A Concurrent Specification of Brzozowski's DFA Construction Algorithm

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

FASTAR Research group
Department of Computer Science, University of Pretoria,
Pretoria, South Africa
{tstrauss, dkourie, bwatson}@cs.up.ac.za

Abstract. In this paper two concurrent versions of Brzozowski's deterministic finite automaton (DFA) construction algorithm are developed from first principles, the one being a slight refinement of the other. We rely on Hoare's CSP as our notation.

The specifications that are proposed of the Brzozowski algorithm are in terms of the concurrent composition of a number of top-level processes, each participating process itself composed of several other concurrent processes. After considering a number of alternatives, this particular overall architectural structure seemed like a natural and elegant mapping from the sequential algorithm's structure.

While we have carefully argued the reasons for constructing the concurrent versions as proposed in the paper, there are of course, a large range of alternative design choices that could be made. There might also be scope for a more fine-grained approach to updating sets or checking for similarity of regular expressions. At this stage, we have chosen to abstract away from these considerations, and leave their exploration to future research.

Keywords: automaton construction, concurrency, concurrent sequential processes, regular expressions

1 Introduction

This research is inspired by two contemporary trends. 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 [6], 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 [5, 7, 8].

It seems, though, that relatively little thought has been given in the finite automaton research community to developing concurrent versions of the various sequential algorithms that are widely in use. The only parallel algorithm that converts a regular expression into an automaton of which we are aware has been described by Ziadi and Champarnaud [9].

Here, two concurrent versions of Brzozowski's deterministic finite automaton (DFA) construction algorithm are developed from first principles, the one being a slight refinement of the other. This will be the theme of section 3. However, before developing the concurrent algorithm, we provide a brief overview of the sequential version in section 2. A brief reflection on this work is given in section 6.

2 Sequential Algorithm

Brzozowski's DFA construction algorithm [2] employs the notion of derivatives of regular expressions to construct a DFA. The algorithm takes a regular expression E as input and constructs an automaton which accepts the language represented by E.

The automaton is represented using the normal five-tuple notation $(D, \Sigma, \delta, S, F)$ where D is the set of states; Σ the alphabet; δ the transition relation mapping a state and an alphabet symbol to a state; and $S, F \subseteq D$ are the start and final states, respectively. $\mathcal{L}$ is an overloaded function giving the language of a finite automaton or a regular expression.

Since each regular expression in D is represented by a node in the automaton, we will sometimes refer to an element in a set as a regular expression, and at other times we will refer to it as a node.

The well-known sequential version of the algorithm is given in Dijkstra's guarded command language in figure 1. The notation assumes that the set operations ensure "uniqueness" of the elements at the level of similarity, i.e. $a \in A$ implies that there is no $b \in A$ such that a and b are similar regular expressions.

func $Brz(E, \Sigma) \rightarrow$
  $\delta, S, F := \varnothing, \{E\}, \varnothing;$
  $D, T := \varnothing, S;$

  do $(T \neq \varnothing) \rightarrow$
    let q be some state such that $q \in T$
    $D, T := D \cup \{q\}, T \setminus \{q\};$
    for $(i : \Sigma) \rightarrow$
      $d := \frac{d}{di} q;$
      if $d \notin (D \cup T) \rightarrow T := T \cup \{d\}$
      $[\!]$ $d \in (D \cup T) \rightarrow$ **skip**
      fi;
      $\delta(q, i) := d;$
    rof;
    if $\varepsilon \in \mathcal{L}(q) \rightarrow F := F \cup \{q\}$
    $[\!]$ $\varepsilon \notin \mathcal{L}(q) \rightarrow$ **skip**
    fi;
  od;
  return $(D, \Sigma, \delta, S, F);$

Figure 1. Brzozowski's DFA construction algorithm

Walking through this sequential algorithm, it will be seen that it relies on two sets: a set T containing the nodes (regular expressions) for which derivatives need to be calculated; and another set D containing the nodes for which derivatives have been found already.

The algorithm then works through all the nodes $q \in T$, finding derivatives with respect to all the alphabet symbols and depositing these nodes (regular expressions)

into T in those cases where no equivalent regular expression has already been deposited into $T \cup D$. Each node, q, dealt with in this fashion from T is then removed from T and added into D.

