Verifying Deterministic Systems using a State Evolution method*

Howard Barringer   Graham Gough   Brian Monahan[†]   Alan Williams [‡]

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

14th February 1994

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

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

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

1 Introduction and Context

One approach used to show that two hardware systems are behaviourally equivalent is to build process algebraic representations of the systems and then to establish a bisimulation relation between the two process terms. The approach suffers, however, from a rapid increase in complexity as the problem size grows, and of course breaks down for systems with infinite data-spaces.

In this paper, we present a *state bisimulation* rule which is provably equivalent to standard (strong) bisimilarity, but which operates on abstract process terms. An accompanying *state evolution* method generates a set of verification conditions needed to establish state bisimilarity. The verification conditions take a regular form involving case analyses, and can be established using a theorem prover for first order logic. The approach described offers the possibility of containing the usual growth in complexity of verification, whilst maintaining a high degree of automation. Indeed it offers the possiblity of verifying *bounded diameter* systems, which may have *unbounded* state-spaces; such systems cannot of course be analysed by state-based verification methods, even using encodings of the state-space[BCM$^+$90, BCL91], without first applying reduction transformations to the systems. The approach can also be extended to non-deterministic systems.

The state bisimulation approach operates for deterministic, synchronous hardware systems. Here, they are represented using a special process algebra called *EPA*[1]. It has similarities with SCCS-style process algebras, but offers distinct features which lead to compact representations of hardware systems. These can be formulated into a *normal form* for *EPA*, which is suitable for analysis using state bisimulation. For the purposes of this paper, systems are represented by state-machine description.

Background to the work presented here is obviously to be found in Milner's work on CCS and bisimilarity[Mil89]. More recently, Hennessy[HL93] has developed techniques for modelling infinite data-spaces and constructing their bisimulations; the VPAM system[Lin93] is somewhat similar to our approach of proving conditions which are generated when constructing a state bisimulation. Work on *EPA* and its development can be found in [BGMW94, BGMW93].

This paper has the following structure: in section 2 we define a state bisimulation for deterministic, synchronous systems. In section 3 the *state evolution* method is presented, a technique for determining whether two systems are state bisimilar; section 4 contains a small example of this. Section 5 then gives the state evolution rule, and section 6 shows its soundness with respect to strong bisimilarity.

2 Processes, States and Bisimulation

Our principal concern in this paper is with the verification of deterministic systems. However, we first need to introduce notation used to describe the basic concepts required.

2.1 Input Output Labelled Transitions systems

An *input output labelled transition system*, T, is a tuple $\langle Prc, IOAct, Trans\rangle$ where Prc is a set of process objects, $IOAct \subseteq (I \times O)$ is a set of input/output actions, and finally $Trans \subseteq (Prc \times IOAct \times Prc)$ is the transition relation for the system. The only unconventional aspect of this are the input/output actions —

[1] *EPA* was developed for the symbolic representation of hardware systems described by the hardware description language, ELLA, marketed by R-Cube Systems, Bristol, England, and also available in the public domain.

in the systems of interest, each action is composed of an input and an output of values. As such, input and output are considered to be *contemporaneous* actions.

The set of transitions *Trans* describes how process $p \in Prc$ evolves into another process $q \in Prc$ by making an input/output transaction (x/y) for some $x \in I,\ y \in O$. If T is a transition system, then transactions are conventionally written as:

$$T \vdash (p \xrightarrow{x/y} q)$$

We shall take our primitive equivalence on processes to be the *strong bisimilarity* of Park and Milner. We fix the transition system T and let $SB \subseteq Prc^2$.

We say that relation SB is a (strong) bisimulation (wrt. T) if and only if:

- SB is symmetric.
- $\forall\, p,\ q \in Prc \bullet \langle p,q \rangle \in SB \ \Rightarrow$
  $\forall\, \alpha \in IOAct,\ p' \in P \bullet T \vdash (p \xrightarrow{\alpha} p') \ \Rightarrow$
  $\exists\, q' \in Prc \bullet T \vdash (q \xrightarrow{\alpha} q') \ \wedge\ \langle p',q' \rangle \in SB$

We say that two processes $p,\ q \in Prc$ are *bisimilar* (wrt. T), denoted as $T \vdash p \sim q$ whenever there exists a strong bisimulation $SB \subseteq P^2$ (wrt. T) such that $\langle p,q \rangle \in SB$.

