the device's behaviour.

We also note that a system can potentially have signals which have fan-out, feed-forward and (delayed) feedback. Syntactically, this is permitted by the unrestricted use of a finite set of declarations which introduce portnames as identifiers. These identifiers may then occur within any of the expressions from a binding in the declaration set. This has some consequences for our transition relations that are now considered.

At this point, we should perhaps emphasise that the semantics merely requires the input signals not to change during the process of computing the next state. This might be implemented perhaps by clocked latching and then appealing to some (unspecified) clocking scheme, or by ensuring that the rate of change of the unlatched input is very much slower than the time taken to produce a stable state. Another possibility is that the input is latched but where the gating signal is derived not from an external clock but by detecting when the output has reached a stable value. The present semantics excludes none of these implementations of the 'input constancy' assumption.

So, suppose that we have a signal history, h, and we consider just one of the bindings, "id = e", from a given declaration set ds. By using the relation ($\mathrm{h} \vdash_{\mathrm{t}} \mathrm{e} \Rrightarrow \mathrm{b}$), we can work out the new value, b, for the portname id in the succeeding environment, h'. The transition relation ($\mathrm{h} \xrightarrow{\mathrm{ds}} \mathrm{h}'$) is defined by rule d1 to compute, simultaneously in parallel, all of the updates due to one of these evaluation stages. It is important to note that, by using the ($\_ \dagger\dagger \_$) operator, rule d1 only changes the *last* element of any signal history, leaving the remainder unchanged. This ensures that the past history remains unaltered, an intuitively obvious requirement!

The above shows how to perform one computation of new values for portnames from the previous values. But the values of the portnames could also have been used elsewhere in the circuit, during the present *epoch* or time period. Accordingly, it is necessary to *push* or propagate these values through again to see if there are any further changes in state. Eventually, this *ripple-through* effect caused by propagating new values will terminate (when the system is stable) and produce a signal history which is left unchanged, h'', by the updating process.

Because of this ripple-through effect of value propagation through a circuit, we therefore need to compute the *fixed point* of the transition relation, ($\_ \xrightarrow{\mathrm{ds}} \_$) that is reachable by iteration from the original signal history, h, but updated by the next input mapping. This propagation of values is achieved for a stable system by use of rule d2. If the system happens not to be stable, then rule d2 cannot be applied and no result is obtained.

By the above informal argument, we claim that the system could only reach a stable state precisely when such a fixed point exists. It is important to see that our semantics does *not* require any of the usual fixed point machinery that accompanies Denotational Semantics - we are only dealing with finite relations over discrete valued structures.

4.2 The Finite History Property

Because a system is completely specified by the signal history, the system apparently has an infinite state space, consisting of arbitrarily long sequences of state mappings. We shall informally show that this is not the case by arguing that the signal histories could always be truncated suitably so that they only contain a finitely bounded amount of information without affecting the behaviour of the system.

We first of all note that each state is made up of a finite mapping from portnames to a finite range of values, implying that the set of possible state mappings is finite. We next observe that each new assignment of value can only depend on the values of its sub-expressions after some finite delay. Since delays may be nested, they may also accumulate within expressions. Therefore, one may then compute the maximum possible delay occuring in each declaration expression and hence the maximum possible delay over all the expressions in a given declaration set.

Since in this model a delayed signal corresponds to a reference to a state variable within an earlier epoch, the above calculation suggests that we could truncate the signal history to a suffix whose length is greater than or equal to the maximum possible delay. This argument informally shows that we could replace the unbounded set of signal histories by a bounded set of *effective* signal histories, EffSigHist, each element of which has length bounded above by (1 + MaxDelay). Moreover this argument suggests that the cardinality of EffSigHist is bounded above by:

$$\begin{aligned} |\,\mathrm{EffSigHist}\,| &\leq (\text{No. of state mappings})^{(1\,+\,\mathrm{MaxDelay})} \\ &= (|\,\mathrm{Vals}\,|^{\,\mathrm{Sigs}})^{(1\,+\,\mathrm{MaxDelay})} \\ &= |\,\mathrm{Vals}\,|^{\,(\mathrm{Sigs}\,\cdot\,(1\,+\,\mathrm{MaxDelay}))} \end{aligned}$$

where:

$$\begin{aligned} \mathrm{Vals} &= \text{the set of possible values associated with the ports} \\ \mathrm{Sigs} &= \text{the total number of ports in the declaration set, ds} \\ \mathrm{MaxDelay} &= \text{the maximum possible delay over ds} \end{aligned}$$

