Theoretical Computer Science 231 (2000) 113–128

Theoretical Computer Science

www.elsevier.com/locate/tcs

Automata of asynchronous behaviors ☆

J.A. Brzozowski *, R. Negulescu [1]

Department of Computer Science, University of Waterloo, Waterloo, Ont., Canada N2L 3G1

Abstract

We survey three applications that use finite automata to specify behaviors of concurrent processes in general, and asynchronous circuits in particular. The applications are: verification of concurrent processes, liveness properties, and delay insensitivity of asynchronous networks. In all three cases, we start with a common model of a nondeterministic finite automaton, and then add certain application-specific features. Typically, the added features involve separating the alphabet or the state set of the automaton into several disjoint subsets. For each application we provide the motivation, describe the type of automaton used, define the most important operations, and state some of the key results. For process verification, we describe a BDD-based tool that implements the respective automata and operations. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Asynchronous; Automaton; Behavior; Circuit; Verification

1. Introduction

We begin by introducing some concepts common to several applications. The notation is loosely based on that of [6].

Definition 1. An *action automaton* is a quadruple $\mathscr{A} = \langle \Sigma, Q, I, E \rangle$, where
1. Σ is a finite set, called the *alphabet* of *actions*.
2. Q is a finite, nonempty set of *states*.
3. $I \subseteq Q$ is a nonempty set of *initial states*.
4. $E \subseteq Q \times \Sigma \times Q$ is a set of *edges*.

A *path* of $\mathscr{A}$ is a finite sequence $c = (q_0, \sigma_1, q_1) \ldots (q_{k-1}, \sigma_k, q_k)$ of consecutive edges; the length of this path is k. The element $s = \sigma_1 \ldots \sigma_k$ of Σ^* is the *word* of the path c. For each state $q \in Q$, there is a *trivial path* ε_q from q to q with word ε, the empty word. Note that trivial paths are not edges. A path with $q_0 \in I$ is called

☆ This research was supported by the Natural Sciences and Engineering Research Council of Canada under grant No. OGP0000871, and by an Ontario Graduate Scholarship.

* Corresponding author.

E-mail address: brzozo@maveric.uwaterloo.ca (J.A. Brzozowski)

[1] Presently with Department of Electrical and Computer Engineering, McGill University. http://www.macs.ece.mcgill.ca/~radu

initialized. The *language* of $\mathscr{A}$ is the set of words of the initialized paths. It follows that the language of an action automaton is *prefix-closed*, i.e., that every prefix of a word in the language also belongs to the language. Also, such a language is never empty, since it contains ε.

Definition 2. An automaton is *deterministic* if it has exactly one initial state, and for each $q \in Q$ and $\sigma \in \Sigma$ there is at most one edge of the form (q, σ, q').

Action automata will be used to represent processes. The product, defined below, of two action automata will then denote their joint behavior.

Definition 3. The *product* of two action automata $\mathscr{A}^1 = \langle \Sigma^1, Q^1, I^1, E^1 \rangle$ and $\mathscr{A}^2 = \langle \Sigma^2, Q^2, I^2, E^2 \rangle$ is an action automaton $\mathscr{A} = \mathscr{A}^1 \times \mathscr{A}^2 = \langle \Sigma, Q, I, E \rangle$ where
1. $\Sigma = \Sigma^1 \cup \Sigma^2$.
2. $Q = Q^1 \times Q^2$ (the Cartesian product of sets Q^1 and Q^2).
3. $I = I^1 \times I^2$ (the Cartesian product of sets I^1 and I^2).
4.
$$\begin{aligned} E = \big\{((q^1,q^2),\sigma,(q'^1,q'^2)) \mid \; & \\ & \sigma \in \Sigma^1 \backslash \Sigma^2 \wedge (q^1,\sigma,q'^1) \in E^1 \wedge q^2 = q'^2 \in Q^2 \\ & \bigvee \sigma \in \Sigma^2 \backslash \Sigma^1 \wedge (q^2,\sigma,q'^2) \in E^2 \wedge q^1 = q'^1 \in Q^1 \\ & \bigvee \sigma \in \Sigma^1 \cap \Sigma^2 \wedge (q^i,\sigma,q'^i) \in E^i, \text{ for } i = 1,2 \big\}. \end{aligned}$$

The rationale for this definition of product is that the edges of $\mathscr{A}$ should be ‘compatible’ with both $\mathscr{A}^1$ and $\mathscr{A}^2$. If we strip an edge of $\mathscr{A}$ of the state components q^2 and q'^2, we should obtain either an edge of $\mathscr{A}^1$ or a self-loop labeled with an action from outside Σ^1. Since going through a self-loop leaves the state unchanged, the action from outside Σ^1 is ‘ignored’ by $\mathscr{A}^1$. A symmetrical property holds for $\mathscr{A}^2$. Note that the language of $\mathscr{A}^1 \times \mathscr{A}^2$ is $\{w \in \Sigma^* \mid w \downarrow \Sigma^1 \in L^1 \wedge w \downarrow \Sigma^2 \in L^2\}$, where L^1 and L^2 are the languages of $\mathscr{A}^1$ and $\mathscr{A}^2$, respectively, and $\downarrow$ is the *projection* operation defined as follows. For word w and alphabet Γ, $w \downarrow \Gamma$ is the word obtained by deleting from w all actions that are not in Γ.

2. Process automata

In [10, 13], a simple theory of concurrent systems was introduced. This theory, called “process spaces”, unifies several types of systems and correctness criteria for such systems by abstracting the notion of “execution”. Executions can be finite or infinite sequences of events, functions of time, etc., but, in the general theory, executions are just elements of an arbitrary set $\mathscr{E}$.

Although executions have no structure in general process spaces, for a particular problem one needs to choose particular executions. Often, we choose finite words over

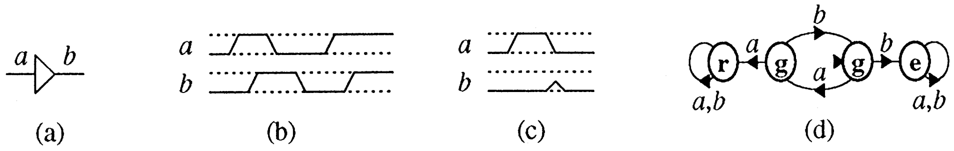

Fig. 1. Example: buffer specification.

a given alphabet $\mathscr{U}$, i.e., $\mathscr{E}=\mathscr{U}^*$. The execution sets of a process are then languages over $\mathscr{U}$, and $\mathscr{U}$ is called the *universe of actions*.

