A Concurrent Specification of an Incremental DFA Minimisation Algorithm

Tinus Strauss, Derrick G. Kourie, and Bruce W. Watson

FASTAR, University of Pretoria
Pretoria, South Africa
{tstrauss,dkourie,bwatson}@cs.up.ac.za
http://www.fastar.org

Abstract. In this paper a concurrent version of a published sequential incremental deterministic finite automaton minimisation algorithm is developed. Hoare's Communicating Sequential Processes (CSP) is used as the vehicle to specify the concurrent processes.
The specification that is proposed is in terms of the composition of a number of concurrent processes, each corresponding to a pair of nodes for which equivalence needs to be determined. Each of these processes are again composed of several other possibly concurrent processes.
To facilitate the specification, a new CSP concurrency operator is defined which, in contrast to the conventional CSP concurrency operator, does not require all processes to synchronise on common events. Instead, it is sufficient for any two (or more) processes to synchronise on such events.

Keywords: DFA minimisation, concurrency, CSP, automata

1 Introduction

As pointed out in [10], two contemporary trends drive the need for the development of concurrent implementations of automata algorithms. On the one hand, finite automaton technology is being applied to ever-larger applications. On the other hand, hardware is tending towards ever-increasing support for concurrent processing. Chip multiprocessors [7], for example, implement multiple CPU cores on a single die. Additionally, scale-out systems [1] – collections of interconnected low-cost computers working as a single entity – also provide parallel processing facilities. These hardware developments present the challenging task of producing quality concurrent software [6,11,12].

The problem of minimising a finite state automaton has been studied quite extensively over the years and many algorithms have been proposed to address this problem. See [14, Chap. 7] for a comprehensive coverage of the area.

Previous parallel algorithms that have been proposed include [5,8,13]. These algorithms typically depend on a specific parallel computational models. In the present case, we abstract away from these considerations and model the algorithm as a process in the Communicating Sequential Processes (CSP) process algebra. Our purpose is to expose maximally the possibilities for concurrency inherent in the problem itself—at least to the extent that these possibilities may be latent in the sequential algorithm. As a consequence, we are not concerned with issues such as allocation of processes to processors, determination of which processes could be coalesced into a single process to limit context switching, etc. These are regarded as implementation issues for later consideration.

Tinus Strauss, Derrick G. Kourie, Bruce W. Watson: A Concurrent Specification of an Incremental DFA Minimisation Algorithm, pp. 218–226.
Proceedings of PSC 2008, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04145-1 © Czech Technical University in Prague, Czech Republic

The article is structured as follows. Section 2 gives the preliminaries of the problem domain under consideration, and the sequential algorithmic solution to the problem is given in section 3. A very brief account is given in section 4 of CSP. We also introduce a so-called optional-parallel operator that will be used. Section 5 then provides a concurrent specification to the problem, before a brief conclusion in the final section.

2 Preliminaries

Throughout this paper, we will consider a specific *DFA* $(Q, \Sigma, \delta, q_0, F)$ where Q is the finite set of states, Σ is the input alphabet, $\delta : Q \times \Sigma \to Q$ is the transition function, $q_0 \in Q$ is the start state, and $F \subseteq Q$ is the set of *final* states. We further assume that all states in the automaton are reachable from q_0. The size of a *DFA*, $|(Q, \Sigma, \delta, q_0, F)|$, is defined as the number of states, $|Q|$.

To make some definitions simpler, we will use the shorthand Σ_q to refer to the set of all alphabet symbols which appear as out-transition labels from state q. Sometimes when it is the case that $\Sigma_p = \Sigma_q$ we will write Σ_{pq} instead of Σ_p or Σ_q to emphasise their equality.

We take $\delta^* : Q \times \Sigma^* \to Q$ to be the transitive closure of δ defined inductively (for state q) as $\delta(q, \varepsilon) = q$ and (for $a \in \Sigma_q$, $w \in \Sigma^*$) $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$.