It is relatively easy to give examples of systems whose state space meets this bound and where each system is such that the values of the parameters MaxDelay, Vals and Sigs can be freely chosen. For instance, specify a ring counter on every output pin which is additionally gated and reset by a logical combination of some delayed input signals. Examples of this kind suggest that the bound given above is tight.

However, what is perhaps more interesting from the point of view of verification is that this bound need *not* be attained in practice and that actual systems may possess many *fewer* states than the above bound would imply.

4.3 Loop checking and instability

The above discussion has shown that systems defined using our language of declarations are essentially finite-state systems and so have finite latency (or memory). Even so, such systems can be susceptible to instability. In this section, we shall carefully distinguish between *internal instability*, due to the failure to find a fixed point of value propagation within an epoch, and the possible lack of *output constancy* for a particular system. The first sort of behaviour is certainly an error and should never occur in a correct design. However, the second sort of behaviour may or may not be undesirable, depending upon the nature and specification of the system.

As these systems are essentially finite-state, this suggests that a decidable rule for internal instability could be formulated as already done above for the case of stable behaviour. One possible rule is given as follows:

$$\boxed{\mathsf{d3}}\ \frac{\begin{array}{l} 1.\ h' = (h \frown \mathrm{last}(h)) \dagger\dagger I \\ 2.\ h' \xrightarrow{ds} \cdots \xrightarrow{ds} h'' \\ 3.\ h'' \xrightarrow{ds} h''' \xrightarrow{ds} \cdots \xrightarrow{ds} h'' \\ 4.\ h'' \neq h''' \end{array}}{h \underset{I}{\overset{ds}{\leadsto}} (h \frown (\mathrm{UNDEF} \dagger I))}$$

where UNDEF is the state that maps all identifiers to '?'

Rule d3 works in much the same way as for rule d2, except that instead of detecting a fixed point, it detects a loop of size strictly greater than 1. It does this by finding a signal history, h'', that occurs more than once and with a successor, h''', that is different from h''.

The conclusion of rule d3 says that the resulting signal history is set to the original history, h, appended to by the state UNDEF, which maps all identifiers to '?' but then overwritten by the input mapping, I. The justification for this "all or nothing" approach is that, since at least one part of the state failed to stabilise, the fact that any other part (except the inputs) did succeed in stabilising is regarded as somewhat incidental, and so they may as well be treated as unstable.

From the point of view of tracing the causes of any instability, this could be determined by investigating the iterated behaviour of the relation ($\_ \xrightarrow{ds} \_$) and then observing in which state components the looping phenomena actually arise.

It is clear that the rules d2 and d3 are mutually exclusive since a fixed point (if it exists) is simply a loop of size 1 - and this excludes the third premise for rule d3. We also note that the relation ($\_ \xrightarrow{ds} \_$) is functional, for a fixed declaration set ds, since expression execution is functional and hence deterministic. Therefore, since the relation ($\_ \xrightarrow{ds} \_$) can only update signal histories by using a finite number of state mappings, there can only be a finite number of iterations before a loop occurs, i.e. at most $\mid \mathrm{Vals} \mid^{\mathrm{Sigs}}$ iterations, although in practice this limit may be significantly smaller.

4.4 The Output Constancy Property

As alluded to above, output constancy tries to capture the property that, if the inputs were held constant *during all succeeding epochs*, the outputs would not cyclicly change, but would *eventually* stabilise to some constant values. Such a property would certainly be required of, say, a multiplier circuit or other circuits for data and signal processing.