In each iteration of the inner `for` loop (i.e. for each alphabet symbol), the δ relation is updated to contain the mapping from node q to its derivative with respect to the relevant alphabet symbol.

Finally if q is nullable, it is added to the set of final states F.

In the forthcoming sections we present a concurrent specification of the algorithm in which we attempt to allow as much concurrency as possible.

3 Concurrent specification

We present here an approach to parallelising the algorithm. Of the many process algebras have been developed to concisely and accurately model concurrent systems, we have selected CSP [4, 3] 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 provide a brief introduction to the CSP operators that are used, and indicate some of the assumptions we make in regard to atomicity of operations.

3.1 Introductory Remarks

CSP is concerned with specifying a system of concurrent sequential processes (hence the CSP acronym) in terms of sequences of 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 main operators used in this article.

$a \rightarrow P$	event a then process Q
$a \rightarrow P \vert b \rightarrow Q$	a then P choice b then Q
$x : A \rightarrow P(x)$	choice of x from set A then $P(x)$
$P \parallel Q$	P in parallel with Q
	Synchronize on common events in the alphabet of P and Q
$b!e$	on channel b output event e
$b?x$	from channel b input to variable x
$P \not< C \not> Q$	if C then process P else process Q
$P; Q$	process P followed by process Q
$P \,\Box\, Q$	process P choice process Q

Table 1. Selected CSP Notation

Full details of the operator semantics and laws for their manipulation are available in [4, 3]. 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 $(x : A \rightarrow P(x)) \parallel (y : B \rightarrow Q(y))$ engages in some nondeterministically chosen event $z \in A \cap B$ and then behaves as the process $P(z) \parallel Q(z)$. However, if $A \cap B = \emptyset$ then deadlock results. A special case of such parallel synchronization is the process $(b!e \rightarrow P) \parallel (b?x \rightarrow Q(x))$. This should be viewed a process that engages in the event $b.e$ and thereafter behaves as the process $P \parallel Q(e)$.

3.2 Atomicity Assumptions

In deploying CSP, we have made the following assumptions relating to atomic execution.

Firstly, if an event maps to a function call, then that function is assumed to be a sequence of code in the original sequential algorithm which runs uninterruptedly to completion on some processor.

Furthermore, in the interest of conciseness and without loss of generality, it will sometimes be convenient to subsume certain assignment operations of the sequential program into the actual parameter list of a process invocation. For example, instead of specifying some recursive parameterised process $P(D)$ as $P(D) = \cdots (D := D \cup \{q\}); P(D)$, we will regard the specification $P(D) = \cdots P(D \cup \{q\})$ as equivalent. This means that operations that are needed to compute the actual parameters for a process invocation are regarded as taking place atomically—i.e. they cannot be interrupted by any other process's activity.

Similarly, where the CSP syntax for a conditional is used, as in $P \not\leftarrow C \not\rightarrow Q$, it will be assumed that the computation of the condition, C, takes place atomically and prior to the activation of any first event possible in the constituent processes, P and Q. This is specifically the case where similarity between regular expressions as implied in the Boolean expression $d \notin (D \cup T)$ has to be computed.

These instances of atomic activity are highlighted, not because they deviate from CSP syntax, but because they represent potential opportunities for a more fine-grained specification of the algorithm than what will be proposed below. However, deeper consideration of whether such a more fine-grained specification would be desirable or possible was deemed to be outside the scope of this present endeavour.

4 The *BRZ* process

The specification that is proposed of the Brzozowski algorithm is in terms of the concurrent composition of three top-level processes, each participating process itself composed of several other concurrent processes. After considering a number of alternatives, this particular overall architectural structure seemed like a natural and elegant mapping from the sequential algorithm's structure.

The first of these three processes is called *OUTER*. It corresponds to the actions of the outer loop of the sequential program. Another process called *DERIVE* caters for the computation of derivatives in the inner loop of the sequential version. Finally, an *UPDATE* process caters for the determination of which derived regular expressions should be used to update the "to do" set T, and also for updating the transition function, δ. The concurrent specification of the Brzozowski algorithm is thus:

$$BRZ(D, T) = OUTER(D, T) \parallel DERIVE \parallel UPDATE$$

Note the sets D and T are required as parameters for the *OUTER* process, because they are explicitly altered within this process. However, we will assume that these sets are globally available for read-only purposes within the other two processes, *DERIVE* and *UPDATE*. It will be convenient to regard the alphabet Σ as well as the sets F and δ as being a globally available to all sub-processes of the concurrent version of *BRZ*.