The right language of a state q, written $\overrightarrow{\mathcal{L}}(q)$, is the set of all words spelled out on paths from q to a final state. Formally, $\overrightarrow{\mathcal{L}}(q) = \{\, w \mid \delta^*(q, w) \in F \,\}$. With the inductive definition of δ^*, we can give an inductive definition of $\overrightarrow{\mathcal{L}}$:

$$\overrightarrow{\mathcal{L}}(q) = \left[\bigcup_{a \in \Sigma_q} \{a\} \overrightarrow{\mathcal{L}}(\delta(q, a)) \right] \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{if } q \notin F \end{cases}$$

We define predicate *Equiv* to be 'equivalence' of states:

$$Equiv(p, q) \equiv \overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q)$$

With the inductive definition of $\overrightarrow{\mathcal{L}}$, we can rewrite *Equiv* as follows:

$$\begin{aligned} Equiv(p, q) \equiv\ & (p \in F \equiv q \in F) \wedge \Sigma_p = \Sigma_q \wedge \\ & \langle \forall\ a \in \Sigma_p \cap \Sigma_q : Equiv(\delta(p, a), \delta(q, a)) \rangle \end{aligned} \tag{1}$$

The primary definition of minimality of a *DFA* M is:

$$\langle \forall\ M' : M' \text{ is equivalent to } M : |M| \leq |M'| \rangle$$

where equivalence of *DFA*s means that they accept the same language. Using right languages, minimality can also be written as the following predicate:

$$\langle \forall\ p, q \in Q : p \neq q : \neg Equiv(p, q) \rangle$$

Equiv indicates whether two states are interchangeable. If they are, then one can be eliminated in favour of the other. (Of course, in-transitions to the eliminated state are redirected to the equivalent remaining one.) This reduction step is not addressed here.

3 The Sequential Algorithm

This section briefly details the sequential algorithm [15] for which we intend to provide a CSP model in the forthcoming sections. The algorithm is different from other minimisation algorithms in the sense that it is incremental, i.e. it may be halted at any time, yielding a partially minimised automaton.

From the problem of deciding the structural equivalence of two types, it is known that equivalence of two states can be computed recursively by turning the mutually recursive set of equivalences *Equiv* into a functional program. If the definition were to be used directly as a functional program, there is the possibility of non-termination in cyclic automata. In order for the functional program to work, it takes a third parameter along with the two states. An invocation $equiv(p, q, \emptyset)$ returns via the local variable *eq* the truth value of $Equiv(p, q)$. During the recursion, it assumes that two states are equivalent (by placing the pair of states in S, the third parameter) until shown otherwise.

Since it is known that the depth of recursion can be bounded by the larger of $|Q|-2$ and 0 (expressed as $(|Q|-2)\,\mathbf{max}\,0$) without affecting the result [14, §7.3.3], we add a parameter k to function *equiv* such that an invocation $equiv(p, q, \emptyset, (|Q|-2)\,\mathbf{max}\,0)$ returns $Equiv(p, q)$.

Purely for efficiency, the third parameter S is made a global variable. We assume that S is initialized to $\emptyset$. When $S = \emptyset$, an invocation $equiv(p, q, (|Q|-2)\,\mathbf{max}\,0)$ returns $Equiv(p, q)$; after such an invocation $S = \emptyset$.

Algorithm 3.1 (Pointwise computation of $Equiv(p, q)$):

```
func equiv(p, q, k) →
    if k = 0 → eq := (p ∈ F ≡ q ∈ F)
    [] k ≠ 0 ∧ {p, q} ∈ S → eq := true
    [] k ≠ 0 ∧ {p, q} ∉ S →
       eq := (p ∈ F ≡ q ∈ F) ∧ (Σ_p = Σ_q);
       S := S ∪ {{p, q}};
       for a : a ∈ Σ_p ∩ Σ_q →
           eq := eq ∧ equiv(δ(p, a), δ(q, a), k − 1)
       rof;
       S := S \ {{p, q}}
    fi;
    return eq
cnuf
```