We say that a transition system T is *input deterministic* whenever we have:

$$\begin{aligned}&\forall\, p,\ q_1,\ q_2 \in Prc\ \forall\, i \in I\ \forall\, o_1,\ o_2 \in O \bullet\\ &\quad T \vdash (p \xrightarrow{i/o_1} q_1) \ \wedge\ T \vdash (p \xrightarrow{i/o_2} q_2) \ \Rightarrow\ (o_1 = o_2) \ \wedge\ (q_1 \sim q_2)\end{aligned}$$

This requires that for a given process p and a particular input i then the output is *determined* for that transition. Note that the processes resulting from transitions do not have to be identical, but simply bisimilar.

Finally, we say that T is *input total* whenever:

$$\begin{array}{l} \forall\, p \in Prc,\ i \in I \bullet \\ \quad \exists\, q \in Prc,\ o \in O \bullet \\ \quad\quad T \vdash (p \xrightarrow{i/o} q) \end{array}$$

2.2 Machine Presentations : Deterministic Normal Form

We now concern ourselves mainly with deterministic systems in a convenient normal form defined below.

For definiteness, let I, O be fixed sets for inputs and output values — they need not be disjoint. To describe what our normal form is, we shall introduce the set of *deterministic machines* with inputs I, outputs O and state-space St to be:

$$DM(I,O,St) \;\underline{\triangle}\; (Init(St) \times RespFn(I,O,St) \times EvolFn(I,O,St))$$

where we have the following sets:

$$\begin{array}{lcll} Init(St) & \underline{\triangle} & St & \texttt{--- initial states} \\ RespFn(I,O,St) & \underline{\triangle} & (I \times St) \to O & \texttt{--- response functions} \\ EvolFn(I,O,St) & \underline{\triangle} & (I \times St) \to St & \texttt{--- evolution functions} \end{array}$$

Accordingly, a *deterministic machine in normal form* is simply a point in the space $DM(I,O,St)$, for appropriate input/output sets I, O and state-space set St.

A deterministic machine $D \in DM(I,O,St)$ is related to the notions of transition presented above in the following way. Given that $D = \langle i_D, r_D, e_D \rangle$, define the transition system $T_D = \langle Prc_D, IOAct_D, Trans_D \rangle$ as follows:

$$\begin{array}{rcl} Prc_D & \underline{\triangle} & \{\, pD(\sigma_D) \mid \sigma_D \in St_D \,\} \\ IOAct_D & \underline{\triangle} & (I \times O) \end{array}$$

and the transitions are defined by the property:

$$\forall\, x \in I, \sigma_D \in St_D \bullet (pD(\sigma_D) \xrightarrow{(x/r_D(x,\sigma_D))} pD(e_D(x,\sigma_D))) \in Trans_D$$

where pD is a suitably injective mapping from the state space into process objects. We have the following lemma:

Lemma 2.1 : Determinism
Let $D \in DM(I,O,St)$. Then the induced transition system, T_D, corresponding to D is both input deterministic and input total. **Proof :** (omitted in this deliverable)

Clearly, we can now use the correspondance to induce a corresponding semantic equivalence on the space of all deterministic systems in normal form.

Definition 2.2 : Bisimilarity of deterministic systems
Let $P = \langle i_P,\ r_P,\ e_P \rangle \in DM(I_1,O_1,St_P)$ and $Q = \langle i_Q,\ r_Q,\ e_Q \rangle \in DM(I_2,O_2,St_Q)$. We then say that, relative to the (disjoint union of) transition systems $(T_P \cup T_Q)$:

$$(P \sim_{DM} Q) \quad \Leftrightarrow \quad (pP(i_P) \sim pQ(i_Q))$$

where pP, pQ are the injective mappings from appropriate state spaces to processes used to form the transition systems T_P, T_Q.

