Computer Science
University of Manchester

The State Evolution Method for Verifying Hardware Systems

Howard Barringer, Graham Gough,

Brian Monahan, Alan Williams

Technical Report UMCS-95-7-1

FORMAL VERIFICATION SUPPORT FOR ELLA

IED 4/1/1357

The State Evolution Method for Verifying Hardware Systems*

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

July 1995

Recent technical reports issued by the Department of Computer Science, Manchester University, are available by anonymous ftp from ftp.cs.man.ac.uk in the directory pub/TR. The files are stored as PostScript, in compressed form, with the report number as filename. They can also be obtained on WWW via URL http://www.cs.man.ac.uk/csonly/cstechrep/index.html. Alternatively, all reports are available by post from The Computer Library, Department of Computer Science, The University, Oxford Road, Manchester M13 9PL, UK.

†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.

A poster presentation of this paper will be given at CHARME'95, Frankfurt, Germany, October 1995.

Abstract

We present a novel *state evolution* method for establishing standard (strong) bisimulation, which gives a tractable verification approach for *deterministic machines*, possibly with infinite state-spaces, and operates at an abstract level. The problem of establishing equivalence is reduced to one of proving the validity of a set of simpler (first-order) logical verification conditions, generated from the state evolution expressions. The approach maintains a high degree of automation, a feature of state-based methods, whilst offering the potential of containing the usual growth in complexity of verification, one advantage of using theorem-proving techniques.

Keywords: symbolic verification, automatic hardware verification, theorem-proving, hardware design aids.

1 Introduction

One approach commonly used for establishing the behavioural equivalence of hardware systems uses state-space exploration to establish a *bisimulation* relation between the systems, modelled as *labelled transition systems*[1]. It has the distinct advantage of being automatic, and can produce counter-example information when a verification fails, although this is usually in low level terms of traces leading to divergence. However, it suffers from the problem of combinatorial explosion, even though its domain of applicability can be significantly extended with the introduction of special encodings, such as binary decision diagrams (BDDs).

A second method represents system behaviour using logical expressions, and then establishes equivalence by employing theorem-proving techniques. It operates abstractly at a high level, thus not necessarily suffering from combinatorial explosion. Further, when two systems are shown to be different, it may be possible to give structured debugging feedback information at the level of the initial system representation. Although the approach can be automated to varying degrees, it currently requires significant user input in order to be able to establish design equivalence.

We introduce a novel verification approach which effectively merges the above techniques. We take Park and Milner's standard (strong) bisimulation equivalence[1] as our basic notion of equivalence between labelled transition systems. We present the transition systems abstractly as *deterministic machines* so that each system is expressed as a pair of (response and evolution) functions, together with an initial state. The bisimulation[1] between the systems can then be established semi-automatically through a *state evolution* rule, proceeding in a truly symbolic manner, at the level at which the designs are expressed.

The state evolution rule defines a pair of logical expressions whose validity establishes the strong bisimulation between the systems. However, because of the structure of the response and evolution functions in the deterministic machines, each logical expression can be further analysed, and decomposed into a set of smaller (first-order) *verification conditions* (*VCs*); the conjunction of these is equivalent to the original expression. Each VC has a regular form, which significantly eases the task of establishing its validity, using a standard theorem-prover/proof assistant — currently we use the PVS proof system[2], which includes decision procedures for equalities and linear arithmetic reasoning, with the potential for a high degree of automation. The problem of checking for bisimilarity has therefore been broken down, via the state evolution rule, into one of validating sets of smaller, regularly structured VCs. In fact, each VC

[1] In fact we define a *state bisimulation* relation between the *state spaces* of the two machines. This is provably equivalent to a strong bisimulation between the *derived* transition systems for the machines.

can be generated and analysed separately — if any is found to be invalid, then the two systems are immediately determined to be different, without any further work being necessary. The user can therefore monitor and control the progress of the state evolution. It is also possible for the user to apply specific design knowledge within the verification process, thus possibly reducing the size of the verification task — expressions can be abstracted and their behaviour represented by axioms introduced into the VCs.

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 possibility 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 state-space encodings, without first applying transformations to the systems, such as data-set reduction[3] — this essentially comes for free with our method. Viewed as a state-based approach, the method establishes bisimulation in a truly symbolic way, thus avoiding many aspects of combinatorial explosion. The logical expressions for the state evolution rule effectively characterise the bisimulation relation between the two systems. Viewed as a theorem-proving approach, the method offers automatic generation of a set of regularly structured VCs from the original design descriptions, thus decomposing the verification task. The VCs are simple enough to be amenable to automatic validation.