In a typical application, we represent waveforms of digital circuits by finite-word executions by associating the occurrence of an action to each signal transition in the circuit. We do not distinguish between rising and falling transitions. For instance, the waveform in Fig. 1(b) is represented by *ababab*.

Definition 4. A *process* over a set $\mathscr{E}$ is a pair (X,Y) of subsets of $\mathscr{E}$ such that $X\cup Y=\mathscr{E}$.

A process represents a contract between a device and its environment; this contract is from the device point of view. A process (X,Y) splits $\mathscr{E}$ into three disjoint subsets: $X\cap Y$, $\overline{X}$, and $\overline{Y}$, called *goals*, *errors*, and *rejects*, respectively. Executions in $X\cap Y$ are legal for both the device and the environment. Executions in $\overline{X}$ represent errors of the device. Finally, executions in $\overline{Y}$ represent bad behavior on the part of the environment, and, as such, can be rejected by the device. Executions in X are called *accessible* by the device, whereas those in Y are called *acceptable* by the device.

For example, the safety properties of the buffer in Fig. 1(a) can be represented by a process over $\mathscr{U}^*$, where $\mathscr{U}\supseteq\{a,b\}$. Refer to the finite automaton of Fig. 1(d), where the initial state is indicated by an incoming arrow; such automata are formally defined later. Execution *abab* is legal for both the device and the environment; thus it is a goal. Execution *abb* should be avoided by the device; thus it is an error. So is execution *abba*, since the second b is the fault of the device. Hazards, i.e., situations where an output transition is enabled and then disabled before being completed, are sometimes considered undesirable, since, as in Fig. 1(c), they may generate signal maxima or minima at intermediate voltages. If hazards are undesirable, execution *aa* should be avoided by the environment, to ensure that the enabled b transition is completed. Accordingly, execution *aa* is a reject.

In process spaces, several operations and correctness conditions from concurrency theory become elementary set operations, as in the definition below.

Definition 5. For processes $p=(X,Y)$, $p'=(X',Y')$, and $p''=(X'',Y'')$,
1. $p''=p\times p'$, read p'' is the *product* of p and p', if $X''=X\cap X'$ and $Y''=(Y\cap Y')\cup\overline{X}\cup\overline{X'}$.
2. $p'=-p$, read p' is the *reflection* of p, if $X'=Y$ and $Y'=X$.
3. p is *robust* if $\overline{Y}$ is empty.
4. $p\sqsubseteq p'$, read p' *refines* p, if $X\supseteq X'$ and $Y\subseteq Y'$.

Briefly, the motivation for these concepts is as follows. The product of p and p' represents the joint behavior of the devices of p and p'. Reflection rewrites the process-environment contract to the environment point of view. A process is robust if it has no rejects, and thus requires no guarantees from the environment. Finally, refinement formalizes that process p' is a 'satisfactory substitute' for process p: Process p' has fewer accessible executions and more acceptable executions than p. Equivalently, p' has more errors and fewer rejects.

The following property links robustness and refinement, and is important for verification applications.

Proposition 1. *For processes p and q,*

$$p \sqsubseteq q \Leftrightarrow -p \times q \text{ is robust.}$$

If its execution languages are regular, a process can be represented as a finite automaton, as follows.

Definition 6. A *process automaton* is a tuple $\mathscr{P} = \langle \mathscr{A}, Q_\mathrm{r}, Q_\mathrm{g}, Q_\mathrm{e} \rangle$ consisting of an action automaton $\mathscr{A} = \langle \Sigma, Q, I, E \rangle$ and three subsets of Q, where

1. Σ is a finite subset of $\mathscr{U}$.
2. Q_r, Q_g, and Q_e, whose elements are called *rejects*, *goals*, and *errors*, respectively, are mutually disjoint.
3. $Q = Q_\mathrm{r} \cup Q_\mathrm{g} \cup Q_\mathrm{e}$.
4. For each $q \in Q$ and $\sigma \in \Sigma$, there exists $q' \in Q$ such that $(q, \sigma, q') \in E$.

A process automaton is said to be *deterministic* if its action automaton is deterministic. We normally work with deterministic process automata. Nondeterministic process automata arise, for instance, as a result of hiding internal actions, but in those cases we prefer to determinize them before performing any other operations. From now on we use only deterministic process automata. Nondeterminism is treated in [14].

Each word from Σ^* represents a possible sequence of process actions. We want to classify each such sequence of actions as being a reject, a goal, or an error for the process. For this reason, we included the fourth property (*completeness*) in the definition above. Now every word from Σ^* is spelled by some initialized path that terminates in Q_r, Q_g, or Q_e.

Notice that the words spelled by the paths of an automaton have actions only from Σ, whereas our processes are over $\mathscr{U}^*$. We resolve this mismatch by labelling an arbitrary word a reject, goal, or error if its projection on Σ is a reject, goal, or error, respectively.

Definition 7. With each process automaton $\mathscr{P} = \langle \langle \Sigma, Q, I, E \rangle, Q_\mathrm{r}, Q_\mathrm{g}, Q_\mathrm{e} \rangle$ we associate a process $\mathbf{pr}\,\mathscr{P} = (X, Y)$ as follows. For every word $w \in \mathscr{U}^*$,

1. If the initialized path spelling $w \downarrow \Sigma$ in $\mathscr{A}$ ends in Q_g or Q_r, then $w \in X$.
2. If the initialized path spelling $w \downarrow \Sigma$ in $\mathscr{A}$ ends in Q_r, then $w \in \overline{Y}$.

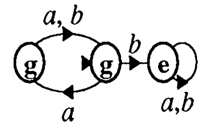

Fig. 2. Example for robustness.

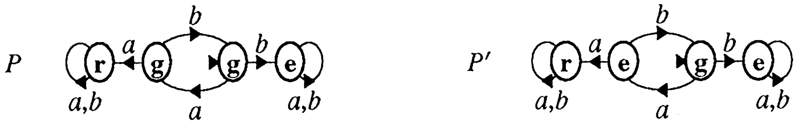

Fig. 3. Example for refinement.