The function *equiv* can be used to compute the relation (i.e. set of pairs) *Equiv*. In variable G, we maintain a set consisting of the pairs of states known to be inequivalent (*distinguished*), while in H, we accumulate pairs of states belonging to the set *Equiv*. To initialize G and H, we note that final states are never equivalent to nonfinal ones, and that a state is always equivalent to itself. Since *Equiv* is an equivalence relation, we ensure that H is transitive at each step. Finally, we have global variable S used in Algorithm 3.1:

Algorithm 3.2 (Computing *Equiv*):

$S, G, H := \emptyset, ((Q \setminus F) \times F) \cup (F \times (Q \setminus F)), \{\, (q, q) \mid q \in Q \,\}$;
{ invariant: $G \subseteq \neg Equiv \wedge H \subseteq Equiv$ }
do $(G \cup H) \neq Q \times Q \rightarrow$
  let $p, q : (p, q) \in ((Q \times Q) \setminus (G \cup H))$;
  if $equiv(p, q, (|Q| - 2)\,\mathbf{max}\,0) \rightarrow$
    $H := H \cup \{(p, q), (q, p)\}$;
    $H := H^+$
  $[\!]$ $\neg equiv(p, q, (|Q| - 2)\,\mathbf{max}\,0) \rightarrow$
    $G := G \cup \{(p, q), (q, p)\}$
  fi
od; { $H = Equiv$ }
merge states according to H
{ $(Q, \Sigma, \delta, q_0, F)$ is minimal }

The repetition in this algorithm can be interrupted and the partially computed H can be safely used to merge states.

4 CSP

Of the many process algebras that have been developed to concisely and accurately model concurrent systems, we have selected CSP [4,3,9] as a fairly simple and easy to use notation. It is arguably better known and more widely used than most other process algebras. Below, we first briefly introduce the conventional CSP operators that are used in the article. Thereafter we also introduce the so-called optional parallel operator—a new proposed CSP operator [2].

4.1 Introductory Remarks

CSP is concerned with specifying a system of concurrent sequential processes (hence the CSP acronym) in terms of sequences of atomic events, called traces. In fact, the semantics of a concurrent system is seen as being precisely described by the set of all possible traces that characterise such as system. A fundamental assumption is that events are instantaneous and atomic, i.e. they cannot occur concurrently. Various operators are available to describe the sequence in which events may occur, as well as to connect processes. Table 1 briefly outlines the operators used in this article.

Full details of the operator semantics and laws for their manipulation are available in [4,3,9]. Note that *SKIP* designates a special process that engages in no further event, but that simply terminates successfully. Parallel synchronization of processes means that if $A \cap B \neq \emptyset$, then process $(x?A \rightarrow P(x)) \;||\; (y?B \rightarrow Q(y))$ engages in some nondeterministically chosen event $z \in A \cap B$ and then behaves as the process $P(z) \;||\; Q(z)$. However, if $A \cap B = \emptyset$ then deadlock results. A special case of such parallel synchronization is the process $(b!e \rightarrow P) \;\underset{\alpha(b)}{||}\; (b?x \rightarrow Q(x))$, where $\alpha(b)$ denotes the alphabet on channel b. This should be viewed as a process that engages in the event $b.e$ and thereafter behaves as the process $P \;\underset{\alpha(b)}{||}\; Q(e)$. Note that parallel composition that involves more than two processes, requires that all processes always synchronise on common events. If they do not, then deadlock occurs. However, in the present context, it was considered desirable to introduce an alternative operator, called the optional parallel operator, that relaxes this requirement.

$a \rightarrow P$	event a then process P
$a \rightarrow P \vert b \rightarrow Q$	a then P choice b then Q
$x?A \rightarrow P(x)$	choice of x from set A then $P(x)$
$P \underset{X}{\vert\vert} Q$	P in parallel with Q
	Synchronize on events in set X
$b!e$	on channel b output event e
$b?x$	from channel b input to variable x
$P;Q$	process P followed by process Q
$P \vert\vert\vert Q$	process P interleave process Q
$P \triangle Q$	process P interrupted by process Q
$P \Box Q$	external choice between processes P and Q
$P \sqcap Q$	internal choice between processes P and Q