From now on, we shall be generally concerned with systems $P,\ Q$ of interest which we assume to be *presented* in the appropriate normal form:

$$P = \langle i_P,\ r_P,\ e_P\rangle \in DM(I,O,St_P)$$
$$Q = \langle i_Q,\ r_Q,\ e_Q\rangle \in DM(I,O,St_Q)$$

2.3 Iterated step and response operators

We can define the following useful families of functions, assuming that $M : DM(I,O,St)$, with $M = \langle i,r,e\rangle$. Let $k \geq 0,\ n \geq 1$.

$$step_k : DM(I,O,St) \to I^k \times St \to St$$
$$step_k(M)(x_k,\ \cdots\ x_1, \sigma) = foldl(e)(\sigma)[x_k,\ \cdots\ x_1]$$

$$resp_n : DM(I,O,St) \to I^n \times St \to O$$
$$resp_{k+1}(M)(x_{k+1},\ x_k,\ \cdots\ x_1, \sigma) = r(x_{k+1},\ step_k(x_k,\ \cdots\ x_1, \sigma))$$

where $foldl : (\alpha \times \beta \to \beta) \to \beta \to \alpha^* \to \beta$ is as defined in functional programming studies (e.g. [BW88]).

By use of properties of *foldl* proven by induction, we can see the following general result:

For all $u,\ v \geq 0$, we have that:

$$step_u(M)(a_u,\ \cdots\ a_1,\ (step_v(M)(b_v,\ \cdots\ b_1, \sigma))) =$$
$$step_{u+v}(M)(a_u,\ \cdots\ a_1, b_v,\ \cdots\ b_1, \sigma)$$

from which the following *step translation* property is obtained:

$$step_{k+1}(M)(x_{k+1},\ \cdots\ x_1, \sigma)$$
$$= e(x_{k+1}, step_k(M)(x_k,\ \cdots\ x_1, \sigma)) =$$
$$= step_k(M)(x_{k+1},\ \cdots\ x_2, e(x_1, \sigma)).$$

This property is used to establish the translation properties of the response formulae (see section 6).

2.4 State Bisimulation

We now describe a useful reformulation of Bisimulation equivalence which directly exploits the notion of state configuration appearing in Deterministic Normal Form. The purpose of these configurations is to summarise the evolution so far achieved by these deterministic systems. In fact, these state configurations arise here as a consequence of using *memory* to enable past input events to influence present and future outputs and succeeding states. As such, functions are treated purely as relationships between data values. This has the advantage that state information can be cleanly represented in a declarative form that is both definitive and flexible.

Let $SB \subseteq (St_P \times St_Q)$ be a binary relation between the *state spaces* of P and Q.

Definition 2.3 : State bisimulation
SB is a state bisimulation for deterministic processes $P,\ Q$ when:

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

This is clearly related to a similar notion defined by Olderog appearing in [Old91]. The difference between the approaches is that we do not wish to immediately identify the concept of "state" with that of "process" as directly as Olderog does. Instead, we shall *prove* that there is the desired association of state with process, in the case of deterministic systems:

Theorem 2.4 : Correctness of state bisimulation
Let $P,\ Q$ be deterministic systems in normal form.

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

Proof : (omitted in this deliverable)

3 State Evolution method

We now describe a technique which can be used to determine if two deterministic systems are state bisimilar and hence behaviourally equivalent.

The basic idea is to use the symbolic evolution of these systems to construct (in effect) a first-order formula whose validity would ensure that the systems were bisimilar. The procedure requires that the systems are presented in normal form. This means that formulae defining the initial state and the response and evolution functions for each system can be taken as given.

We shall first describe the method in a brief, informal way in order that the reader may gain some familiarity. This is then followed by a more precise description in terms of psuedocode. In the next section, a short example is then given illustrating how the method works.

3.1 Informal description

We assume that we are dealing with normalised, deterministic systems P and Q as in previous sections. The procedure begins by taking an *arbitrary*, but general point, $\langle \sigma_P,\ \sigma_Q\rangle$, in the product state space. $(St_P \times St_Q)$. As we know nothing about this point to start with, we can simply name it (or rather its components) using appropriate symbolic variables.

We then proceed to symboliclly evolve both of the systems on the working assumption that, for all inputs, the responses produced by each system should be in complete agreement at each stage. The idea now is to *either* show that, for some input, the responses clearly disagree at some stage (i.e. P and Q

are different) *or* to demonstrate that our symbolic simulation is, in some sense, ‘sufficiently complete’, somehow implying that the systems P and Q are behaviourally equivalent.