However, it is surprisingly not a mandatory requirement that all systems should have the property of output constancy. For instance, if the inputs had been *clocked* so that all major transitions of the system and it's sub-components were synchronised then the output constancy property would certainly not be required since each major transition would involve a fresh input sample. As we have not completely fixed our model of input change and it's relationship to other timing issues, we have not enforced this property within our semantics.

This is quite a subtle constraint to formalise since it involves looking ahead to check that this eventuality property holds over the evolving system behaviour, on the assumption that no further change in input would occur. In principle, this appears to involve infinite lookahead and could therefore be an undecidable property to check. However, by recourse to our observations concerning the Finite History property, we may argue that a finite bound to the lookahead can be determined in advance and so that, after all, it is decidable.

Before giving a more formal characterisation, we first motivate why a designer might want this property rather than some other similar sounding properties to hold. Suppose that we had simply required the property that if the input is held constant over successive epochs, then there should be no change in output (i.e. it remain constant). This clearly rules out oscillatory behaviour — but it also rules out much more as well, such as the behaviour for a finite buffer of latency $k > 0$. This could take at most k steps to produce a constant output, depending on what the previous k inputs had been. More generally, any output with non-zero latency would be declared illegal using the simplified rule.

So, to formalise our output stablity property, we introduce yet another transition relation:

$$(\_ \overset{\_}{\underset{\_}{\Rrightarrow}} \_) \subseteq (\mathrm{SigHist} \times \mathrm{Decls} \times \mathrm{Inputs} \times \mathrm{SigHist})$$

where $(\mathrm{h} \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\Rrightarrow}} \mathrm{h}')$ shows how, for the next input mapping I, the history h *stably* maps to h', under the given declaration set ds.

The rule os1 below defines $(\_ \overset{\_}{\underset{\_}{\Rrightarrow}} \_)$ in the case where the behaviour is stable:

$$\boxed{\mathrm{oc1}}\ \frac{\begin{array}{l} 1.\ \mathrm{h} \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\leadsto}} \mathrm{h}' \\[2ex] 2.\ \mathrm{h}' \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\leadsto}} \cdots \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\leadsto}} \mathrm{h}'' \\[2ex] 3.\ \text{for all } \mathrm{h}''' \bullet (\mathrm{h}'' \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\leadsto}} \cdots \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\leadsto}} \mathrm{h}''') \wedge (\mathrm{last}(\mathrm{h}'') = \mathrm{last}(\mathrm{h}''')) \end{array}}{\mathrm{h} \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\Rrightarrow}} \mathrm{h}'}$$

Notice how the relation $(\_ \overset{-}{\underset{-}{\Rrightarrow}} \_)$ is a sub-relation of $(\_ \overset{-}{\underset{-}{\rightsquigarrow}} \_)$ obtained by adding the lookahead conditions (2.) and (3.) as constraints. Essentially, condition (2.) finds a successor signal history h'' for h' so that, by condition (3.), every successor, h''', to h'' has the same end states. (The universal quantifier for h''' is necessary since negative occurrences of free variables like those only occuring (positively) in the premisses are otherwise implicitly existentially quantified.)

As for our previous discussions of constancy, we can also give a rule that covers the corresponding unstable case as well.

$$\boxed{\mathrm{oc2}}\quad \frac{\begin{array}{l} 1.\ \mathrm{h} \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\rightsquigarrow}} \mathrm{h}' \\[1ex] 2.\ \text{for all } \mathrm{h}'' \bullet (\, (\mathrm{h}' \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\rightsquigarrow}} \cdots \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\rightsquigarrow}} \mathrm{h}'') \Rightarrow \mathrm{h}'' \text{ non-constant } (\mathrm{ds}, \mathrm{I})\,) \end{array}}{\mathrm{h} \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\Rrightarrow}} (\mathrm{h} \frown (\mathrm{UNDEF} \dagger \mathrm{I}))}$$

where UNDEF is the state that maps all identifiers to '?'