For example, the process automaton in Fig. 1(d) represents the buffer process discussed previously.

Definition 8. Corresponding to the properties and operations of processes given in Definition 5, we define the following concepts for process automata.

1. The *product* of two process automata $\mathscr{P}^1 = \langle \mathscr{A}^1, Q^1_{\rm r}, Q^1_{\rm g}, Q^1_{\rm e} \rangle$ and $\mathscr{P}^2 = \langle \mathscr{A}^2, Q^2_{\rm r}, Q^2_{\rm g}, Q^2_{\rm e} \rangle$ is a process automaton $\mathscr{P}^1 \times \mathscr{P}^2 = \langle \mathscr{A}, Q_{\rm r}, Q_{\rm g}, Q_{\rm e} \rangle$ such that
   (a) $\mathscr{A} = \mathscr{A}^1 \times \mathscr{A}^2$.
   (b) $(q^1, q^2) \in Q_{\rm r}$ if and only if $q^1 \in Q^1_{\rm r} \wedge q^2 \in Q^2_{\rm r}$ or $q^1 \in Q^1_{\rm r} \wedge q^2 \in Q^2_{\rm g}$ or $q^1 \in Q^1_{\rm g} \wedge q^2 \in Q^2_{\rm r}$.
   (c) $(q^1, q^2) \in Q_{\rm g}$ if and only if $q^1 \in Q^1_{\rm g} \wedge q^2 \in Q^2_{\rm g}$.
   (d) $(q^1, q^2) \in Q_{\rm e}$ if and only if $q^1 \in Q^1_{\rm e}$ or $q^2 \in Q^2_{\rm e}$.
2. *Reflection* swaps $Q_{\rm r}$ and $Q_{\rm e}$; thus, $-\mathscr{P} = \langle \mathscr{A}, Q_{\rm e}, Q_{\rm g}, Q_{\rm r} \rangle$.
3. Process automaton $\mathscr{P} = \langle \mathscr{A}, Q_{\rm r}, Q_{\rm g}, Q_{\rm e} \rangle$ is *robust* if no state from $Q_{\rm r}$ is on an initialized path.
4. *Refinement* is a binary relationship $\sqsubseteq$ on process automata such that $\mathscr{P}^1 \sqsubseteq \mathscr{P}^2$ if and only if $-\mathscr{P}^1 \times \mathscr{P}^2$ is robust.

The automaton in Fig. 1(d) is not robust, since its reject state can be reached by, say, *aa*. A robust process for the buffer, this time ignoring hazards as ‘inertial’ models do, is represented in Fig. 2. There, none of the reachable states is a reject. In particular, the second *a* in *aa* cancels the first *a* by leading back to the initial state, a goal state.

The process automaton P' in Fig. 3 is yet another possible process for the buffer, which forbids stopping after *a*, since the device is supposed to respond by *b*. The state to which *a* leads is an error, making it illegal for the buffer device to complete its operation by stopping there. One can verify that $P \sqsubseteq P'$. Since P' has more errors, it is more determinate and more constrained than P. On the other hand, since *a* is accessible for P but not for P', P does not refine P'. Informally, P has the option of stopping after *a*, but P' does not allow this.

The process automaton operations correspond to process operations.

Proposition 2. *For process automata $\mathscr{P}^1$ and $\mathscr{P}^2$,*
1. $\mathbf{pr}(\mathscr{P}^1 \times \mathscr{P}^2) = \mathbf{pr}\,\mathscr{P}^1 \times \mathbf{pr}\,\mathscr{P}^2$.
2. $\mathbf{pr}(-\mathscr{P}^1) = -(\mathbf{pr}\,\mathscr{P}^1)$.
3. $\mathbf{pr}\,\mathscr{P}^1$ *is robust if and only if* $\mathscr{P}^1$ *is robust.*
4. $\mathbf{pr}\,\mathscr{P}^1 \sqsubseteq \mathbf{pr}\,\mathscr{P}^2$ *if and only if* $\mathscr{P}^1 \sqsubseteq \mathscr{P}^2$.

Process equality induces an equivalence relationship on process automata.

Proposition 3. *For process automata $\mathscr{P}^1$ and $\mathscr{P}^2$, $\mathbf{pr}\,\mathscr{P}^1 = \mathbf{pr}\,\mathscr{P}^2$ if and only if every state (q^1, q^2) of $\mathscr{P}^1 \times \mathscr{P}^2$ that appears on an initialized path satisfies $q^1 \in Q^1_{\mathrm{r}} \wedge q^2 \in Q^2_{\mathrm{r}}$ or $q^1 \in Q^1_{\mathrm{g}} \wedge q^2 \in Q^2_{\mathrm{g}}$ or $q^1 \in Q^1_{\mathrm{e}} \wedge q^2 \in Q^2_{\mathrm{e}}$.*

3. BDD implementation of process automata

Process automata and the operations mentioned in Section 2 are implemented in a BDD-based tool called FIREMAPS (for finitary and regular manipulation of processes and systems) [11]. Several other operations on process automata are also implemented in FIREMAPS, most notably minimization, determinization, and operations for projecting a process automaton on an alphabet and for hiding internal actions.

Binary decision diagrams (BDDs) are graph-based data structures for representing Boolean functions [3]. BDDs allow for greater average-case efficiency of the Boolean operations than do the traditional representations of Boolean functions. The tool uses a BDD library [2] which offers the basic routines for manipulating large Boolean functions.

The standard BDDs, used in FIREMAPS, express Boolean functions of the form $f : \{0,1\}^V \to \{0,1\}$, where V is a finite set, whose elements are referred to as Boolean variables. A *valuation* for V is a function $x : V \to \{0,1\}$, which gives a Boolean value to each variable. A valuation can be regarded as a binary vector with $|V|$ bits, where $|\cdot|$ denotes the cardinality of a set. The set of all possible valuations for V is denoted as $\{0,1\}^V$.