So the main issues become:

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

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

$$\forall\, x_N,\ \cdots\ x_1 \bullet r_P(x_N,\ p_N) = r_Q(x_N,\ q_N)$$

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

We should now check two things:

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

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

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

If the second check fails, then we proceed to the next stage of the iteration.

This completes our informal description of the method. The essence of it is that symbolic evolution of deterministic systems can be used as a basis of comparing the responses that would result after some number of inputs. If we can then show that the agreement of successive steps of each system’s evolution follows logically from the agreement of responses that have gone before then we argue that the systems are equivalent. Of course, we must also check that evolution from the initial states also establishes agreement for as many steps.

3.2 Psuedocode for the method

Based upon our informal presentation of the method above, we now present it again, but in a more formal way, making precise more of the details that were previously glossed over.

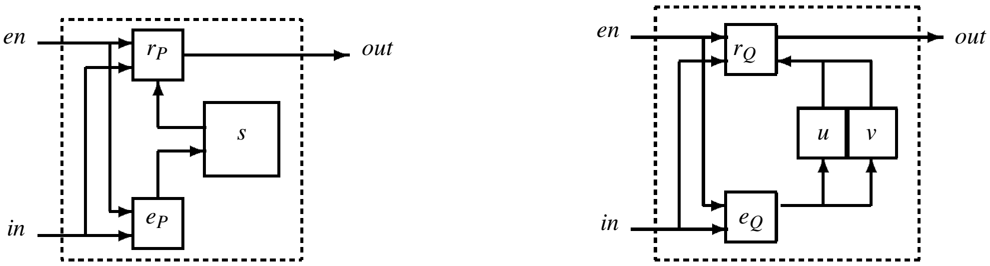
Taking P and Q to be fixed from now on, the following *response formula* $Rsp(v, \sigma_P, \sigma_Q)$ is the basic property of interest, and asserts that the responses of deterministic systems P and Q agree when stepping from a given pair of states, $\langle \sigma_P,\ \sigma_Q \rangle$, via input $v \in I$.

$$Rsp(v, \sigma_P, \sigma_Q)\ \underline{\triangle}\ \text{‘}r_P(v, \sigma_P) = r_Q(v, \sigma_Q)\text{’}$$

```
program StateEvolution(Γ,P,Q) =
    begin
       Γ0 := Γ;
       ⟨iP,rP,eP⟩ := P;
       ⟨iQ,rQ,eQ⟩ := Q;
       Input-Var-List := [ ];
       State-Term-P := σP;
       State-Term-Q := σQ;
       loop
           --- calculate new input variable
           Input-Var := New-Var();
           Var-List  := Input-Var :: Input-Var-List;

           --- calculate next state expressions
           State-Term-P := Next-State(Γ,eP,Input-Var,State-Term-P);
           State-Term-Q := Next-State(Γ,eQ,Input-Var,State-Term-Q);

           --- calculate response formulae
           Response-Form :=
                 Compute-Response(Input-Var,State-Term-P,State-Term-Q);
           Response-Form := Close-Vars(Var-List,Response-Form);
           Inst-Resp-Form := Instantiate(Response-Form,{σP ↦ iP,  σQ ↦ iQ});

           --- perform validity checks
           if Check-Valid(Γ0, Inst-Resp-Form) then
               if Check-Valid(Γ, Response-Form) then
                  return(true)
               else
                  Γ := {Response-Form} ∪ Γ
               end
           else
               abort(Inst-Resp-Form)
           end
       endloop
    end

fun Compute-Response(i,σP,σQ)
   --- computes the formula: Rsp(i,σP,σQ).

fun Check-Valid(Γ,P)
   --- in principal, checks for FO validity of Γ ⊨ P.

fun Next-State(Γ,e,x,s)
   --- computes the term 'e(x,s)' and simplifies it relative
   --- to appropriate definitions in Γ.

fun Close-Vars(vars,P)
   --- calculates the universal closure '∀ vars • P'

fun Instantiate(P,θ)
   --- calculates the instantiation of P by
   --- the substitution θ

fun New-Var()
   --- delivers a fresh variable
```

Figure 1: State Evolution method : Psuedocode

Using the above notation, the naive pseudocode for the method is presented in Figure 3.2 together with a brief statement concerning the auxiliary functions used.