The essence of this rule is that the second premiss corresponds to the *failure* of the second and third constraints for rule os1. The subsidiary relation:

$$(\_ \text{ non-constant } (\_, \_)) \subseteq (\mathrm{Decls} \times \mathrm{Inputs} \times \mathrm{SigHist})$$

is defined by the following rule:

$$\boxed{\mathrm{u1}}\quad \frac{(\mathrm{h} \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\rightsquigarrow}} \cdots \overset{\mathrm{ds}}{\underset{\mathrm{I}}{\rightsquigarrow}} \mathrm{h}')\,, \quad \mathrm{last}(\mathrm{h}) \neq \mathrm{last}(\mathrm{h}')}{\mathrm{h} \text{ non-constant } (\mathrm{ds}, \mathrm{I})}$$

This predicate says that, eventually, there exists a successor state history, h', to h which, for constant input I, has a different end state.

4.5 Behavioural equivalence

We shall briefly mention a *behavioural equivalence* that is naturally associated with the transition relation $(\_ \overset{-}{\underset{-}{\rightsquigarrow}} \_)$. This can be motivated by saying that two declaration sets are *equivalent* if each has the same underlying behaviour, which is to say that the transition relation $(\_ \overset{-}{\underset{-}{\rightsquigarrow}} \_)$ for either declaration set will accept and reject identical signal histories for all inputs. This may be phrased more precisely in mathematical terms below. Let ds_0, ds_1 be arbitrary declaration sets, then we can define:

$$(\mathrm{ds}_0 \equiv_{\mathrm{beh}} \mathrm{ds}_1) \quad \Leftrightarrow \quad \forall\, \mathrm{I} \in \mathrm{Inputs}, \mathrm{h}, \mathrm{h}' \in \mathrm{SigHist} \bullet (\mathrm{h} \overset{\mathrm{ds}_0}{\underset{\mathrm{I}}{\rightsquigarrow}} \mathrm{h}') \Leftrightarrow (\mathrm{h} \overset{\mathrm{ds}_1}{\underset{\mathrm{I}}{\rightsquigarrow}} \mathrm{h}')$$

This is obviously a reflexive and symmetric relation and transitivity follows from the transitivity of $\Leftrightarrow$ and the use of universal quantification to ensure that the transition relations for each declaration set are compared over all inputs and signal histories.

Taking this definition and our previous analysis concerning the Finite History property leads us to conclude that our behavioural equivalence is *decidable*. This is because, without loss of generality, we can restrict the Inputs set to the finite set of mappings defining values solely for the input ports of each declaration set. Also, we can use the numerical bounds given previously to restrict the set of signal histories to the set of effective signal histories, EffSigHist, another finite set. Hence, we have eliminated the potentially infinite quantifications on favour of finite ones which, in principle, could be directly expanded into a finite, logical conjunction of tests.

As may be readily appreciated, the above formulation of behavioural equivalence is not amenable to efficient computation. It is an objective of continuing research into state machine verification to find more efficient and computationally tractable techniques for the inference of properties like the equivalence defined above.

5 A Flip-Flop Example

In this section we demonstrate how these rules would operate in principle, by examining how a simple Flip-Flop circuit would behave according to our operational semantics. This example is clearly relevant to the semantics study as it demonstrates that even this very small language possesses the capability of memory. As remarked in the Ella tutorial[3], the Ella RAM primitive could therefore be constructed out of elements such as these. However, for obvious pragmatic reasons. this would be cumbersome and inefficient for simulation and fabrication purposes. Hence, RAM memory is added as a primitive on the assumption that it would be implemented by a collection of custom memory chips.

The definitions that we shall use are as follows:

$$\{\ c = \Delta_s(\overline{d + a}, 1),\ d = \Delta_s(\overline{c + b}, 1)\ \}$$

and for convenience, we shall refer to this set by the name ff.

From the above, we shall see that any state mapping for ff must contain value assignments for at least the portnames a, b, c and d. Also, the ports a and b occur in a binding expression but they are not themselves bound and hence they must be input ports. Although strictly we should introduce additional ports for outputs, we shall use the ports c and d as outputs even though they are also used internally in the above declaration set.