To manipulate process automata by Boolean functions, we encode the actions, states, and edges as binary vectors over some of the Boolean variables available, and we represent the finite sets of actions, states, and edges by characteristic functions of sets of binary vectors. Let us formalize this BDD representation, following loosely the treatment of symbolic analysis in [4]. Given a finite set $\mathscr{D}$, we encode it by an injective function $\pi_{\mathscr{D}} : \mathscr{D} \to \{0,1\}^{W_{\mathscr{D}}}$, where $W_{\mathscr{D}} \subseteq V$. Under this encoding, the set $\mathscr{D}$ is represented by a function $\chi_{\mathscr{D}} : \{0,1\}^V \to \{0,1\}$ such that $\chi_{\mathscr{D}}(x) = 1$ if and only if the components of x corresponding to $W_{\mathscr{D}}$ are the code-word of an element of $\mathscr{D}$.

Notice that the function $\chi_{\mathscr{D}}$ depends only on the variables in $W_{\mathscr{D}}$. However, we defined $\chi_{\mathscr{D}}$ over all variables in V rather than just $W_{\mathscr{D}}$ for two reasons: to follow closely the BDD implementation, and because we need conjunctions or disjunctions of functions that may depend on different variable sets.

In FIREMAPS, all actions are encoded over a set $W_{\mathscr{U}}$ of Boolean variables. Alphabets are represented by the corresponding characteristic functions.

The states of a process automaton $\langle\langle\Sigma, Q, I, E\rangle, Q_{\mathrm{r}}, Q_{\mathrm{g}}, Q_{\mathrm{e}}\rangle$ are represented over a set W_Q of variables, where $2^{|W_Q|} \geqslant |Q|$ and W_Q is disjoint from $W_{\mathscr{U}}$. To each process automaton, we associate functions for Q, I, $Q_{\mathrm{r}} \cup Q_{\mathrm{g}}$, and $Q_{\mathrm{e}} \cup Q_{\mathrm{g}}$, all these functions being from $\{0,1\}^{W_Q}$ to $\{0,1\}$.

The edge set of a process automaton is a subset of $Q \times \mathscr{U} \times Q$. The edges are represented over the variable set $W_Q \cup W_{\mathscr{U}} \cup W_Q'$ where the variables in W_Q' encode the 'next state', those in W_Q encode the 'current state', and those in $W_{\mathscr{U}}$ encode the action of an edge. The set W_Q' is disjoint from W_Q and $W_{\mathscr{U}}$, and there is a bijective function $s: W_Q \to W_Q'$ that defines the 'next state' encoding as the valuation $\pi_Q' : Q \to \{0,1\}^{W_Q'}$ that satisfies $\forall q \in Q,\ \xi \in W_Q$: $(\pi_Q'(q))(s(\xi)) = (\pi_Q(q))(\xi)$.

The product operation is implemented as follows. If $\mathscr{P}^1$ and $\mathscr{P}^2$ are the operands and $\mathscr{P}$ is the result, then

1. $\chi_\Sigma = \chi_{\Sigma^1} \vee \chi_{\Sigma^2}$.
2. $\forall \xi \in W_{Q^1}$: $s(\xi) = s^1(\xi)$ and $\forall \xi \in W_{Q^2}$: $s(\xi) = s^2(\xi)$.
3. $\chi_Q = \chi_{Q^1} \wedge \chi_{Q^2}$.
4. $\chi_{\mathrm{I}} = \chi_{\mathrm{I}^1} \wedge \chi_{\mathrm{I}^2}$.
5. $\chi_E = \chi_{E^1} \wedge \chi_{E^2}$.
6. $\chi_{Q_{\mathrm{r}} \cup Q_{\mathrm{g}}} = \chi_{Q_{\mathrm{r}}^1 \cup Q_{\mathrm{g}}^1} \wedge \chi_{Q_{\mathrm{r}}^2 \cup Q_{\mathrm{g}}^2}$.
7. $\chi_{Q_{\mathrm{e}} \cup Q_{\mathrm{g}}} = \chi_{Q_{\mathrm{e}}^1 \cup Q_{\mathrm{g}}^1} \wedge \chi_{Q_{\mathrm{e}}^2 \cup Q_{\mathrm{g}}^2} \vee \neg\chi_{Q_{\mathrm{r}}^1 \cup Q_{\mathrm{g}}^1} \vee \neg\chi_{Q_{\mathrm{r}}^2 \cup Q_{\mathrm{g}}^2}$.

This implementation of product follows the definition straightforwardly, except that the sets of state variables of $\mathscr{P}^1$ and $\mathscr{P}^2$ are not necessarily disjoint.

Sharing of state variables among different process automata can provide substantial savings, but requires certain compatibility conditions among the process automata involved and among their Boolean function representations. We do not go deeper into this issue, but we note that the product automaton is compatible in this sense with the factors, justifying the variable sharing among the product and the factors. The same holds for the result and the operand of reflection.

Reflection is implemented by simply swapping the functions for $Q_{\mathrm{r}} \cup Q_{\mathrm{g}}$ and $Q_{\mathrm{e}} \cup Q_{\mathrm{g}}$.

Robustness is verified by reachability analysis, which checks whether any reject state is reachable from the initial states. For variable set $U \subseteq V$ and for function $\chi : \{0,1\}^V \to \{0,1\}$, we use the notation $(\exists U : \chi)$ for function $\chi'(y) = (\exists y' \in \{0,1\}^V : \chi(y') \wedge (y'|_{V\setminus U} = y|_{V\setminus U}))$, where $\cdot|.$ is domain restriction; and, for variable set $U' \subseteq V$ and bijective function $s : U \to U'$ we use the notation $[s]\chi$ for the function obtained by simultaneously substituting in χ the variables from U by their images through s. For process automaton $\langle\langle\Sigma, Q, I, E\rangle, Q_{\mathrm{r}}, Q_{\mathrm{g}}, Q_{\mathrm{e}}\rangle$, let R denote the set of states currently reached, and let F denote the edge set without the action information, i.e., $F = \{(q,q') \mid \exists a \in \mathscr{U} : (q,a,q') \in E\}$. The reachability analysis algorithm is roughly as follows:

```
1. $\chi_F = (\exists W_{\mathscr{U}} : \chi_E)$
2. $\chi_R = \chi_{\mathrm{I}}$
3. repeat
```

```
4.    if ($\chi_R \wedge \chi_{Q_r} \neq 0$) then stop, NOT ROBUST
5.    $\chi_R := \chi_R \vee [s^{-1}](\exists W_Q : \chi_R \wedge \chi_F)$
6. until (no change in $\chi_R$)
7. stop, ROBUST
```

For process automata $\mathscr{P}^1$ and $\mathscr{P}^2$, we check refinement by checking robustness of $-\mathscr{P}^1 \times \mathscr{P}^2$. Thus refinement is also checked by reachability analysis.