Note that we have restricted the method described here to apply to deterministic systems, in order to simplify the presentation. Of course, this means that (strong) bisimulation coincides with trace equivalence between two systems. However, the method can be extended to the non-deterministic case, although there will be an associated increase in complexity.

1.1 Context and Related Work

Our work on foundations and verification support has formed part of a collaborative project, involving Manchester, DRA (Malvern), Harlequin Ltd. (Cambridge), and GEC Plessey Semiconductors (Swindon), and sponsored by the DTI and SERC. In this we have developed a hardware design and verification environment to support the commercial hardware description language (HDL) ELLA[4]. A formal process algebra semantics has been developed for ELLA, and a compiler based on this translates ELLA expressions into process terms representing their behaviour. The process terms can in turn be translated into deterministic machine representations.

The aim of the project is to introduce verification support into industrial design methods for commercially sized systems, by providing semi-automatic tools which can be operated by the hardware designer, and do not require extensive formal methods expertise. A second requirement is that the verification tool can provide structured debugging information at the ELLA level, when designs are found to be different. The methods described in this paper fulfil many of these requirements. An overview of work from the project appears in [5][2].

Background to the work presented here is found in Milner's work on CCS and bisimilarity[1]. More recently, Hennessy[6] has developed techniques for modelling infinite data-spaces and constructing their bisimulations; the VPAM system[7] is somewhat similar to our approach of proving conditions which are generated when constructing a state bisimulation. Other work related to these include CIRCAL[8].

There has been extensive research into state-based verification methods such as model-checking[9, 10]. Although these have been traditionally restricted by the problem of 'state-explosion', the capabilities of these techniques have been enhanced by the introduction of compact state-space encodings, namely

[2] The ELLA Project World-Wide Web page is at URL:
`http://www.cs.man.ac.uk/fmethods/projects/ella-project.html`

BDDs — see [11, 12, 13] for key expositions. Commercial verification systems are now emerging, based on such techniques; for example the VFORMAL system (COMPASS) checks equivalence for synchronous systems described using VHDL. However, the state-spaces of the systems under analysis must be identical here.

Our work is clearly related to that of theorem-proving, sharing many of the benefits of this approach, although our method has additional advantages in terms of potential ease of (automated) proof. The HOL system[14] is an extensive example of theorem-proving tools development, which can be applied to design verification. The work also includes a system for *semantically embedding* HDLs, including ELLA, into HOL[15]. The related MEPHISTO tool[16] is embedded within HOL, and has been incorporated into a hardware design framework, offering partially automated verification via the development of proof tactics suitable for hardware systems. At Edinburgh, operational semantics are provided for simplified versions of ELLA, and embedded into the LAMBDA system[17], which supports the semi-automated synthesis of circuits from logical constraints. The PVS proof system utilised here, can be employed to model systems directly, and then to support the required proof task[18]. Currently these tools based on theorem-proving require formal methods expertise in order to operate them.

In [19], a symbolic simulation technique for high-level processor verification is described, which performs symbolic reachability analysis on the product state machine of the two designs. Currently, however, this can lead to false negatives, because it relies on syntactic identity to determine equality between the expressions involved — this problem does not arise with our state evolution method.

1.2 Structure of Paper

This paper has the following structure: in section 2 we define a state bisimulation for deterministic, synchronous systems, and relate this to strong bisimilarity. An integer latch example illustrates the approach, and is introduced in section 2.4. In section 3 the *state evolution* rule is presented. The method for establishing the rule using verification condition generation is described in section 4, and its implementation is outlined in section 4.2. Section 4.3 shows the verification of the latch example.

2 Deterministic Machines and Labelled Transition Systems

We describe a *state bisimulation* relation for *deterministic machines* P and Q, which is equivalent to strong bisimilarity between processes in the *derived labelled transition systems* T_P and T_Q for the two machines. An accompanying *state evolution* rule defines two logical expressions whose validity establishes state bisimilarity. We use a simple example of integer latch designs as illustration. Note that even in this example, the bisimulation relation is infinite, and so could not be established directly using standard state exploration techniques, even with special encodings of the transition relations.

2.1 Deterministic Machines

We consider a system described as a *deterministic machine* D in the following way: let I, O be sets for input and output values, and let St_D be the state-space for D. Then

$D = \langle i_D, r_D, e_D \rangle$

where

i_D	$\in St_D$	— `initial state`
r_D	$: (I \times St_D) \rightarrow O$	— `response function`
e_D	$: (I \times St_D) \rightarrow St_D$	— `evolution function`

2.2 State Bisimulation

In order to establish the equivalence between two deterministic machines P and Q, we introduce the following *state bisimulation* relation on the state spaces of P and Q:

Definition 2.1 : State Bisimulation

SB is a state bisimulation for deterministic machines P, Q when:

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

Two systems P and Q are said to be *state bisimilar* — denoted $P \sim_{SB} Q$ — whenever there exists a state bisimulation SB for P and Q.