It turns out that the deduction of behaviour for even this small system contains considerable detailed formulae which become extremely unwieldy and hard to present in a useful and informative way. Instead, we shall simply demonstrate how these rules could be used informally to construct in an evolutionary way a signal history corresponding to a particular sequence of inputs for a and b.

To do this, we shall start off with the completely undefined state as the first element in our initial signal history:

$$h_0 = \begin{array}{|c|c|} \hline a & ? \\ b & ? \\ c & ? \\ d & ? \\ \hline & 0 \\ \hline \end{array}$$

and then show how this can be extended to successive signal histories by applying further inputs to a and b.

Suppose that the following inputs are initially applied:

$$I_1 = \begin{array}{|c|c|} \hline a & 0 \\ b & 1 \\ \hline \end{array}$$

To calculate h_1, the immediate extension of h_0, we shall need to deduce that the relation $(h_0 \overset{\mathsf{ff}}{\underset{I_1}{\leadsto}} h_1)$ holds. So, if we are to be successful in deriving a value for h_1, we shall have to apply rule d2 which therefore involves satisfying it's three premisses.

To satisfy the first premiss of rule d2, we need to calculate an initial approximation to the extension of the signal history:

$$h_0{}' = \begin{array}{|c|cc|} \hline a & ? & 0 \\ b & ? & 1 \\ c & ? & ? \\ d & ? & ? \\ \hline & 0 & 1 \\ \hline \end{array} = (h_0 \frown \mathrm{last}(h_0)) \dagger\!\dagger\, I_1$$

The second premiss of rule d2 says to sucessively calculate the relation ($\_ \xrightarrow{\mathrm{ff}} \_$), starting from $h_0{}'$, until a signal history $h_0{}''$ is reached which just happens to satisfy the third premiss of rule d2.

To do this successive calculation of the relation ($\_ \xrightarrow{\mathrm{ff}} \_$), we need to apply rule d1 repeatedly until the third premiss of rule d2 is achieved. This calculation will be recorded by simply noting the intermediate signal histories, ih_0, ih_1, etc. that are related via ($\_ \xrightarrow{\mathrm{ff}} \_$).

Also, to cut down on inessential complexity, we shall omit any explicit calculation of expression value from the rules and instead simply quote the end result. Of course, the interested reader may check any expression evaluation for themselves by using the rules as stated earlier.

So, proceeding, we first of all set ih_0 to $h_0{}'$, and then using rule d1 we can show that ($ih_0 \xrightarrow{\mathrm{ff}} ih_1$) holds where:

$$ih_1 = \begin{array}{|c|cc|} \hline a & ? & 0 \\ b & ? & 1 \\ c & ? & ? \\ d & ? & ? \\ \hline & 0 & 1 \\ \hline \end{array}$$

because the unit delays in the bindings for c and d, in effect, forces the processes that define values c and d to execute one epoch behind. Therefore, $ih_0 = ih_1$ and so this signal history satisfies both the second and third premisses of rule d2.

So, by using rule d2 with the above data, we have shown that $h_0 \underset{I_1}{\overset{\mathrm{ff}}{\leadsto}} h_1$ holds where:

$$h_1 = \begin{array}{|c|cc|} \hline a & ? & 0 \\ b & ? & 1 \\ c & ? & ? \\ d & ? & ? \\ \hline & 0 & 1 \\ \hline \end{array}$$

To continue, suppose that the next inputs to be supplied are:

$$I_2 = I_1 = \begin{array}{|c|c|} \hline a & 0 \\ b & 1 \\ \hline \end{array}$$

As before, we need to apply rule d2 to calculate a stable successor, and so proceeding, we get:

$$h_1{}' = \begin{array}{|c|ccc|} \hline a & ? & 0 & 0 \\ b & ? & 1 & 1 \\ c & ? & ? & ? \\ d & ? & ? & ? \\ \hline & 0 & 1 & 2 \\ \hline \end{array} = (h_1 \frown \mathrm{last}(h_1)) \dagger\!\dagger\, I_2$$

as the initial approximation for h_2, the immediate extension of h_1.