Furthermore, we assume that the first statement of the sequential algorithm—$\delta, S, F := \varnothing, \{E\}, \varnothing$;—takes place before the concurrent algorithm starts off. Given

a regular expression, E, the concurrent process $BRZ(\varnothing, \{E\})$ is equivalent to the sequential algorithm $BRZ(\{E\}, \Sigma)$.

In the subsequent sections these constituent processes of *BRZ* are explored and defined in greater detail. Figure 2 provides a graphical representation of the structure of *BRZ*. However, it also includes a refinement to this model that incorporates buffers for greater efficiency. This refinement is described in subsection 5.2.

4.1 The *OUTER* process

This process corresponds to the iterations of the outer loop in the sequential algorithm, in that it selects the next q to be processed, and caters for the updating of the two sets T and D.

The *OUTER* process has these two sets as parameters. As in the sequential case, D contains all the regular expressions for which derivatives have been found and will become nodes in the automaton. T is the set of regular expressions for which derivatives are still to be found.

The process is responsible for extracting an arbitrary node from T and then passing it on to the *DERIVE* process. It also updates the sets D, T, and F.

The process is defined in terms of a choice between two sub-processes. This is indicated by the CSP process choice operator, $\Box$. The first sub-process operand in the choice is initiated by engaging in an event that consists of selecting one of the regular expressions in T. The selected regular expression is called q. Thereafter, the *OUTER* process behaves as the parallel composition of two processes that take q as a parameter: *EXTRACT* and *FINAL*. This parallel composition has to run to completion before the *OUTER* process repeats, now with q added to D and removed from T.

Before considering the detail of the processes *EXTRACT* and *FINAL* that constitute the parallel composition, consider the second sub-process operand of the process choice operator in *OUTER*. It monitors a channel that is called *insert*, inputting a regular expression represented by the variable q from the channel whenever such an input becomes available. Thereafter *OUTER* repeats with q added to T. Again, we assume that this set union operation is atomic. This corresponds to the part in the sequential algorithm where the derivative d is added to T inside the inner loop.

$$\begin{aligned} OUTER(D,T) = {} & (q : T \rightarrow (EXTRACT(q) \parallel FINAL(q)); OUTER(D \cup \{q\}, T \setminus \{q\})) \\ & \Box \\ & (insert?q \rightarrow OUTER(D, T \cup \{q\})) \end{aligned}$$

Now consider the two processes within *OUTER* which are to execute as a parallel composition: *EXTRACT* and *FINAL*.

We begin with *EXTRACT*. Our task here is to express the fact that its parameter q should be broadcast to a set of processes that will independently compute the derivative of q, each with respect to a different symbol in the alphabet. To this end, *EXTRACT* is regarded as the parallel composition of a set of processes, designated $EXTRACT_i$ for each i in the alphabet Σ. Each $EXTRACT_i$ process passes its parameter, q, along its own channel, dIn_i, and then terminates successfully. The CSP specification to express the above is as follows:

$$EXTRACT(q) = \|_{i:\Sigma}\ EXTRACT_i(q) \quad \text{where}$$

$$EXTRACT_i(q) = (dIn_i!q \rightarrow SKIP)$$

As will be seen a little later, there is a $DERIVE_i$ process for each alphabet symbol i in Σ. Each of these processes will receive the outputted q on the associated dIn_i channel.

The *FINAL* process checks whether q is nullable and then adds q to the set of final states and then terminates successfully; otherwise *FINAL* terminates successfully without engaging in any action.

$$FINAL(q) = F := F \cup \{q\} \not< \varepsilon \in \mathcal{L}(q) \not> SKIP$$

Note that the set of events that take place in *EXTRACT* and *FINAL* are disjoint. The parallel composition of these two processes can therefore be described by the arbitrary interleaving of their respective event trace sets.