With minor modifications, the algorithm above can be used to find and return a reject execution for a process, if that process is not robust.

4. Behavior automata

The present section is based on the work in [13, 14]. Liveness properties such as fairness and deadlock-freedom have been modeled traditionally by using explicit specifications of the infinite sequences of actions that may occur in a system. It has been considered that finitary specifications, consisting basically of the finite words that may occur in a system, are not sufficiently powerful for deciding liveness properties [1]. Without challenging the statements in [1], we argued in [14] that implicit liveness properties can be associated in a sensible manner to structures that consist of a finitary language and two alphabets. We also proposed a relative liveness condition whose verdict is uniquely determined by the finitary structures involved. If the language is regular, such a structure can be represented by a finite automaton, called a *behavior automaton*, and the condition can be checked directly on the automata. We have implemented the test for our liveness condition in an experimental tool, called ACUTE (for automaton checks using transition enablings).

Definition 9. A *trace structure* is a triple $\langle \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, L \rangle$ consisting of two disjoint sets Σ_I and Σ_O, called the sets of *input* and *output* actions, respectively, and a prefix-closed, non-empty language L of finite words from $(\Sigma_\mathrm{I} \cup \Sigma_\mathrm{O})^*$.

For the formal treatment, we make explicit the liveness properties of a trace structure by associating a language of finite and infinite sequences of actions to such a structure.

Definition 10. For trace structure $\langle \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, L \rangle$, word u from L, and action a from $(\Sigma_\mathrm{I} \cup \Sigma_\mathrm{O})$, *u enables a* if $ua \in L$. An *output trap* is either
1. a (finite) word $u \in L$ that enables no output, or
2. an infinite sequence e of actions from $\Sigma_\mathrm{I} \cup \Sigma_\mathrm{O}$ such that every (finite) prefix of e is in L, and any output that is enabled by infinitely many prefixes of e also appears infinitely many times in e.

We want to capture the following intuitive concepts of violation of liveness. First consider a finite sequence u in the language L^1 of a trace structure $T^1 = \langle \Sigma_\mathrm{I}^1, \Sigma_\mathrm{O}^1, L^1 \rangle$ representing the specification of a process. If a is an output of the process, and ua

is in L^1, then a is enabled in T^1 after u, i.e., output a is expected after u. Suppose trace structure $T^2 = \langle \Sigma_I^2, \Sigma_O^2, L^2 \rangle$ represents a proposed implementation of T^1. Suppose further that u is in L^2, but ua is not. Then, normally, T^2 is not a good implementation of T^1, because a is not produced after u, as required by T^1. Secondly, consider again a specification $T^1 = \langle \Sigma_I^1, \Sigma_O^1, L^1 \rangle$ and an infinite sequence e over $(\Sigma_I^1 \cup \Sigma_O^1)$ in which infinitely many prefixes u enable an output a, in the sense that $ua \in L^1$. This enabling creates a 'pressure' on the process to produce output a, and the pressure can be 'relieved' only if a appears infinitely many times in e. Suppose, however, that in the implementation $T^2 = \langle \Sigma_I^2, \Sigma_O^2, L^2 \rangle$ only finitely many, possibly zero, prefixes of u enable a, but u has all its prefixes in L^2. Since no pressure builds up for a, the implementation is not under any obligation to deliver a infinitely many times. As a result, the pressure apparent in the specification is not relieved and the specification is 'starved' for a. This is also a violation of liveness. More details are given in [13].

Formally, our liveness condition, called *traplock-freedom*, is a relative condition comparing two trace structures. Some examples will be discussed later.

Definition 11. For trace structures $T^1 = \langle \Sigma_I^1, \Sigma_O^1, L^1 \rangle$ and $T^2 = \langle \Sigma_I^2, \Sigma_O^2, L^2 \rangle$, we have $T^1 \sqsubseteq_{tf} T^2$, read T^2 is *traplock-free* for T^1, if for every finite or infinite sequence e of actions from $\Sigma_I^1 \cup \Sigma_O^1 \cup \Sigma_I^2 \cup \Sigma_O^2$ that satisfies
1. $e \downarrow (\Sigma_I^2 \cup \Sigma_O^2)$ is an output trap for T^2, and
2. every prefix of $e \downarrow (\Sigma_I^1 \cup \Sigma_O^1)$ is in L^1,

we have that

$$e \downarrow (\Sigma_I^1 \cup \Sigma_O^1) \text{ is also an output trap for } T^1.$$

Next we define automata for representing trace structures whose languages are regular.

Definition 12. A *behavior automaton* is a triple $\mathscr{B} = \langle \mathscr{A}, \Sigma_I, \Sigma_O \rangle$ consisting of an action automaton $\mathscr{A} = \langle \Sigma, Q, I, E \rangle$ and two subsets Σ_I and Σ_O of Σ, where
1. $\Sigma_I \cap \Sigma_O = \emptyset$.
2. $\Sigma_I \cup \Sigma_O = \Sigma$.
3. $\mathscr{A}$ is deterministic.

With each behavior automaton $\mathscr{B} = \langle \mathscr{A}, \Sigma_I, \Sigma_O \rangle$ we associate a trace structure $\mathbf{tr}\, \mathscr{B} = \langle \Sigma_I, \Sigma_O, L \rangle$, where L is the language of $\mathscr{A}$.

A liveness condition is also defined on behavior automata.

Definition 13. The *composite* of two behavior automata $\mathscr{B}^1 = \langle \mathscr{A}^1, \Sigma_I^1, \Sigma_O^1 \rangle$ and $\mathscr{B}^2 = \langle \mathscr{A}^2, \Sigma_I^2, \Sigma_O^2 \rangle$ is the behavior automaton $\mathscr{B} = \langle \mathscr{A}, \Sigma_I, \Sigma_O \rangle$ where $\mathscr{A} = \langle \Sigma, Q, I, E \rangle$ and
1. $\mathscr{A} = \mathscr{A}^1 \times \mathscr{A}^2$.
2. $\Sigma_I = (\Sigma_I^1 \cup \Sigma_I^2) \backslash (\Sigma_O^1 \cup \Sigma_O^2)$.
3. $\Sigma_O = \Sigma_O^1 \cup \Sigma_O^2$.