Again, we need to successively compute the relation $(\_ \xrightarrow{\mathsf{ff}} \_)$ in much the same way as before, but this time starting from $h_1{}'$, which we accordingly set equal to ih_2.

So, after applying rule d1 and the rules for expression execution, this would show that $(ih_2 \xrightarrow{\mathsf{ff}} ih_3)$ holds where:

$$ih_3 = \begin{array}{|c|ccc|}\hline a & ? & 0 & 0 \\ b & ? & 1 & 1 \\ c & ? & ? & ? \\ d & ? & ? & 0 \\ \hline & 0 & 1 & 2 \\ \hline\end{array}$$

Interestingly, because our 'or' operator, $(\_ + \_)$, is non-strict with respect to '?' (see figure 1), we are able to find a defined value for d but not for c. Specifically, this is because the value of (? + 1) is equal to 1 according to our rules. On the other hand, we can see that, using rule 6, the value of (? + 0) is equal to ?, which is not 0.

So, since ih_2 is not equal to ih_3, we need to apply rule d1 again to show that $(ih_4 \xrightarrow{\mathsf{ff}} ih_4)$ where:

$$ih_4 = \begin{array}{|c|ccc|}\hline a & ? & 0 & 0 \\ b & ? & 1 & 1 \\ c & ? & ? & ? \\ d & ? & ? & 0 \\ \hline & 0 & 1 & 2 \\ \hline\end{array}$$

Now, since $ih_3 = ih_4$, we can again satisfy all three premisses in order to apply rule d2. This shows that $h_1 \overset{\mathsf{ff}}{\underset{I_1}{\leadsto}} h_2$ where:

$$h_2 = \begin{array}{|c|ccc|}\hline a & ? & 0 & 0 \\ b & ? & 1 & 1 \\ c & ? & ? & ? \\ d & ? & ? & 0 \\ \hline & 0 & 1 & 2 \\ \hline\end{array}$$

We can continue on in this fashion, building up the signal history based upon given inputs. For each epoch. we will need to successfully apply rule d2 and so satisfy it's three premisses. To do so, rule d1 will be needed to successively compute results until a fix-point is reached. This fix-point is then taken as the extended signal history.

Proceeding with our calculations, we see that by applying the following input *sequence*:

$$I_{3..8} = \begin{array}{|c|cccccc|}\hline a & 0 & 1 & 1 & 1 & 1 & 0 \\ b & 1 & 0 & 0 & 1 & 1 & 1 \\ \hline & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline\end{array}$$

the following signal history would then result:

$$h_8 = \begin{array}{|c|ccccccccc|}
\hline
a & ? & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
b & ? & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\
c & ? & ? & ? & 1 & 1 & 0 & 0 & 0 & 0 \\
d & ? & ? & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
\hline
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\hline
\end{array}$$

by application of rules d2 and d1.

6 Future Work

This work will provide a starting point and foundation for the work on Ella semantics and thereafter the work on Ella verification. One way in which this work might proceed is as follows:

- definition of the Core Ella abstract syntax, possibly based on the ‘interpreter modes’ used by the simulator
- definition of well-formedness relations for Core Ella, including type structures and other structural constraints
- definition of configuration structures and semantic value sets for representing device behaviour
- definition of semantic transition relations based on configuration definitions
- definition of state machine correspondence and associated behavioural equivalence relations

In addition, further consideration must also be given to techniques for symbolically exploring the consequences of these definitions as a way of checking their correspondence to the existing ‘operational’ meaning of Ella as encoded within the simulator. To this end, it would be beneficial to have a direct symbolic animation of processes of calculation similar to those described in this deliverable.

7 Conclusions

In this deliverable we have demonstrated the use of operational semantic techniques to formalise the execution behaviour of a simplified Ella-like language. Examples have been given of how a semantics could be used directly in deducing properties of systems written using this language. This work provides a basis for a more refined and detailed semantic study of the complete Ella language and so prepares the way for the development of state-machine based verification techniques for application to Ella.

Our thanks to John Morison (RSRE) and to Graham Gough (Univ. of Manchester) for their comments and observations on this work. Alan Williams (Univ. of Manchester) also made various helpful remarks.