2.3 Derived Labelled Transition Systems

State bisimulation can be related to the standard (strong) bisimulation between processes. First we define the *input/output* labelled transition system $T_D = \langle Prc_D, Act_D, Trans_D \rangle$ corresponding to a deterministic machine D. Here Prc_D is a set of process objects, $Act_D \subseteq (I \times O)$ is the set of input/output actions, and $Trans_D \subseteq (Prc_D \times Act_D \times Prc_D)$ is the transition relation for the system, which describes how a process $p \in Prc_D$ evolves into another process $q \in Prc_D$ by making an input/output transition (x/y) for some $x \in I,\ y \in O$:

$$
\begin{aligned}
Prc_D = \quad & \{pD(\sigma_D) \mid \sigma_D \in St_D\} \\
Trans_D = \quad & \{p \xrightarrow{x/y} q \mid \sigma_D \in St_D, x \in I, \\
& \qquad\qquad p = pD(\sigma_D), q = pD(e_D(x, \sigma_D)), \\
& \qquad\qquad y = r_D(x, \sigma_D) \\
& \}
\end{aligned}
$$

where pD is a suitably injective mapping from the state space of D into process objects[3]. Note that T_D will be *input deterministic*, so that if two transitions in $Trans_D$ have the same starting process and the

[3] For example, if the set of process objects is chosen to coincide with the state-space, i.e. $Prc_D = St_D$, then the mapping pD from states to processes may be the identity — however, this particular correspondence between states and processes is not a necessary condition.

same input value on their labels, then they will have the same destination process, and have the same output value on their labels.

Definition 2.2 : (Strong) Bisimilarity

We take equivalence on processes to be the *strong bisimilarity* of Park and Milner[1]. We fix the transition system T and let $S \subseteq Prc^2$: The relation S is a (strong) bisimulation if and only if:

- S is symmetric.
- $\langle P, Q \rangle \in S \ \Rightarrow$
  If $P \xrightarrow{\alpha} P'$ then there is some Q, such that $Q \xrightarrow{\alpha} Q'$ and $\langle P', Q' \rangle \in S$

Two processes P, Q are said to be *bisimilar* — denoted $P \sim Q$ — whenever there exists a strong bisimulation S such that $\langle P, Q \rangle \in S$.

The required association between the state bisimulation relation on deterministic machines P and Q, and the (strong) bisimulation for the corresponding labelled transition systems T_P and T_Q is expressed in the following way:

Theorem 2.3 : Correctness of State Bisimulation

Let $P,\ Q$ be deterministic machines:

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

where pP, pQ are suitable injective mappings from the state spaces of P, Q, and $pP(i_P)$ and $pQ(i_Q)$ are the processes derived from the initial states of the two machines P and Q. This says that there is a state bisimulation between machines P and Q, if and only if there is a strong bisimulation between the processes derived from their initial states, with respect to the derived transition systems T_P and T_Q.

The proof is straightforward.

2.4 Example: Integer Latch

As a simple example, consider the following two implementations of an enabled integer latch. The basic design P has the standard structure. It takes a data input $in \in \mathbb{Z}$ and an enable control $en \in control$, and outputs the last stored value m. When $en = old$, then the latch retains the old value, and when $en = new$, the current input value in is stored.

A second design Q is intended to have the same behaviour as P. Here, the latch store consists of two registers, and before being stored, an input value is first split into its *low* and *high* components using the arithmetic operators div and mod, and an offset constant $A \in \mathbb{Z}$. On output the original value is retrieved in the following way:

$$output = low + A * high$$

Deterministic Machines The deterministic machines for the two designs are as follows:

$$control = \{new, old\}$$
$$I = \mathbb{Z} \times control, \quad O = \mathbb{Z}, \quad St_P = \mathbb{Z}, \quad St_Q = \mathbb{Z} \times \mathbb{Z}$$

$$\begin{array}{rcl} r_P(\langle in, en\rangle, m) & \triangleq & m \\ e_P(\langle in, en\rangle, m) & \triangleq & \text{if } en = new \text{ then } in \text{ else } m \\ r_Q(\langle in, en\rangle, \langle low, high\rangle) & \triangleq & low + A * high \\ e_Q(\langle in, en\rangle, \langle low, high\rangle) & \triangleq & \text{if } en = new \text{ then } (in \bmod A, in \operatorname{div} A) \text{ else } (low, high) \end{array}$$

for some constant $A \in \mathbb{Z}$, with the initial states given by $i_P = 0, i_Q = \langle 0, 0\rangle$.