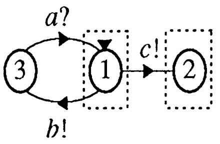

Fig. 4. Examples of knots corresponding to finite sequences.

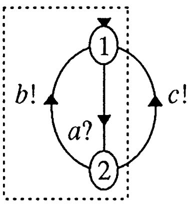

Fig. 5. Example of knot corresponding to an infinite sequence.

The *projection* of a strongly connected subgraph H of $\mathscr{B}$ on $\mathscr{B}^1$ is obtained by deleting the states of $\mathscr{B}^2$ from the states and edges of H. The projection on $\mathscr{B}^2$ is defined symmetrically.

For convenience, we identify inputs and outputs in a behavior automaton by appending '?' and '!', respectively.

A strongly connected subgraph of a behavior automaton is called a *knot* if it has at least one state and that state is on an initialized path. The states in boxes in Fig. 4 are knots corresponding to finite sequences in the usual manner. Some of the corresponding sequences are ε and ba for the left knot and c and $babac$ for the right knot. The subgraph in the box in Fig. 5 is a knot corresponding to the infinite sequence $abab\dots$.

A knot *enables* an action if that action is on an edge that leaves a state of the knot, and *fires* an action if that action is on an edge in the knot. A knot is an *output trap* if it fires all the actions that it enables. Several examples are given in Fig. 6. In the first row, the knot on the left is an output trap, since it fires c and it enables no other output; the knot on the right is not an output trap, since it enables output c but it does not fire it. In the second row, both knots enable c but neither knot fires c. The left knot is an output trap because there c is an input, but the right knot is not an output trap, since c is an output.

For behavior automata $\mathscr{B}^1$ and $\mathscr{B}^2$, $\mathscr{B}^2$ is *traplock-free* for $\mathscr{B}^1$ if, for every knot G in the composite of $\mathscr{B}^1$ and $\mathscr{B}^2$ such that the projection of G on $\mathscr{B}^2$ is an output trap in $\mathscr{B}^2$, the projection of G on $\mathscr{B}^1$ is an output trap in $\mathscr{B}^1$.

Fig. 7 shows an example of deadlock as a violation of traplock-freedom. The automaton on the left represents a specification, and the automaton on the right an implementation. From the initial state, an internal action c may occur on the right, stranding the implementation in the marked output trap, but leaving the specification in the initial state, which is not an output trap and thus is supposed to be left eventually. Fig. 8 shows an example of unfairness as a violation of traplock-freedom. Infinite sequence

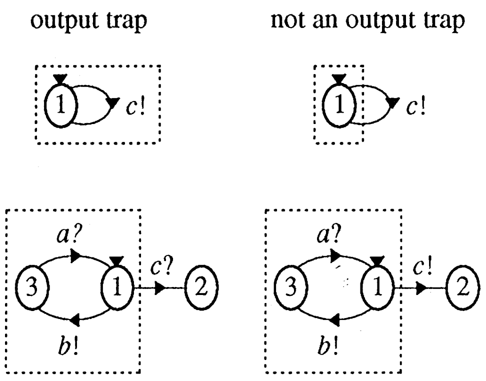

Fig. 6. Examples of output traps and knots that are not output traps.

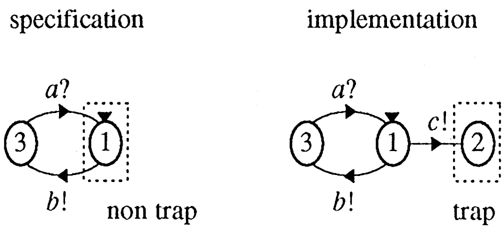

Fig. 7. Traps and deadlock.

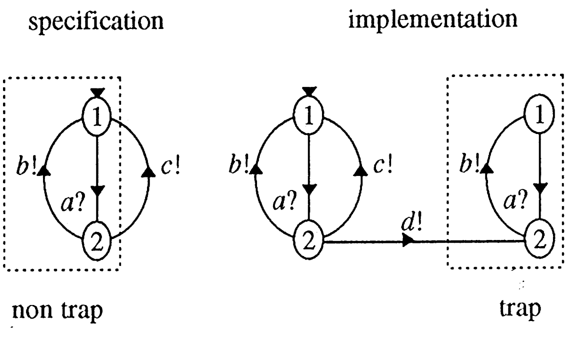

Fig. 8. Traps and unfairness.

adbababa..., corresponding to the trap on the right, projects on the specification alphabet as *ababaha*..., which is not a trap in the specification, since it is unfair to c. The corresponding specification knot, marked by a box, should be eventually left by firing c, but, as shown, this might not happen in the implementation.

The two definitions of our liveness condition are related as follows.

Theorem 1. *For behavior automata* $\mathscr{B}^1$ *and* $\mathscr{B}^2$, $\mathscr{B}^2$ *is traplock-free for* $\mathscr{B}^1$ *if and only if* $\mathbf{tr}\, B^2$ *is traplock-free for* $\mathbf{tr}\, B^1$.

It follows from the theorem that the information needed to decide traplock-freedom on behavior automata is contained in their finitary trace structures. For classes of systems in which the implicit liveness properties are appropriate, the users need not specify explicitly the infinite sequences of actions that may occur in their systems. It suffices to provide the input and output alphabets and the languages of finite words that may occur in such systems. This way we specify liveness properties implicitly.

5. Delay-insensitivity of asynchronous networks

This section follows closely the work in [4, 5, 18]. We study asynchronous sequential networks consisting of sets of modules interconnected by wires. Modules have binary inputs and outputs and, normally, have binary internal state variables. To hide the details of the internal design of a module, however, we represent its internal state by a single multivalued variable.