Ella$^{\text{TM}}$ is a registered Trade Mark of the Secretary of State for Defence, and winner of a 1989 Queen’s Award for Technological Achievement.

A A glossary of semantics notation

A.1 General concepts

Configuration : A *configuration* is a collection of objects whose purpose is to give an instantaneous description of the system of interest.

Transition Relation : A *transition relation* is a relation, often binary, whose purpose is to relate the present configuration of a system to some next configuration of the system (there may be none, one or many). This relation may also involve additional parameters to restrict or constrain the evolution of the system in some fashion.

Operational Semantics : An *operational semantics* for a formal language L, is a mathematical model of the process of execution. Modern operational definitions usually comprise a system of relations defined by means of inference rules, defined in a *syntax directed* manner.

Concrete syntax : A *concrete syntax* is a set of productions which determines a formal language in terms of a set of strings.

Abstract syntax : An *abstract syntax* is a set of equations which determines a formal language in terms of sets of (labelled) tree objects.

Composite syntax : A *composite syntax* is, formally, a set of equations decorated with syntactic markers. From such a set, it is possible to extract both a concrete and an abstract syntax. A composite syntax is usually feasible when the structure of both concrete and abstract syntax correspond very closely.

Inference Rule : An *inference rule* represents an explicit method of deducing a conclusion from a finite number of premisses:

$$\boxed{\textit{rulename}}\frac{\text{Premiss}_1\ ,\quad \cdots\ ,\quad \text{Premiss}_k}{\text{Conclusion}}$$

As such, each inference rule acts like a conditional transformation of propositions so that if all premisses above the line are satisfied, then the conclusion below the line is also satisfied. Clearly, this represents conditional reasoning of the 'if' — 'then' — type.

The transition relations are often defined by using a table of inference rules whose conclusions match the relation being defined. Since there could be many relations of appropriate structure that satisfy these rules, we take the *least* such relation, whenever it exists, as the relation determined by the rules.

Meta notation in inference rules :

We have used various *meta* notations in an attempt to keep the size and number of rules down to a manageable and comprehensible collection. Such notation is explained briefly below:

A premiss of the form:- (h $\nvdash_t$ e $\Rightarrow$ b)

This simply means that the relation (h $\vdash_t$ e $\Rightarrow$ b) does *not* hold.

A premiss of the form:- $\bigwedge_{i \in S} \Phi[i]$

This is an abbreviation of the (finite) collection of premisses:

$$\Phi[i/a], \Phi[i/b], \cdots, \Phi[i/n]$$

where S = { a, b, $\cdots$, n } .

The informal notation Φ[i] indicates that formula Φ freely depends upon the parameter named i, and Φ[i/e] means uniformly substitute the expression e freely for the parameter i throughout the formula Φ.

A premiss of the form:- for all h $\bullet$ Φ[h]

This states the proposition that every premiss of the form:

$$\Phi[h/s]$$

will always hold, where s is any value having the same type as h. Alternatively, this says that no value of s could ever be substituted for h which made the formula Φ[h/s] not hold.

A premiss of the form:- $(a \xrightarrow{x} \cdots \xrightarrow{x} b)$:

This is an abbreviation for:

Either $(a \xrightarrow{x} b)$

Or $\exists\ n \geq 1 \bullet \exists\ c_1, \cdots, c_n \bullet$

1. $(a \xrightarrow{x} c_1)$
2. $\forall\ 1 \leq i < n \bullet (c_i \xrightarrow{x} c_{i+1})$
3. $(c_n \xrightarrow{x} b)$

This notation avoids introducing an auxiliary relation symbol for the ‘transitive closure’ of relations like $(\_ \xrightarrow{\_} \_)$.