Derived Labelled Transition System The derived labelled transition system for latch design P is given by:

$$T_P = \langle Prc_P, IOAct_P, Trans_P\rangle$$

where

$$\begin{array}{rl} Prc_P = & \{pP(m) \mid m \in St_P\} \\ Act_P = & (\mathbb{Z} \times control) \times \mathbb{Z} \\ Trans_P = & \{pP(m) \xrightarrow{in, en/m} pP(n) \mid m, in, n \in \mathbb{Z}, en \in control, \\ & \qquad\qquad n = (\text{ if } en = new \text{ then } in \text{ else } m) \\ & \} \end{array}$$

A transition system for design Q can be derived in a similar way.

State Bisimulation We claim that the following relation SB is a state bisimulation between P and Q, and that the two designs are therefore equivalent, in the sense that the derived labelled transition systems T_P and T_Q are strongly bisimilar:

$$\begin{array}{rl} SB & \subseteq St_P \times St_Q \\ & \subseteq \mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z}) \\ & = \{\langle x, (v, w)\rangle \mid x \in St_P, \langle v, w\rangle \in St_Q \bullet (x = v + A * w)\} \end{array}$$

The equivalence of P and Q is established in §4.3 using the methods presented below.

3 State Evolution

The problem of establishing the equivalence of two systems P and Q has now been defined as one of establishing a state bisimulation between the state spaces of P and Q. This can be performed in a truly symbolic manner by using the *state evolution* rule. This generates a set of verification conditions from the response and evolution functions for P and Q whose validity ensures that the systems are (state) bisimilar. The idea is to take an *arbitrary* point $\langle \sigma_P,\ \sigma_Q \rangle$, in the product state space $(St_P \times St_Q)$. As we know nothing about this point to start with, we can simply name it using appropriate symbolic variables.

We then proceed to symbolically evolve both of the systems, checking that for all inputs, the responses produced by each system are in agreement at each stage. Now, either:

1. we show that, for some input, the responses disagree at some stage, meaning that P and Q are different, *or*

2. we demonstrate that the symbolic simulation is 'sufficiently complete', implying that the systems P and Q are behaviourally equivalent.

So the main issues become:

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

Let us imagine that we are at stage N in applying the method, and so we consider the comparison of responses for the current step, based upon previous steps:

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

where sp_N and sq_N are expressions, dependent upon inputs $x_{N\text{-}1},\ \cdots\ x_1$ and initial product state $\langle \sigma_P,\ \sigma_Q \rangle$, describing symbolically the state that is reached after N-1 input steps. Therefore, R_N will hold if the responses are equal at step N in the symbolic simulation. We also require the 'accumulating' version of R:

$$AR_N(\sigma_P, \sigma_Q) \ \Leftrightarrow\ \bigwedge_{i=1}^{N} R_i(\sigma_P, \sigma_Q)$$

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 $1 \ldots N$, agreement of response is actually *forced* for the next step $N+1$.

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

If both the first and second check are positive, then we stop with success since, intuitively, we have shown that the next step is implied by the immediate past history. Hence, the systems will continue to behave identically, if a starting product state can be found where they will agree for N steps — of course the success of the first check shows that this has already been achieved from the initial state.

If the *second* check fails, then we proceed to the next stage of the iteration, since we have not been able to show one way or another whether the two systems are equivalent or not.

Definition 3.1 : State Evolution rule

We now give the State Evolution inference rule, for deterministic systems $P = \langle i_P, r_P, e_P\rangle$ and $Q = \langle i_Q, r_Q, e_Q\rangle$:

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

The theory Γ, apart from containing process definitions, may also contain design assumptions in the form of axioms.

This 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 step $N+1$ from states $\langle \sigma_P,\ \sigma_Q\rangle$ follows from equivalent responses for all preceding steps from the states $\langle \sigma_P,\ \sigma_Q\rangle$

then we can conclude that the two machines P and Q are state bisimilar. This also means, by Theorem 2.3, that the two processes $pP(i_P)$ and $pQ(i_Q)$ are strongly bisimilar as required, with respect to the labelled transition systems T_P and T_Q derived from the deterministic machines P and Q.

3.1 Soundness of State Evolution Rule

We outline the strategy for demonstrating soundness, which is to show that if the premises of the rule hold, then this information can be used to construct a state bisimulation between the deterministic machines P and Q.