(a) Latch P (b) Latch Q

Figure 2: Two Latches

3.3 Further notation

In the following sections we shall need some further notation. The formula $R_n(\sigma_P, \sigma_Q)$ is the basic property of interest, and says that the responses of deterministic systems P and Q agree (for all inputs) on every state at distance n from a given pair of states, $\langle \sigma_P,\ \sigma_Q \rangle$.
$R_n(\sigma_P, \sigma_Q) \quad \Leftrightarrow \quad \forall\, x_n\ \cdots\ \forall\, x_1 \bullet resp_n(P)(x_n,\ \cdots\ x_1, \sigma_P) = resp_n(Q)(x_n,\ \cdots\ x_1, \sigma_Q)$ We also need the following 'accumulating' version of predicate R:

$$AR_n(\sigma_P, \sigma_Q) \quad \Leftrightarrow \quad \bigwedge_{i=1}^{n} R_i(\sigma_P, \sigma_Q)$$

If valid, $AR_n(\sigma_P, \sigma_Q)$ says that $R_i(\sigma_P, \sigma_Q)$ is valid for all $1 \leq i \leq n$. In other words, the responses of systems P and Q agree for each step up to and including n, starting from states $\langle \sigma_P,\ \sigma_Q \rangle$.

As part of a more rigorous analysis, we demonstrate the soundness of the inference rule that underlies this method in Section 6. Issues concerning termination of this method will also be discussed in that section.

4 Short Example

As an example of state bisimulation and the state evolution method, two versions of an integer latch will be compared, as shown in Figure 2. Each latch has a data input *in*, a write-enable *en* and an output *output*. In both cases, when *en* is true then a new value supplied via *in* is stored in the latch; otherwise the previous value is retained. The difference between the two implementation lies in the method of storage, and the subsequent changes to the response and evolution functions: in latch P the integer is stored as a whole, but in latch Q the tens and units parts are stored separately and assembled on output. In the following system descriptions, a *single* input is used, taken over the input space $I = \mathbb{Z} \times \mathbb{B}$.

$$\begin{aligned}
P &= \langle i_P, r_P, e_P \rangle \\
i_P &= \langle 0 \rangle \\
r_P(\langle en, in \rangle, s) &= \textsf{if } en \textsf{ then } in \textsf{ else } s \\
e_P(\langle en, in \rangle, s) &= \textsf{if } en \textsf{ then } in \textsf{ else } s
\end{aligned}$$

$$\begin{aligned}
Q &= \langle i_Q, r_Q, e_Q \rangle \\
i_Q &= \langle 0, 0 \rangle \\
r_Q(\langle en, in \rangle, \langle u, v \rangle) &= \textsf{if } en \textsf{ then } in \textsf{ else } (10u + v) \\
e_Q(\langle en, in \rangle, \langle u, v \rangle) &= \textsf{if } en \textsf{ then } \langle in \textsf{ div } 10, in \textsf{ mod } 10 \rangle \\
&\qquad \textsf{else } \langle u, v \rangle
\end{aligned}$$

where i_P, i_P are the starting states for each system, r_P, r_Q are their response functions and e_P, e_Q are the evolution functions. The integer operations $\textsf{div}, \textsf{mod}$ have their usual meaning of integer division and modulus.

In order to show that P and Q are strongly bisimulation equivalent, we need to calculate the relations R_i, for $i = 1, 2, \cdots$, checking that each is valid for the initial state, stopping when we can show R_N is valid, under a set of assumptions $\Gamma_{N\text{-}1} = \Gamma_0, R_1, \cdots, R_{N\text{-}1}$. This procedure will generate a set of verification conditions at each stage, which are to be proved under Γ by a first order logic theorem prover. For this example, Γ_0 requires the addition of a theorem over the integers:

$$T1 = \forall\, x \in \mathbb{Z} \bullet (x = 10(x \text{ div } 10) + x \text{ mod } 10)$$
$$T1 \in \Gamma_0$$

where Γ_0 will contain other general assumptions to be used by the theorem prover. The first iteration is now performed, to calculate R_1, and the evolution expressions s_1, u_1, v_1:

$$\begin{aligned}
R_1(s_0, \langle u_0, v_0\rangle) &= \forall\, i_1 \in \mathbb{Z}, e_1 \in \mathbb{B} \bullet (\text{if } e_1 \text{ then } i_1 \text{ else } s_0 = \text{if } e_1 \text{ then } i_1 \text{ else } (10u_0 + v_0)) \\
s_1 &= e_P(\langle e_1, i_1\rangle, s_0) \\
&= \text{if } e_1 \text{ then } i_1 \text{ else } s_0 \\
\langle u_1, v_1\rangle &= e_Q(\langle e_1, i_1\rangle, \langle u_0, v_0\rangle) \\
&= \text{if } e_1 \text{ then } \langle i_1 \text{ div } 10, i_1 \text{ mod } 10\rangle \text{ else } \langle u_0, v_0\rangle
\end{aligned}$$

Clearly R_1 is valid when it is instantiated for the initial state (i.e., when $s_0 = u_0 = v_0 = 0$). However it is not valid for an arbitrary starting point $\langle s_0, \langle u_0, v_0\rangle\rangle$ in the product state space. We therefore need to continue the iteration, this time with set of assumptions Γ_1, which includes those obtained from the first iteration, i.e., R_1:

$$\Gamma_1 = \Gamma_0 \cup \{R_1\}$$
$$R_2(s_0, \langle u_0, v_0\rangle) = \forall\, i_2, i_1 \in \mathbb{Z}, e_2, e_1 \in \mathbb{B} \bullet \text{if } e_2 \text{ then } i_2 \text{ else } s_1 = \text{if } e_2 \text{ then } i_2 \text{ else } (10u_1 + v_1)$$

Again, R_2 is valid when instantiated with the initial state. We now need to see whether $R2$ is valid for the arbitrary starting point, assuming Γ_1:

$$\Gamma_0, R_1(s_0, \langle u_0, v_0\rangle) \models R_2(s_0, \langle u_0, v_0\rangle)$$

Expanding R_2 produces two verification conditions $VC1, VC2$:

$$VC1\text{: } \forall\, i_2, i_1 \in \mathbb{Z}, e_2, e_1 \in \mathbb{B} \bullet e_2 \;\Rightarrow\; i_2 = i_2$$
$$VC2\text{: } \forall\, i_2, i_1 \in \mathbb{Z}, e_2, e_1 \in \mathbb{B} \bullet \neg e_2 \;\Rightarrow\; (s_1 = 10u_1 + v_1)$$

$VC1$ is clearly valid; $VC2$ can be further split, by substituting for the state variables s_1, u_1, v_1, and then expanding the resulting expressions:

$$VC2a\text{: } \forall\, i_2, i_1 \in \mathbb{Z}, e_2, e_1 \in \mathbb{B} \bullet \neg e_2 \wedge \neg e_1 \;\Rightarrow\; s_0 = 10u_0 + v_0$$
$$VC2b\text{: } \forall\, i_2, i_1 \in \mathbb{Z}, e_2, e_1 \in \mathbb{B} \bullet \neg e_2 \wedge e_1 \;\Rightarrow\; i_1 = 10(i_1 \text{ div } 10) + i_1 \text{ mod } 10$$

Verification condition $VC2a$ is valid assuming R_1, whilst $VC2b$ is valid assuming $T1$. We have therefore shown that $P \sim Q$.

5 State Evolution rule

In this section, we shall abstract the logical effect of the operational method described above, by extracting a corresponding inference rule.

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

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

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

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

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

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

In the following section, we discuss the soundness of this rule and also the termination of the State Evolution method for finding N.

6 Soundness of State Evolution rule

Our basic strategy for demonstrating soundness of the above rule is to show that if the premisses of the rule hold, then this information can be used to construct a state bisimulation between the deterministic processes P and Q.

To begin, we need a few preparatory properties, concerning the interaction between the functions $step_n$ and the formulae R_n and AR_n.