Table 1. Selected CSP Notation

4.2 Optional parallel operator

To position the new optional parallel operator, consider first the definition of the *generalised parallel* operator. The definition is expressed in terms of a so-called step law. The step law describes the initial actions in which the process may engage and then, for each possible initial action, it defines the behaviour of the process following that action. Suppose $R_1 = ?x : A_1 \rightarrow R_1'$ and $R_2 = ?x : A_2 \rightarrow R_2'$. Referring to Figure 1 the $\underset{X}{\vert\vert}$ -step law, provided by Roscoe [9, §2.4] can be expressed as the external choice between four different processes:

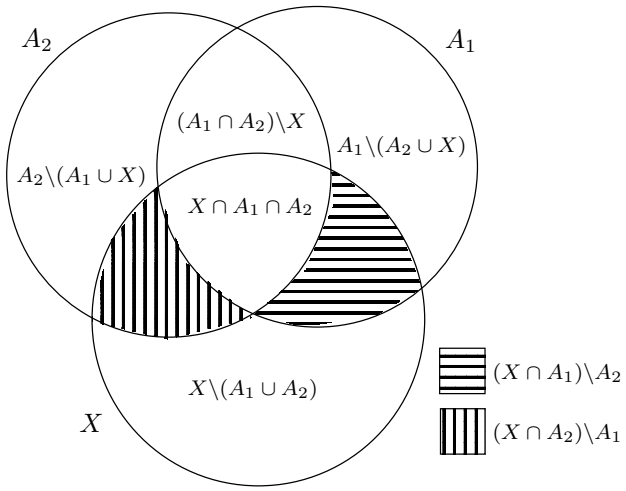

Figure 1. Venn diagram for sets of first actions.

$$
\begin{aligned}
R_1 \underset{X}{\vert\vert} R_2 = {} & (?x : X \cap A_1 \cap A_2 \rightarrow (R_1' \underset{X}{\vert\vert} R_2')) \\
& \Box\, (?x : (A_1 \cap A_2) \backslash X \rightarrow (R_1' \underset{X}{\vert\vert} R_2) \sqcap (R_1 \underset{X}{\vert\vert} R_2')) \\
& \Box\, (?x : A_1 \backslash (X \cup A_2) \rightarrow (R_1' \underset{X}{\vert\vert} R_2)) \\
& \Box\, (?x : A_2 \backslash (X \cup A_1) \rightarrow (R_1 \underset{X}{\vert\vert} R_2'))
\end{aligned}
$$

Note that the $\overset{||}{}_X$ -step law does *not* allow for any progress when the environment offers only some x in the shaded areas of Figure 1, i.e. when $x \in ((X \cap A_1)\backslash A_2) \cup ((X \cap A_2)\backslash A_1)$.

Suppose, however, that those restrictions were to be lifted. The resulting operator is then our optional (or partial) parallel operator, denoted by $\overset{\Uparrow}{}_X$. The optional parallel operator's step law needs to indicate what is to happen when the environment offers $x \in ((X \cap A_1)\backslash A_2)$ or $x \in ((X \cap A_2)\backslash A_1)$. Evidently, in the first case an interaction between the environment and R_1 should occur, and in the second case, an interaction between the environment and R_2. The $\overset{\Uparrow}{}_X$ -step law is therefore:

$$\begin{aligned} R_1 \overset{\Uparrow}{}_X R_2 = {} & (?x : X \cap A_1 \cap A_2 \to (R_1' \overset{\Uparrow}{}_X R_2')) \\ & \Box\ (?x : (A_1 \cap A_2)\backslash X \to (R_1' \overset{\Uparrow}{}_X R_2) \sqcap (R_1 \overset{\Uparrow}{}_X R_2')) \\ & \Box\ (?x : A_1\backslash(X \cup A_2) \to (R_1' \overset{\Uparrow}{}_X R_2)) \\ & \Box\ (?x : A_2\backslash(X \cup A_1) \to (R_1 \overset{\Uparrow}{}_X R_2')) \\ & \Box\ (?x : (X \cap A_1)\backslash A_2 \to (R_1' \overset{\Uparrow}{}_X R_2)) \\ & \Box\ (?x : (X \cap A_2)\backslash A_1 \to (R_1 \overset{\Uparrow}{}_X R_2')) \end{aligned}$$