Definition 14. A *module* is a sequential machine $M = \langle \mathscr{S}, \mathscr{X}, y, \mathscr{Z}, \delta, \lambda \rangle$, where
1. $\mathscr{S}$ is a finite, nonempty set of *module internal states.*
2. $\mathscr{X} = \{x_1, \ldots, x_m\}$, $m \geqslant 0$, is a finite set of *module input variables*; also, $x = (x_1, \ldots, x_m)$ is the m-tuple of module input variables.
3. y is the *module internal state variable.*
4. $\mathscr{Z} = \{z_1, \ldots, z_p\}$, $p \geqslant 0$, is a finite set of *module output variables*; also, $z = (z_1, \ldots, z_p)$ is the p-tuple of module output variables.
5. δ is the *module excitation function*, $\delta : \{0,1\}^m \times \mathscr{S} \to 2^{\mathscr{S}} - \{\emptyset\}$, and for any $a \in \{0,1\}^m$ and $b \in \mathscr{S}$, either $\delta(a,b) = \{b\}$ or $b \notin \delta(a,b)$.
6. $\lambda = (\lambda_1, \ldots, \lambda_p)$ is the *module output function*, $\lambda : \mathscr{S} \to \{0,1\}^p$.

With each module we associate a directed graph, the *module graph* defined below. The vertices of this graph are of three types: There are m input vertices, one internal state vertex, and p output vertices. For convenience, we identify these vertices with their corresponding module variables. Thus $G = \langle \mathscr{V}, \mathscr{E} \rangle$, where
1. $\mathscr{V} = \mathscr{X} \cup \{y\} \cup \mathscr{Z}$ is the set of *module vertices*, and
2. $\mathscr{E} = (\mathscr{X} \times \{y\}) \cup (\{y\} \times \mathscr{Z})$ is the set of *module edges.*

The definition above permits us to represent arbitrary nondeterministic sequential machines of the Moore type [9]. Examples of such machines are delays, inverters, forks, arbitrary m-input/n-output logic gates, latches, counters, C-elements, and arbiters. A collection of such modules can be connected by wires to form a network, as defined below. In the sequel, different modules are distinguished by superscripts.

Definition 15. A *network* is a pair $N = \langle \mathscr{M}, G \rangle$, where
1. $\mathscr{M} = \{M^1, \ldots, M^n\}$, $n \geqslant 1$, is a set of modules.
2. $G = \langle \mathscr{V}, \mathscr{E} \rangle$ is a connected directed graph, the *network graph*, where
   (a) $\mathscr{V} = \bigcup_{i=1}^{n} \mathscr{V}^i$ is the set of *network vertices.*

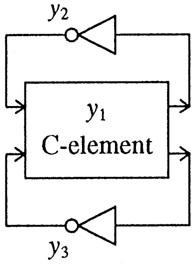

Fig. 9. A network.

(b) $\mathscr{E}$ is the set of *network edges*, such that
  (i) $\mathscr{E} \supseteq \bigcup_{i=1}^{n} \mathscr{E}^i$ and $\mathscr{E} \subseteq (\bigcup_{i=1}^{n} \mathscr{Z}^i \times \bigcup_{i=1}^{n} \mathscr{X}^i) \cup \bigcup_{i=1}^{n} \mathscr{E}^i$,
  (ii) for each module input vertex $x \in \mathscr{V}$, there exists exactly one module output vertex $z \in \mathscr{V}$ such that $(z,x) \in \mathscr{E}$,
  (iii) for each module output vertex $z \in \mathscr{V}$, there exists exactly one module input vertex $x \in \mathscr{V}$ such that $(z,x) \in \mathscr{E}$.

We denote the set of network edges that are not internal module edges by $\mathscr{K}$, and refer to these edges as *connections*. An example of a network is given in Fig. 9, showing the modules, connections, and module variables.

The set of *state variables* of the network N is $\mathscr{Y} = \{y^1, \dots, y^n\}$. The set of *states* of N is $\mathscr{S} = \mathscr{S}^1 \times \cdots \times \mathscr{S}^n$. If $s \in \mathscr{S}$, the ith component of s is denoted by s_i.

Let $y^i \in \mathscr{Y}$. The *network excitation function* $\Delta_i : \mathscr{S} \to 2^{\mathscr{S}^i} - \{\emptyset\}$ of y^i, is the module excitation function δ^i of M^i with arguments changed as follows. If (z_k^h, x_l^i) is a connection, then the lth input argument of δ^i is $\lambda_k^h(y^h)$. Since λ^h depends solely on y^h, the excitation function Δ_i becomes a function of y^h. For $y^i \in \mathscr{Y}$ and $s \in \mathscr{S}$, the *excitation* of y^i in state s is denoted by S_i, and is defined to be $S_i = \Delta_i(s)$. A state s is *stable* if $S_i = \{s_i\}$ for all i; otherwise, s is *unstable*. We denote the set of unstable state variables in state s by

$$\mathscr{U}(s) = \{y^i \in \mathscr{Y} \mid S_i \neq \{s_i\}\}.$$

When a network is in a given state $s = (s_1, \dots, s_n)$ and has some unstable variables, any one of the unstable variables may change to any module state in its excitation. The state t so reached is a possible next state of the network. Under these conditions we say that $s\mathscr{R}t$, i.e., s is related to t by the binary relation $\mathscr{R}$ on $\mathscr{S}$. This relation is known as the *general single-winner* relation GSW [4].

We now define the behavior of a network. It is usually assumed that each of the modules in the network has a single initial state.

Definition 16. The *network automaton* of a network N is an action automaton $\mathscr{N} = \langle \Sigma, Q, I, E \rangle$, where
1. $\Sigma = \mathscr{Y}$, i.e., the actions are the module variables.
2. $Q = \mathscr{S}$ is the set of network states.

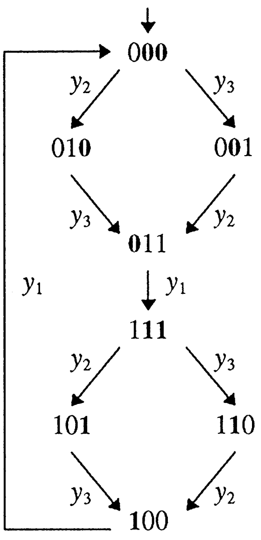

Fig. 10. Network automaton.

3. I is a singleton set, the ordered tuple of the initial states of the modules.
4. E is derived from the GSW relation $\mathscr{R}$ as follows: If $s\mathscr{R}t$ and the variable in which s and t differ is y^i, then $(s, y^i, t) \in E$, and there are no other edges.

The network automaton for the network of Fig. 9 is shown in Fig. 10, where unstable entries are shown in boldface type.