- $R_{n+1}(\sigma_P, \sigma_Q) \ \Leftrightarrow\ \forall\, y \in I \bullet R_n(e_P(y, \sigma_P),\ e_Q(y, \sigma_Q))$
- $AR_{n+1}(\sigma_P, \sigma_Q) \ \Leftrightarrow\ R_1(\sigma_P, \sigma_Q) \ \wedge\ \forall\, y \in I \bullet AR_n(e_P(y, \sigma_P),\ e_Q(y, \sigma_Q))$

The above can be used to show some closure properties for the following family of state-space relations $C_n \subseteq (St_P \times St_Q)$, for $n \geq 1$:

$$C_n \quad = \quad \{\langle \sigma_P,\ \sigma_Q\rangle \in (St_P \times St_Q) \mid AR_n(\sigma_P, \sigma_Q)\}$$

Observe that the C_i relations are monotonically decreasing, $C_{i+1} \subseteq C_i$ for all $i \in \mathbb{N}$. Furthermore, we have that: $\forall\, y \in I,\ \langle \sigma_P,\ \sigma_Q\rangle \in C_{i+1} \bullet \langle e_P(y, \sigma_P),\ e_Q(y, \sigma_Q)\rangle \in C_i$.
It can be shown that:

$$\begin{array}{l} C_i \subseteq C_{i+1} \ \Leftrightarrow \\ \quad \forall\, y \in I,\ \langle \sigma_P,\ \sigma_Q\rangle \in C_i \bullet \langle e_P(y, \sigma_P),\ e_Q(y, \sigma_Q)\rangle \in C_i \end{array}$$

which relates closure under step with containment properties between them.

It is now relatively straightforward to obtain the desired theorem:

Theorem 6.1 : State Evolution as State Bisimulation
Suppose that we have established the following, for some fixed N:

(a) $\Gamma \vdash AR_N(i_P, i_Q)$

(b) $\Gamma \vdash \forall\, \sigma_P,\ \sigma_Q\ \bullet AR_N(\sigma_P, \sigma_Q)\ \Rightarrow\ R_{N+1}(\sigma_P, \sigma_Q)$

Then, we have that, with respect to Γ, C_N is a state bisimulation for P and Q.

Proof : (omitted in this deliverable)

As an immediate corollary we then get that:

Corollary 6.2 : Correctness of State Evolution method
If the premisses of the State Evolution rule can be established for some particular value, $N \geq 1$, then the systems P and Q are strongly bisimilar.

6.1 Termination of the method

As formulated above, there is no apriori reason why the State Evolution method should terminate at any stage N. In fact, it is easy to cast arbitrary deterministic Turing Machine descriptions as machines in normal form. In this case, behavioural equivalence of such machines corresponds to strong bisimilarity, thus showing that, in general, this method is incomplete and may not terminate for some machine descriptions P and Q. However, note that the risk of non-termination only arises when P and Q are actually *equivalent*. For when P and Q are inequivalent, there exists some input trace from the initial states which makes them respond differently. This trace is of some particular length, say r and so we have that $\not\models AR_r(i_P, i_Q)$. Accordingly, the method will abort at stage r, assuming decidability of the formulae $AR_i(i_P, i_Q)$.

Since the method involves the active use of theorem-proving techniques to judge validity, non-termination may arise for the 'trivial' reason that the formulae needing to be checked for validity may not be tractable or that they may be in some way undecidable. In this section we are more interested in non-trivial reasons for non-termination (i.e. reasons that do not concern the strength of theorem-proving techniques required to prove the premisses). Accordingly, within this section, we shall assume that it is possible to decide validity of the formulae involved (c.f completeness in the sense of Cook). Issues concerning practical implementation will be discussed further in later sections.

We shall now give a sufficient condition on the form of the machines involved for termination of this method. Let M be a deterministic machine in normal form (i.e. $M = \langle i, r, e \rangle \in DM(I, O, St)$):

We define a *state sphere* of radius n, $State_n(M, \sigma)$ as the set $\{step_n(M)(x_n,\ \cdots\ x_1, \sigma_P) \mid x_1,\ \cdots\ x_n \in I\}$. The *reachable* states of machine M is defined by:

$$Reachable(M) \triangleq \bigcup_{j=0}^{\infty} State_j(M, i)$$

A *span* of width n of machine M is the set of reachable states whose state spheres have saturated at radius n, i.e.

$$Span_n(M)\ \triangleq\ \{\sigma \in Reachable(M) \mid State_n(M, \sigma) = State_{n+1}(M, \sigma)\} \qquad .$$

The *diameter* of machine M is the minimum number of steps required to go from any (reachable) state to any other state that is reachable from it (i.e. it is a minmax property). We take this to be:

$$diam(M) = \begin{cases} \min reach(M) & \text{if } reach(M) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}$$
(where $reach(M) = \{n \in Nat \mid Span_n(M) = Reachable(M)\}$)

Finally, we say that a machine M is of *bounded diameter* whenever $diam(M)$ is finite.