A.2 General Mathematical Notation and Auxiliary Definitions

Set operators : For any sets A, B, and $a_0, \cdots, a_n \in A$:

$\{ a_0, a_1, \cdots, a_k \}$	finite set of elements from A
$\{ e[v] \in A \mid P[v] \}$	Set comprehension: subset of A determined by predicate P and expression e
$A \cup B$	union of A with B
$A \cap B$	intersection of A with B
$A \times B$	Cartesian product of A with B
$A \triangle B$	symmetric difference of A and B: $= (A \cup B) - (A \cap B)$
A^*	sequences of elements in A, but possibly empty
A^+	non-empty sequences of elements in A
$A \longrightarrow B$	functions from A to B
$A \xrightarrow{\sim} B$	partial functions from A to B

Sequence operators : For any sequences $l_1, l_2 \in A^*$, $a_0, a_1, \cdots, a_k \in A$:

$[a_0, a_1, \cdots, a_k]$	finite sequence of elements
$l_1 \frown l_2$	list l_1 appended with list l_2
head(l)	first element of (non-empty) sequence l
tail(l)	all but first element of (non-empty) sequence l
last(l)	last element of (non-empty) sequence l
pref(l)	all but last element of (non-empty) sequence l

Partial Function operators :

For partial function $f \in (A \xrightarrow{\sim} B)$, $a_0, \cdots, a_n \in A$, $b_0, \cdots, b_n \in B$:

$\{ a_0 \mapsto b_0 , \cdots, a_n \mapsto b_n \}$	finite mapping in $(A \xrightarrow{\sim} B)$
$\{ a \mapsto e[a] \mid P[a] \}$	Function comprehension: function with domain determined by predicate P and result value $e[v] \in B$
$\partial_0(f)$	domain of function f (also dom(f))
$\partial_1(f)$	range of function f (also ran(f))

State update : $(\_ \dagger \_) : (A \xrightarrow{\sim} B)^2 \longrightarrow (A \xrightarrow{\sim} B)$

For any partial functions $f, g \in (A \xrightarrow{\sim} B)$, and $a \in A$:

(a) g is defined at a $\Rightarrow$ $(f \dagger g)(a) = g(a)$

(b) g is not defined at a
but f is defined at a $\Rightarrow$ $(f \dagger g)(a) = f(a)$

(c) neither f nor g is defined at a $\Rightarrow$ $(f \dagger g)$ is not defined at a

History update : $(\_ \dagger\dagger \_) : ((A \xrightarrow{\sim} B)^* \times (A \xrightarrow{\sim} B)) \longrightarrow (A \xrightarrow{\sim} B)$

For any $f, m \in (A \xrightarrow{\sim} B)$, $s \in (A \xrightarrow{\sim} B)^*$:

(a) $([\ ] \dagger\dagger m) = [m]$
(b) $s \neq [\ ] \Rightarrow (s \dagger\dagger m) = pref(s) \frown (last(s) \dagger m)$

Functional Difference : $diff(\_,\_) : (A \xrightarrow{\sim} B)^2 \longrightarrow \mathbb{P}A$

For any partial functions $f, g \in (A \xrightarrow{\sim} B)$:

$$diff(f,g) = \{ a \in A \mid \text{dis-similar}(f,g,a) \}$$

where:

$$\begin{aligned} &\text{dis-similar}(f,g,a) \Leftrightarrow \\ &\qquad (a \in (\partial_0(f) \triangle \partial_0(g))) \\ &\quad \vee \quad (a \in (\partial_0(f) \cap \partial_0(g)) \wedge f(a) \neq g(a)) \end{aligned}$$

References

[1] R. Dettmer. Ella - a language for VLSI. *Electronics and Power*, pages 517–522, July 1986.

[2] 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.

[3] Praxis Electronic Design, 20, Manvers Street, Bath, Avon, BA1 1PX, United Kingdom. *The Ella User Manual*, 4.0th edition, 1990.

[4] G. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Sept 1981.

[5] M. Hennessy. *The Semantics of Programming Languages*. John Wiley & Sons, 1990.

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

[7] M.B. Davies. Mathematical equivalence in a primitive Ella. Memorandum Number 4225, RSRE, 1988.

[8] M.G. Hill and E.V. Whiting. Ella timing. Technical Report D3.3/RSRE/3, ESPRIT project 2260 (SPRITE), 1989.