An important question in the theory of asynchronous circuits is the problem of sensitivity of network behaviors to delays in the modules and connecting wires [4, 5, 16–18]. In our formal model of a module, we have effectively associated a delay with each module. A module variable y^i has a value s_i in a given state s of the network. If y^i is unstable, then its excitation S_i differs from $\{s_i\}$. According to our model, y^i can change to a state in S_i at any time, and this represents an arbitrary delay of the module. There are no delays associated with the module outputs, since the module state completely determines the values of the module outputs.

In some design styles, assumptions are made about the relative sizes of module delays; such designs improve efficiency, but are sensitive to delay fluctuations. If the correctness of the network behavior is independent of the module delays, then the network is called *speed-independent*. Connecting wires may also have appreciable delays. If the correctness of the network behavior is independent of the delays of the modules and wires, the network is called *delay-insensitive*.

In [5, 18] we take the following approach to the study of delay-insensitivity of networks. We assume that we have a speed-independent network, i.e., a network whose behavior, as defined in our model, is acceptable with respect to some specification. We then wish to know whether this behavior is still acceptable in the presence of arbitrary delays in the connecting wires. Note that, in our model, the connecting wires have no delays. However, we can model wire delays by inserting (zero or more) delay modules in each connection. A delay module has one binary input x, one binary output z and one binary internal variable y. The excitation is always equal to the input, and the output always agrees with y. Thus, the output always tries to follow the input, after an arbitrary delay.

Suppose we have a network N and a network $\hat{N}$ obtained from N by the insertion of a number of delays in the connections of N. Then $\hat{N}$ is called a *delay-extension* of N. We need to compare the behaviors of N and $\hat{N}$. Formally, $\hat{N}$ has more state variables, since each inserted delay module has a state variable. We 'project out' these delay module variables, and then compare the state of the remaining variables with the state of those of N. We consider N, started in a given state, to be delay-insensitive if every delay-extension $\hat{N}$, started in the corresponding state, is bisimilar [7] to N, after the added delay variables have been projected out.

Testing whether a network is delay-insensitive according to our definition is an infinite process, since there is an infinite number of delay extensions of any network. Fortunately, there is an equivalent definition that requires only a finite test.

Let N be a network and $\mathcal{N}$ its network automaton. A state s of $\mathcal{N}$ is *semi-modular* if, for all $t \in \mathcal{S}$, if $s\mathcal{R}t$ and the variable in which s and t differ is y^i, then the excitations S_j and T_j satisfy $S_j \subseteq T_j$, for all $j \neq i$ such that $y^j \in \mathcal{U}(s)$. In words, if y^i changes causing a transition from state s to state t, and y^j, $j \neq i$, is unstable in state s and can change to the value v_j, then y^j should still be unstable in state t, and should still be able to change to v_j. Thus a transition that is enabled for y^j, cannot be disabled by a transition on another variable y^i. A network automaton $\mathcal{N}$ is semi-modular if each of its states is semi-modular.

Two modules in a network are *adjacent* if the output of one module is connected to the input of the second module. A network is *delay-dense* if of every pair of adjacent modules, one module is a delay module. We can make any network delay-dense by inserting one delay module in each connection of the network. Obviously, delay-dense networks are appropriate models for the study of delay-insensitivity in asynchronous circuits.

The following theorem [5] relates delay-insensitivity and semi-modularity.

Theorem 2. *A delay-dense network N is delay-insensitive iff its network automaton $\mathcal{N}$ is semi-modular.*

6. Conclusions

We have described three applications of automaton theory to the theory of concurrent processes and asynchronous circuits. These applications lead to new operations on automata and their languages. These include testing for robustness and refinement in process automata, for traplock-freedom in behavior automata, and for semi-modularity in network automata. We have described formally a BDD-based implementation of process automata and their operations in the verification tool FIREMAPS.

The results in Sections 2 and 3 have been applied to the verification of several experimental asynchronous circuits; also see [8, 11–13, 15].

References

[1] D.L. Black, On the existence of delay-insensitive fair arbiters: trace theory and its limitations, Distributed Computing 1 (1986) 205–225.

[2] K.S. Brace, R.L. Rudell, R.E. Bryant, Efficient implementation of a BDD package, Proc. 27th ACM/IEEE Design Automation Conf., 1990, pp. 40–45.

[3] R.E. Bryant, Graph based algorithms for Boolean function manipulation, IEEE Trans. Comput. C-35 (1986) 677–691.

[4] J.A. Brzozowski, C.-J.H. Seger, Asynchronous Circuits, Springer-Verlag, 1995.

[5] J.A. Brzozowski, H. Zhang, Delay-insensitivity and semi-modularity, in: Formal Methods in System Design, to appear. Also Research Report CS-97-11, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1997.

[6] S. Eilenberg, Automata, Languages, and Machines, vol. A, Academic Press, New York, 1974.

[7] R. Milner, Communication and Concurrency, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[8] C.E. Molnar, I.W. Jones, B. Coates, J. Lexau, A FIFO ring oscillator performance experiment, Proc. Internat. Symp. on Advanced Research in Asynchronous Circuits and Systems, 1997.

[9] E.F. Moore, Gedanken experiments on sequential machines, Automata studies, Ann. Math. Studies 34 (1956) 129–153.

[10] R. Negulescu, Process spaces, Research Report CS-95-48, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1995.

[11] R. Negulescu, A technique for finding and verifying speed-dependences in gate circuits, Proc. ACM/IEEE Internat. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems, 1997.

[12] R. Negulescu, Event-driven verification of switch-level correctness concerns, Proc. Internat. Conf. on Application of Concurrency to System Design, 1998.

[13] R. Negulescu, Process spaces and formal verification of asynchronous circuits, Ph.D. Thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1998. http://www.macs.ece.mcgill.ca/~radu/ps.html

[14] R. Negulescu, J.A. Brzozowski, Relative liveness: from intuition to automated verification, Formal Methods System Des. 12 (1998) 73–115.

[15] R. Negulescu, A. Peeters, Verification of speed-dependences in single-rail handshake circuits, Proc. Internat. Symp. on Advanced Research in Asynchronous Circuits and Systems, 1998.

[16] J.T. Udding, Classification and composition of delay-insensitive circuits, Ph.D. Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, Eindhoven, The Netherlands, 1984.

[17] J.T. Udding, A formal model for defining and classifying delay-insensitive circuits and systems, Distributed Computing 1 (1986) 197–204.

[18] H. Zhang, Delay-insensitive networks, MMath. Thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1997.