We first give a useful property of the step predicate AR:

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

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

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

Observe from the definition that the C_i relations are monotonically decreasing, i.e. $C_{i+1} \subseteq C_i$ for all $i \in \mathbb{N}$. Furthermore, we can show 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:

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

using the step predicate for AR which relates closure under step with containment properties between them.

It is now relatively straightforward to obtain the desired theorem:

Theorem 3.2 : 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, with respect to Γ, C_N is a state bisimulation for P and Q. This follows because clause (a) says that the initial states are in C_N. and clause (b) says that C_N is closed under step.

3.2 Termination

The state evolution method described above will terminate for any two deterministic systems P and Q when they are *not* equivalent. The method will also *always* terminate when P and Q are finite. However, clearly termination is not always guaranteed for arbitrary equivalent systems, since the problem is undecidable. A sufficient condition for termination involves considering the form of the *product machine* $(P \times Q)$: a deterministic machine has *bounded diameter* whenever it requires a bounded number of steps to proceed from any (reachable) state to any other state that is reachable from it. The state evolution method will then terminate when the product machine has a bounded diameter and the set of its reachable states covers the product state-space. For example, for machines which are finite state, or have a *bounded history*, then the product machines will have bounded diameter, and the state evolution rule will therefore terminate. A machine P has a bounded history if its starts in a state σ_P, and when it has received a finite number of inputs, then the response of the system is *independent* of σ_P.

3.3 Pseudo-code for the method

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 $v \in I$.

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

Using the above notation, the pseudo-code for the method is presented in Figure 1. Note that `StateEvolution` simultaneously checks the validity of the two premises (a) and (b) in the state evolution rule. The logical expressions `Inst-Resp-Form` and `Response-Form` represent, respectively, the response formulae for premises (a) and (b) at step $i \in \{1, \ldots, N\}$ — the former starts from the initial state pair $\langle i_P,\ i_Q \rangle$ of machines P and Q, whereas the latter starts from an *arbitrary* state pair $\langle \sigma_P,\ \sigma_Q \rangle$, i.e. variables universally quantified over the two state-spaces.

The auxiliary functions used have the following behaviour:

```
program StateEvolution(Γ,P,Q, N) =
    begin
       Γ0 := Γ;
       ⟨iP,rP,eP⟩ := P;
       ⟨iQ,rQ,eQ⟩ := Q;
       Input-Var-List := [ ];
       State-Term-P := σP;
       State-Term-Q := σQ;
       loop for i in 1..N do
           --- calculate new input variable
           Input-Var := New-Var();
           Var-List  := Input-Var :: Input-Var-List;

           --- 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 simplification/validity checks
           if (Check-Valid(Γ0, Inst-Resp-Form) and
                  Check-Valid(Γ, Response-Form))
           then  Γ := {Response-Form} ∪ Γ
           else return(false)
           end

           --- 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);

       endloop

       --- calculate new input variable for N+1
       Input-Var := New-Var();
       Var-List  := Input-Var :: Input-Var-List;

       --- calculate response formula
       Response-Form :=
                Compute-Response(Input-Var,State-Term-P,State-Term-Q);
       Response-Form := Close-Vars(Var-List,Response-Form);

       --- perform final validity check
       Check-Valid(Γ, Response-Form)
    end
```

Figure 1: State Evolution method : Pseudocode

`Compute-Response(`i,σ_P,σ_Q`)`	— *computes the formula:* $Rsp(i,\sigma_P,\sigma_Q)$.
`Check-Valid(`Γ,P`)`	— *in principal, checks for first-order validity of* $\Gamma \models P$.
`Next-State(`Γ,e,x,s`)`	— *computes the term '*$e(x,s)$*' and simplifies it relative*
	— *to appropriate definitions in* Γ.
`Close-Vars(`$vars,P$`)`	— *calculates the universal closure '*$\forall\ vars \bullet P$*'*
`Instantiate(`P,θ`)`	— *calculates the instantiation of P by the substitution* θ
`New-Var()`	— *delivers a fresh variable*