This can be simplified to:

$$\begin{aligned} R_1 \overset{\Uparrow}{}_X R_2 = {} & (?x : X \cap A_1 \cap A_2 \to (R_1' \overset{\Uparrow}{}_X R_2')) \\ & \Box\ (?x : (A_1 \cap A_2)\backslash X \to (R_1' \overset{\Uparrow}{}_X R_2) \sqcap (R_1 \overset{\Uparrow}{}_X R_2')) \\ & \Box\ (?x : A_1\backslash A_2 \to (R_1' \overset{\Uparrow}{}_X R_2)) \\ & \Box\ (?x : A_2\backslash A_1 \to (R_1 \overset{\Uparrow}{}_X R_2')) \end{aligned}$$

Now, for $P = (x \to P')$, the process $P \overset{||}{}_X (R_1 \overset{\Uparrow}{}_X R_2)$ will lead to the desired behaviour: the process evolves into $P' \overset{||}{}_X (R_1' \overset{\Uparrow}{}_X R_2)$ or $P' \overset{||}{}_X (R_1 \overset{\Uparrow}{}_X R_2')$, depending on whether $x \in (X \cap A_1)\backslash A_2$ or $x \in (X \cap A_2)\backslash A_1$ respectively, as depicted in the Venn diagram of Figure 1. Of course, if $x \in (X \cap A_1 \cap A_2)$ then the process evolves into $P' \overset{||}{}_X (R_1' \overset{\Uparrow}{}_X R_2')$.

Of course, of itself the step law does not fully define the operator's semantics. This has been provided elsewhere [2], giving the denotational, trace, divergence and failures sematics, as well as the firing rules that specify the operational semantics.

It should be noted that the inspiration for this operator derives specifically from our earlier attempts to specify the present problem. The existing CSP operators were found to be deficient for our purposes. Once the semantics of the operator had been worked out, it became apparent that it can be usefully employed to specify a range of announcer/listener or reader/write-type problems. It will be seen that the operator neatly expresses the interaction between the fine-grained abstract processes that we have defined to solve the DFA minimisation problem.

5 Concurrent Specification

The principle concern in translating Algorithm 3.2 into a concurrent specification, is to translate its outer do-loop into a set of equivalent concurrent processes. To simplify the discussion, assume that P is the set of state pairs for which that loop would execute, i.e. $P = (Q \times Q)\backslash(G \cup H)$, where G and H are as initialised in Algorithm 3.2.

The overall system of interacting processes, called *Sys*, is conceived of as two processes that run in parallel with each other, and communicate via an alphabet, α. The two processes are called *Global* and *PairEquiv* respectively:

$$Sys = Global \underset{\alpha}{\|} PairEquiv \tag{2}$$

Global is a process that, for each state pair to be investigated, (p, q), receives information about the equivalence status of these states on a $from_{pq}$ channel (whose alphabet is therefore $\{from_{pq}.true,\ from_{pq}.false\}$) and announces the equivalence status of state-pairs on a to_{pq} channel (whose alphabet is therefore $\{to_{pq}.true,\ to_{pq}.false\}$). It is defined as follows:

$$\begin{aligned} Global &= |||_{(p,q)\in P}\ In_{pq} \\ In_{pq} &= from_{pq}?e \rightarrow Announce_{pq}(e) \\ Announce_{pq}(e) &= (to_{pq}!e \rightarrow Announce_{pq}(e) \\ &\quad \mid from_{pq}?e \rightarrow Announce_{pq}(e)) \end{aligned} \tag{3}$$