We now explicitly introduce the *product* of deterministic machines as follows. Given machines $P \in DM(I, O_P, St_P)$ and $Q \in DM(I, O_Q, St_Q)$ where:

$$\begin{array}{lcl} P & = & \langle i_P, r_P, e_P \rangle \\ Q & = & \langle i_Q, r_Q, e_Q \rangle \end{array}$$

the *product machine* is denoted by: $(P \times Q) \in DM(I, (O_P \times O_Q), (St_P \times St_Q))$ and is defined by: $(P \times Q) \mathrel{\underline{\triangle}} \langle i_{PQ}, r_{PQ}, e_{PQ} \rangle$ and where $i_{PQ} = \langle i_P,\ i_Q \rangle$, $r_{PQ}(x, \langle \sigma_P,\ \sigma_Q \rangle) = \langle r_P(x, \sigma_P),\ r_Q(x, \sigma_Q) \rangle$, and $e_{PQ}(x, \langle \sigma_P,\ \sigma_Q \rangle) = \langle e_P(x, \sigma_P),\ e_Q(x, \sigma_Q) \rangle$.

We then have the following lemma:

Lemma 6.3 : Bounded diameter and step evolution
If machines P and Q are such that

- the product machine $(P \times Q)$ is of bounded diameter.
- $Reachable(P \times Q) = (St_P \times St_Q)$

then the State Evolution method terminates.

Proof : (omitted in this deliverable)

Unfortunately, there exist machines P and Q of bounded diameter whose product machine is $P \times Q$ is not of bounded diameter (Another exercise for the reader!). However, there are subclasses of bounded diameter machines which are closed under formation of product machines:

- finite state systems
- bounded history systems

A machine is said to be *finite state* when its state space is finite; a machine P has the *bounded history* property whenever:

$$\begin{array}{l} \exists\, N > 0 \bullet \forall\, \sigma_1, \sigma_2 \in Reachable(St_P) \bullet \\ \quad \forall\, x_N \ \cdots\ x_1 \bullet resp_N(P)(x_N,\ \cdots\ x_1,\ \sigma_1) = resp_N(P)(x_N,\ \cdots\ x_1,\ \sigma_2) \end{array}$$

In other words, the response of the system is *independent* of the state it was in after N input steps have been made. This typically happens when all data in the system is of a transitory nature.

7 Summary

We have introduced a state bisimulation approach for determining whether state-machine-like representations of two deterministic, synchronous hardware systems are behaviourally equivalent or not, in the sense of Milner's strong bisimulation. The approach can operate on bounded diameter systems which may have unbounded state-space, due to perhaps infinite data-sets. A state evolution method determines whether a state bisimulation holds. This effectively generates a set of verification conditions, which can be presented to standard first order logic theorem provers. The method offers the potential of a high degree of automation.

The state bisimulation approach will be extended to cover non-deterministic systems in future work.

8 References

[BCL91] J.R. Burch, E.M. Clarke, and D.E. Long. Symbolic model checking with partitioned transition relations. In *VLSI91*, Edinburgh, 1991.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, 1990.

[BGMW93] H. Barringer, G.D. Gough, B.Q. Monahan, and A. Williams. A Process Algebraic Semantics for Core ELLA. Technical report, FORMAL VERIFICATION SUPPORT FOR ELLA, IED 4/1/1357, University of Manchester, May 1993.

[BGMW94] H. Barringer, G.D. Gough, B.Q. Monahan, and A. Williams. A Process Algebra Foundation for Reasoning about Core ELLA. Technical report, Department of Computer Science, University of Manchester, January 1994.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.

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

[Lin93] H. Lin. A Verification Tool for Value-Passing Processes. Technical Report 8/93, University of Sussex, April 1993.

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

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