4 Verification Condition Generation

We consider deterministic systems where the task of checking the agreement of the response forms at each stage performed by `Check-Valid(`Γ,P`)` is greatly simplified due to their regular nature. Function expressions for the response and evolution functions use a case statement as their control structure, where the chooser sets or guards on the case arms are disjoint and covering. Functions F and G, which depend on an input x, can be written as follows:

$$F(x) \triangleq \bigwedge_{i=0}^{k}(A_i[x] \;\Rightarrow\; t_i), \qquad G(x) \triangleq \bigwedge_{j=0}^{m}(B_j[x] \;\Rightarrow\; u_j)$$

where $A_i[x], B_j[x]$ are the control guards possibly dependent on input x, and t_i, u_j are the (case-less) expressions to be evaluated for each guard. This means that for any particular selection value, precisely one branch will be chosen. The function equalities from the state evolution rule can then be converted into a *functional normal form* in the following way:

$$\boxed{\text{FNF}}\;\frac{F(x) = G(x)}{\bigwedge_{i,j}(A_i[x] \wedge B_j[x] \;\Rightarrow\; t_i = u_j)}$$

where the terms t_i and u_j are case-less. Each equality is translated into a conjunction of *verification conditions* (*VCs*). A VC is regularly structured as an implication, where the conclusion is an equality, and the premise is a conjunction of constraints and equalities, as follows:

$$((constr_1 \wedge \cdots \wedge constr_k) \wedge (equality_1 \wedge \cdots \wedge equality_m) \;\Rightarrow\; concl)$$

where

$$\forall\ i \in \{1,\ldots,k\} \bullet constr_i = (var_i \in set_i)$$
$$\forall\ i \in \{1,\ldots,m\} \bullet equality_i = (lhs\text{-}expr_i = rhs\text{-}expr_i)$$
$$concl = (concl\text{-}expr_1 = concl\text{-}expr_2)$$

All variables are universally quantified over appropriate data sets. This simple structure enables the validity checking for each VC to be highly automated using a standard algebraic-solving tool, without the need to resort to the more general capabilities of a first-order theorem-prover.

4.1 Complexity

The number of VCs which are generated for a particular pair of designs is naturally highly dependent on the exact structure of the designs involved. Let design P have a total control branching of m and Q have a branching of n. Let N be the (fixed) number of steps required by the state evolution rule — this is usually related to the latency within the designs, and to their testability. The worst-case number of VCs will then be $O((mn)^N)$. This assumes that the control structures of the two designs are effectively independent, so that every control branch in one design must be checked against every branch in the other design.

However, in practice, if the systems have been designed with the intention of their being equivalent, then it is likely that the control structures will intersect — certain VCs will be trivially true and can be discarded, because the constraints on control variables are contradictory. So consider the case when the control structures coincide completely, then the number of VCs generated will be $O(m^N)$. This can be observed with the latch example, in §4.3.

4.2 Implementation

A verification condition generation (VCG) tool has been implemented in Common Lisp, and installed into a design and verification environment[20] developed[4] for the commercial hardware description language ELLA. To check the equivalence of ELLA designs, they are first compiled into deterministic machines, according to the formal semantics of ELLA[5]. The VCG then constructs the logical expressions for the premises of the state evolution rule — the user supplies the number of steps required It then converts the function expressions representing the machines into functional normal form and merges the resulting elements for each control branch to form a set of VCs. The VCG also performs extensive simplification, discarding those VCs which is can determine to be true.

The remaining set of VCs is then converted into a format suitable for the PVS theorem prover[2], which can be used to establish the validity of the VCs. If the VCs are satisfiable, but not valid, then the VCG may be re-run for a larger number of state evolution steps.

For further details of the implementation of the VCG, together with several examples of design verification, see [22].

4.3 Latch Example: Verification

For the latch example, described in §2.4, the state evolution rule (Definition 3.1) needs to be applied for a single step, i.e. $N = 1$. The step premise (b) from the state evolution is given by:

$$\begin{array}{ll|ll}
 & r_P(\langle in_1, en_1\rangle, m_0) = r_Q(\langle in_1, en_1\rangle, \langle low_0, high_0\rangle) & & R_1(m_0, \langle low_0, high_0\rangle) \\
\wedge & m_1 = e_P(\langle in_1, en_1\rangle, m_0) & & \\
\wedge & \langle low_1, high_1\rangle = e_Q(\langle in_1, en_1\rangle, \langle low_0, high_0\rangle) & & \\
\Rightarrow & r_P(\langle in_2, en_2\rangle, m_1) = r_Q(\langle in_2, en_2\rangle, \langle low_1, high_1\rangle) & \Rightarrow & R_2(m_0, \langle low_0, high_0\rangle)
\end{array}$$

The left-hand side gives the step premise in terms of the response and evolution functions of P and Q, and the right-hand side simply restates the logical expression for the premise. Note also that the initial premise (a) from the state evolution rule contains the antecedents from above, with $m_0 = low_0 = high_0 = 0$, i.e. it is given by $R_1(0, \langle 0, 0\rangle)$,

[4]The design and verification environment is built on top of the LispWorks programming environment[21], which is a trademark of, and produced by, Harlequin Ltd. (Cambridge, UK).