4.2 The *DERIVE* Process

The *DERIVE* process finds, in parallel, the derivatives of a regular expression with respect to all the symbols $i \in \Sigma$. The objective is to define *DERIVE* in such a way that each of its sub-processes can resume computing yet another derivative for a given alphabet symbol as soon as its task is complete, independently of the progress of its peer sub-processes. Here, a first order definition of *DERIVE* is given that does not fully meet this objective. This can be achieved by a simple refinement of the overall specification, as will be discussed later.

A sub-process that is designated $DERIVE_i$ receives input on channel dIn_i and outputs results of its computation to $dOut_i$. The channels have the same alphabet, namely, the set of all possible derivatives of regular expressions that can be constructed from Σ. Each $DERIVE_i$ process repeatedly accepts some arbitrary regular expression, and then emits the associated derivative with respect to i.

The parent *DERIVE* is therefore the parallel composition of all $DERIVE_i$ processes. Thus:

$$DERIVE = \|_{i:\Sigma} DERIVE_i \quad \text{where}$$

$$DERIVE_i = dIn_i?q \rightarrow dOut_i!(q, \frac{d}{di}q) \rightarrow DERIVE_i$$

Recall that data is put onto the dIn_i channel by process $EXTRACT_i$. Thus, in principle, the sub-processes $DERIVE_i$ and $EXTRACT_i$ can synchronise independently on events on channel dIn_i and run ahead of a pair of their peer sub-processes, say $DERIVE_j$ and $EXTRACT_j$. Unfortunately, the parent process of the $EXTRACT_i$ processes, namely *EXTRACT*, can only complete once *all* its constituent sub-processes have completed. And a fresh regular expression, q, can only be offered to $DERIVE_i$ via $EXTRACT_i$ once *EXTRACT* has completed, since only then can the recursive call to *OUTER* take place. This deficiency will be corrected in subsection 5.1, where a buffer will be placed on each channel.

At this stage, operations for updating δ and feeding the derivatives back to T are discussed.

4.3 The *UPDATE* Process

The *UPDATE* process is designed to receive a regular expression and its derivative with respect to i as a pair (q, d) from each $dOut_i$ channel. This is to happen independently of the state of readiness to receive some other regular expression and

derivative pair on an alternative channel, say $dOut_j$. In each case, the pair is passed on for updating δ and the derivative is considered for possible updating of T. The process is formed by the parallel composition of $DERIVE_i$ processes for each i in Σ. This can be expressed as follows:

$$UPDATE = \|_{i:\Sigma}\ UPDATE_i$$

After receiving the regular expression and derivative pair on the $dOut_i$ channel, each $UPDATE_i$ process behaves as the parallel composition of two sub-processes. One, called UPT_i, corresponds to the action of conditionally adding the derivative to T. The other, called UPD_i, corresponds to the action of unconditionally updating δ.

$$UPDATE_i = (dOut_i?(q,d) \rightarrow (\ UPT_i(d) \parallel\ UPD_i(q,d))\ );\ UPDATE_i$$

$UPT_i(d)$ establishes whether or not d is in $D \cup T$. If it is, then UPT_i simply terminates successfully. Otherwise, it outputs d on the *insert* channel, thus feeding d back to $OUTER$ where d is added to T. After this, the sub-process terminates successfully.

$$UPT_i(d) = insert!d \rightarrow SKIP \lneq d \notin (D \cup T) \gneq SKIP$$

UPD_i unconditionally updates δ and then terminates. The relation is updated by adding an entry into δ that represents a transition from q to d as a result of symbol i. Because each such update will always be with respect to a different (q,i) pair, there is no need to protect the data structure used to represent δ from write conflicts. How such concurrent access to the relevant data structure can actually be achieved is left as an implementation issue.

$$UPD_i(d) = \delta(q,i) := d$$

Note that $UPDATE_i$ starts again after the two sub-processes terminate successfully. Only then will a given $UPDATE_i$ sub-process be ready to input another (q,d) from its respective channel. Once more, there is scope modelling each of the $dOut_i$ channels as a buffer. This would ensure that any holdup in the execution of sub-processes UPT_i and UPD_i (in particular, the computation of the Boolean result of the condition in UPT_i) will not delay the supplier of data on the $dOut_i$ channel. However, in the interests of simplicity, this will not be modelled here. Instead, we illustrate below how the idea of buffering can be included between the $DERIVE_i$ and $EXTRACT_i$ processes, as previously suggested.