The *Global* process is thus the interleaving of In_{pq} processes—one for each (p, q) pair in the system. Each In_{pq} process receives the equivalence status of its associated (p, q) pair on the $from_{pq}$ channel and then repeatedly announces this status on the to_{pq} channel. Each In_{pq} engages in events from the alphabet: $\alpha(pq) = \{from_{pq}.true,\ from_{pq}.false,\ to_{pq}.true,\ to_{pq}.false\}$. The *Global* process communicates with the $PairEquiv$ process via the alphabet given by

$$\alpha = \bigcup_{(p,q)\in P} \alpha(pq)$$

The *PairEquiv* process not only passes on the equivalence status of each state pair to *Global*; it also acts as the "audience" to whom *Global* announces the equivalence status of pairs. It is the optional parallel composition, synchronising on events in α, of a set of processes called $Equiv_{pq}$, there being one such process for each (p, q) pair. *PairEquiv* is thus defined as:

$$PairEquiv = \underset{\alpha}{\Uparrow}_{(p,q)\in P}\ \ Equiv_{pq}(\emptyset, (|Q| - 2)\,\mathbf{max}\,0)$$

Note that, in general, each $Equiv_{pq}$ process has a parameter indicating the set S of pairs inspected by it to date, as well as the "recursion level", k, apparent in the sequential algorithm. In the initial $Equiv_{pq}$ processes as encountered in *PairEquiv*, $S = \emptyset$ and $k = (|Q| - 2)\,\mathbf{max}\,0)$).

Also note that the fact that these subprocesses of *PairEquiv*, namely $Equiv_{pq}$, mutually interact under optional parallelism with one another through α, while *PairEquiv* interacts with *Global* under generalised parallelism, also through α, means that whenever one *or more* of these subprocesses are ready to interact with *Global* on some arbitrary *to* or *from* channel, that interaction will take place as soon as the corresponding *Global* subprocess is ready to engage in it.

The generalised definition of $Equiv_{pq}$ is given by:

$$
\begin{aligned}
&Equiv_{pq}(S,k) = \\
&\qquad \text{if } (p = q) \text{ then } from_{pq}!true \rightarrow SKIP \\
&\qquad \text{else if } (k = 0) \text{ then } from_{pq}!(p \in F \equiv q \in F) \rightarrow SKIP \\
&\qquad\quad \text{else } (EqSet := \emptyset \\
&\qquad\qquad ;\ FanOut_{pq}(S,k) \\
&\qquad\qquad ;\ (eq :=\wedge_{e \in EqSet}\ e) \\
&\qquad\qquad ;\ (from_{pq}!eq \rightarrow SKIP))
\end{aligned}
$$

The mapping from the above process to its sequential counterpart is direct, except that the test of circularity in paths visited to date is shifted to just before the recursive invocation of *Equiv*, as shown below, in the expansion of the *FanOut* process:

$$
\begin{aligned}
&FanOut_{pq}(S,k) = |||_{a \in \Sigma_{pq}} \\
&\qquad (\text{if } (\{\delta(p,a), \delta(q,a)\} \notin S) \text{ then} \\
&\qquad\quad (\ Equiv_{\delta(p,a),\delta(q,a)}(S \cup \{(p,q)\}, k-1) \bigtriangleup \qquad (4) \\
&\qquad\quad (to_{\delta(p,a),\delta(q,a)}?eq_a \rightarrow (EqSet := EqSet \cup \{eq_a\})\ ) \\
&\qquad (\text{else } (EqSet := EqSet \cup \{true\})\ ) \qquad (5)
\end{aligned}
$$

The interrupt operator $\bigtriangleup$ in (4) requires justification. In the first place, it has been used there to ensure that the equivalence status of a pair is not needlessly sought by virtue of recursive calls to *Equiv* subprocesses, when that status had already been established and announced on the *from* channel to *Global* by some prior instance of an *Equiv* subprocess[1].

In the second place, the interrupt operartor's use here is justified, even though the CSP definition of the operator requires that if it is used in say $(P \bigtriangleup (a \rightarrow Q))$, then a may not be in the alphabet of P. This is in order to ensure that non-determinism cannot arise, such as in a situation where, say, $P = a \rightarrow R$. In such an case, the determination of the evolved process description after the occurrence of a would have to be non-deterministically chosen between Q and P. In the case of (4) above, such a non-deterministic choice will never be offered.