Functional Normal Form Each equality from above can be translated into functional normal form. For example the FNF for the expression:

$$\langle low_1, high_1 \rangle = e_Q(\langle in_1, en_1 \rangle, \langle low_0, high_0 \rangle)$$

is given by

$$\forall\, in_1 \in \mathbb{Z}, en_1 \in control \bullet \left(\begin{array}{ll} & en_1 \in \{new\} \\ \Rightarrow & \langle low_1, high_1 \rangle = \langle in_1 \;\mathsf{mod}\; A, in_1 \;\mathsf{div}\; A \rangle \end{array} \right) \wedge \left(\begin{array}{ll} & en_1 \in control - \{new\} \\ \Rightarrow & \langle low_1, high_1 \rangle = \langle low_0, high_0 \rangle \end{array} \right)$$

Verification Conditions The other expressions are expanded in a similar way, and the resulting FNFs are then merged and simplified. This results in the following two VCs for the step premise of the state evolution rule:

$$\begin{array}{ll|ll} & en_1 \in \{new\} & & en_1 \in control - \{new\} \\ \wedge & m_0 = low_0 + A * high_0 & \wedge & m_0 = low_0 + A * high_0 \\ \wedge & m_1 = in_1 & \wedge & m_1 = m_0 \\ \wedge & \langle low_1, high_1 \rangle = \langle in_1 \;\mathsf{mod}\; A, in_1 \;\mathsf{div}\; A \rangle & \wedge & \langle low_1, high_1 \rangle = \langle low_0, high_0 \rangle \\ \Rightarrow & m_1 = low_1 + A * high_1 & \Rightarrow & m_1 = low_1 + A * high_1 \end{array}$$

Note that merging of the FNFs initially results in four VCs — two each result from expanding e_P and e_Q, since the control variable en can make two choices. However two of the VCs contain contradictions in the constraints for en_1, when $(en_1 \in \{new\})$ from e_P and $(en_1 \in control - \{new\})$ from e_Q, and vice versa. The VCG can detect the contradictions in the set expressions and immediately discard the two VCs. Also note that the disjointness and covering of the control branches of the two evolution functions e_P, e_Q is exploited when merging the FNFs to generate the VCs.

Figure 2 shows the implemented VCG tool displaying the second VC from above, having performed further simplifications to remove the variable bindings. The VC has been converted into a format suitable for PVS.

Proof of VCs In order to prove the second VC, then the following axiom is required by PVS, which characterises exactly the property of `div` and `mod` used to develop latch Q:

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

Both VCs can now be proved in PVS in at most two user steps, once the expressions have been converted into sequent form and the universally quantified variables replaced by Skolem names. First, the axiom `div_ax` is installed as a rewrite rule, and then the automated PVS simplification routines and arithmetic decision procedures[2] are invoked to complete the proof, without further user intervention.

So the two latch designs are therefore equivalent, as claimed in §2.4. Here the VCG has already automatically discarded two VCs during merging, because the constraints were contradictory. The two remaining VCs generated have a regular structure, and designer knowledge is used during the proof, via the `div_ax` axiom. Notice also how the size of the input data set does not affect the proof in any way — the number of VCs generated is dependent on the size of the control branching, and the pure data components have been abstracted.

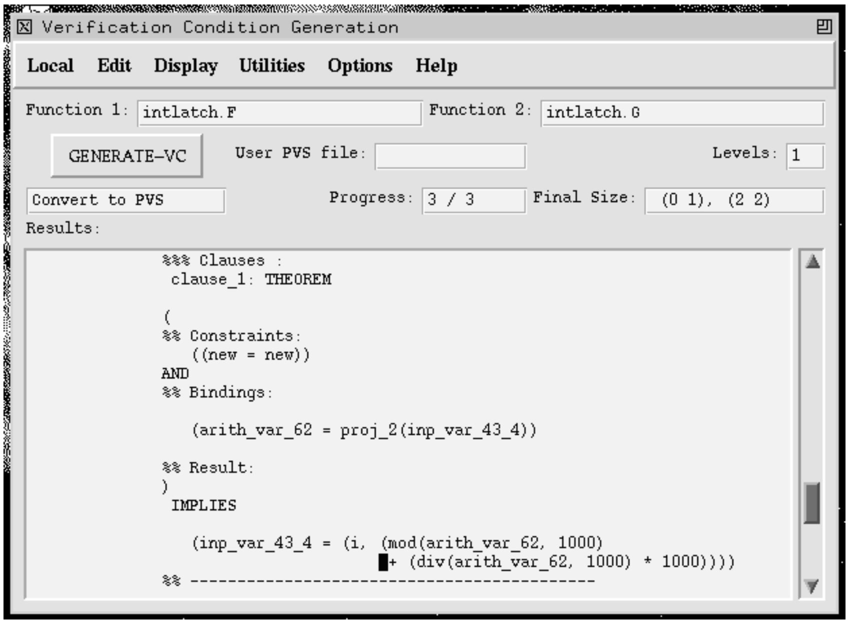

Figure 2: Verification Condition Generator

5 Summary

We have introduced a novel *state evolution* approach for determining the behavioural equivalence, in the sense of Park and Milner's strong bisimulation, of state-machine-like representations of two deterministic, synchronous hardware systems. The approach can operate on bounded diameter systems which may have unbounded state-space, due to perhaps infinite data-sets. The state evolution rule is used to determine whether a bisimulation holds. This effectively generates a set of regularly structured verification conditions, which can be presented to standard first order logic theorem proving tools. The method offers the potential of a high degree of automation, and also of containing the usual growth in complexity of verification. When systems are found to be different, then debugging feedback information can be presented at the level of the original design descriptions. A verification condition generation tool has been implemented and embedded into a design and verification environment for the commercial HDL ELLA, with encouraging results.

A description of an extension to the state evolution method, to non-deterministic systems, will be presented in a future paper.

6 References

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

[2] N. Shankar, S. Owre, and J. M. Rushby. The PVS Proof Checker: A Reference Manual (Beta Release). Technical report, Computer Science Laboratory, SRI International, March 1993.

[3] R. Kurshan. Reducibility in Analysis of Coordination. In *Lecture Notes in Control and Information Sciences*, volume 103, pages 19–39. Springer, 1987.

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

[5] H. Barringer, G. Gough, B. Monahan, and A. Williams. A Process Algebra Foundation for Reasoning about Core ELLA. Technical Report UMCS-94-12-1, University of Manchester, December 1994.

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

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

[8] G. A. McCaskill and G. Milne. Hardware Description and Verification Using the Circal-System. Technical Report HDV-24-92, Department of Computer Science, University of Strathclyde, June 1992.

[9] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of finite state concurrent systems using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2), 1986.

[10] H. Barringer, M. Fisher, and G.D. Gough. Fair SMG and Linear Time Model Checking. In Sifakis [23].

[11] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[12] J.R. Burch, E.M. Clarke, and D.E. Long. Representing Circuits More Efficiently in Symbolic Model Checking. In *DAC91*, 1991.

[13] O. Coudert, C. Berthet, and J-C. Madre. Verification of Synchronous Sequential Machines based on Symbolic Execution. In Sifakis [23].

[14] M. Gordon and A. Pitts. The HOL logic and system (Chapter 3). In J. Bowen, editor, *Towards Verified Systems*, pages 49–70. Elsevier, October 1994.

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

[16] K. Schneider, R. Kumar, and T. Kropf. Automating Most Parts of Hardware Proofs in HOL. In K. Larsen, editor, *Computer Aided Verification, (CAV '91), LNCS*, volume 575, Aarlborg, Denmark, July 1991. Springer-Verlag.

[17] K.G.W. Goossens. *Embedding Hardware Description Languages in Proof Systems*. PhD thesis, University of Edinburgh, 1992.

[18] N. Shankar. Verification of Real-Time Systems Using PVS. In C. Courcoubetis, editor, *Computer Aided Verification, (CAV '93), LNCS*, volume 697, Elounda, Greece, June 1993. Springer-Verlag.

[19] F. Corella. Automated High-Level Verification against Clocked Algorithmic Specifications. In D. Agnew, L. Claesen, and R. Camposano, editors, *CHDL '93*, Canada, 1993. North-Holland.

[20] H. Barringer, G. Gough, B. Monahan, and A. Williams. The ELLA Verification Environment: A Tutorial Introduction. Technical Report UMCS-94-12-2, University of Manchester, December 1994.

[21] Harlequin Ltd, Cambridge UK. Information about LispWorks is available on the World-Wide Web, URL: `http://www.harlequin.co.uk/full/products/sp/lispworks.html`.

[22] H. Barringer, G. Gough, B. Monahan, and A. Williams. Formal Verification Support for the ELLA Hardware Description Language: A Review. To appear in CHARME95, Frankfurt, Germany, October 1995.

[23] J. Sifakis, editor. *Automatic Verification Methods for Finite State Systems, LNCS*, volume 407, Grenoble, France, 1989. Springer-Verlag.