To realise that this is indeed the case, note the general form of the line, namely:

$Equiv_{pq}(S,k) \bigtriangleup (to_{pq}?eq \rightarrow \cdots)$

Now the only point at which an event on the channel to_{pq} can occur in $Equiv_{pq}(S,k)$ is when the chain of subprocesses spawned by $Equiv_{pq}(S,k)$ has cycled back to a new instance of itself. However, this is explicitly prevented by the condition of the if-statement preceding the process $Equiv_{pq}(S,k) \bigtriangleup (to_{pq}?eq \rightarrow \cdots)$. In such an case the process defined in (5) is executed. Thus, in the present context, the relaxation of the rule governing the use of the interrupt operator is justified—non-deterministic confusion cannot arise.

Note also that when the if-statement's condition is false—i.e. when a cycle has been detected on the fan-out branch from (p,q) for symbol a, then (as in the sequential

[1] Note that the semantics of the interrupt operator is such that when its first event occurs, all further activity of the initial process ceases, and the overall process behaves henceforth as the interrupting process. Furthermore, if the main process runs to completion, then the interrupting process plays no further role.

algorithm) this branch plays no role in the determination of (p, q)'s equivalence status. This is expressed by adding *true* into the *EqSet* set. It could equally well have been expressed by executing the *SKIP* process instead.

6 Conclusions

Just as in the case of the sequential algorithm, the foregoing specification has many optimisation possibilities. For example, the transitivity of the equivalence relation could be used to bring some of the *Equiv* processes more rapidly to an end. Also, as already mentioned, symmetry allows us to remove $Equiv_{qp}$ if $Equiv_{pq}$ is to be run.

This is the second well-known FA algorithm for which we have provided a concurrent CSP specification, the first one having been in [10]. The next phase of this ongoing project will be to experiment with implementations of these specifications. This will require reflection on ways in which the fine-grained processes that have been defined here can be coalesced with one another, and/or allocated to limited numbers of processors.

References

1. T. Agerwala and M. Gupta: *Systems research challenges: A scale-out perspective.* IBM Journal of Research and Development, 50(2/3) March/May 2006, pp. 173–180.
2. S. Gruner, D. G. Kourie, M. Roggenbach, T. Strauss, and B. W. Watson: *A new CSP operator for optional parallelism*, 2008, Submitted.
3. C. A. R. Hoare: *Communicating sequential processes.* Communications of the ACM, 26(1) 1983, pp. 100–106.
4. C. A. R. Hoare: *Communicating sequential processes (electronic version)*, 2004, `http://www.usingcsp.com/cspbook.pdf`.
5. J. F. JáJá and K. W. Ryu: *An efficient parallel algorithm for the single function coarsest partition problem*, in SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures, New York, NY, USA, 1993, ACM Press, pp. 230–239.
6. R. McDougall: *Extreme software scaling.* ACM Queue, 3(7) September 2005, pp. 36–46.
7. K. Olukoton and L. Hammond: *The future of microprocessors.* ACM Queue, 3(7) September 2005, pp. 26–29.
8. B. Ravikumar and X. Xiong: *A parallel algorithm for minimization of finite automata*, in IPPS: 10^{th} International Parallel Processing Symposium, IEEE Computer Society Press, 1996.
9. A. W. Roscoe: *The theory and practice of concurrency*, Prentice Hall, 1997.
10. T. Strauss, D. G. Kourie, and B. W. Watson: *A concurrent specification of Brzozowski's DFA construction algorithm*, in Proceedings of Prague Stringology Conference '06, 2006, pp. 90–99.
11. H. Sutter: *A fundamental turn toward concurrency in software.* Dr. Dobb's Journal, 30(3) March 2005, pp. 16–20,22.
12. H. Sutter and J. Larus: *Software and the concurrency revolution.* ACM Queue, 3(7) September 2005, pp. 54–62.
13. A. Tewari, U. Srivastava, and P. Gupta: *A parallel DFA minimization algorithm*, in Proceedings of HiPC2002, Lecture Notes in Computer Science 2552, Springer, 2002, pp. 34–40.
14. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Eindhoven University of Technology, September 1995.
15. B. W. Watson and J. Daciuk: *An efficient incremental DFA minimization algorithm.* Journal of Natural Language Engineering, 9(1) March 2003, pp. 49–64.