5 The *BRZBUFF* process

The $DERIVE_i$ and $EXTRACT_i$ processes are connected by synchronous channels. A process outputting a message onto a channel can only proceed when the receiving process inputs the message. This implies that $EXTRACT$ will only terminate once q has been read by all the $DERIVE_i$ processes. This will in turn prevent $OUTER$ from producing another q. So if, for example, there is a very slow $DERIVE_i$ process, all the others will have to wait for it to complete before continuing. This is clearly not desirable. Work for any processes should ideally be produced at least as fast as it can consume the work.

For the above reason it was decided to connect the $DERIVE_i$ and $EXTRACT_i$ processes through buffers. New q's can then be placed into the buffers without having to wait for all the $DERIVE_i$ processes to complete.

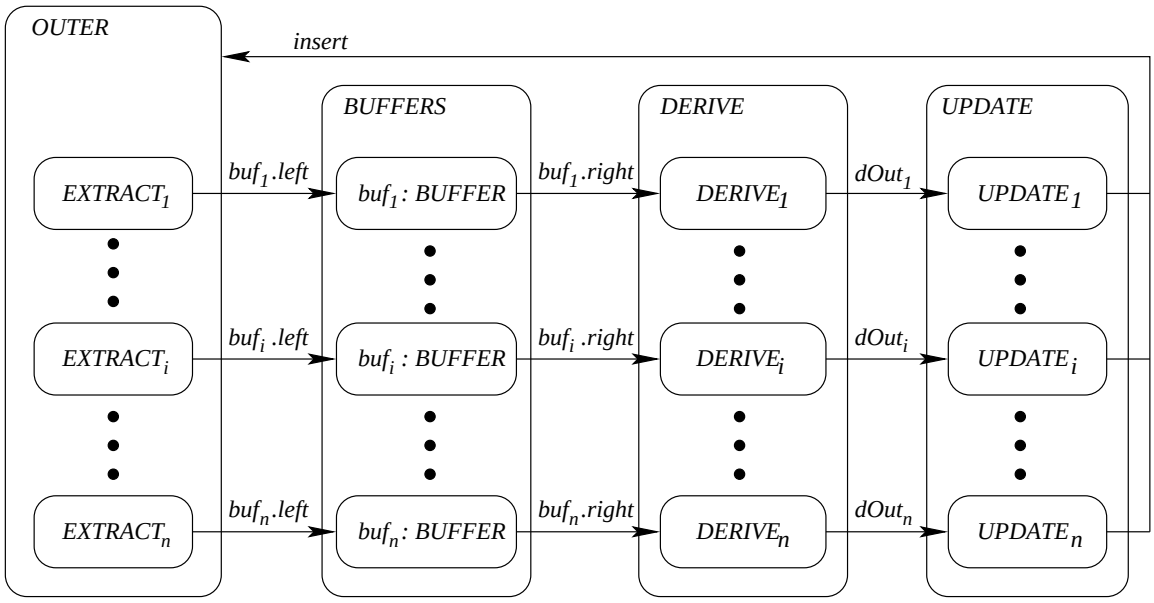

Figure 2. Graphical representation of the *BRZBUFF* process

5.1 The *BUFFERS* process

As suggested in [4], a buffer may be modelled using a process called *BUFFER*. It behaves like a queue—messages enter at the right and exit on the left in the same order that they arrived.

$$\begin{aligned} BUFFER &= P_{\langle\rangle} \quad \text{with} \\ P_{\langle\rangle} &= left?x \rightarrow P_{\langle x\rangle} \\ P_{\langle x\rangle^\frown s} &= (left?y \rightarrow P_{\langle x\rangle^\frown s^\frown\langle y\rangle} \\ &\qquad right!x \rightarrow P_s) \end{aligned}$$

Since each pair of processes, $EXTRACT_i$ and $DERIVE_i$, need to be connected through a buffer, multiple labelled copies of the *BUFFER* process are required. As a matter of convenience we will define a process called *BUFFERS* as the parallel composition of these *BUFFER* processes:

$$BUFFERS = \|_{i:\Sigma}(buf_i : BUFFER)$$

The only remaining step is to modify the respective definitions of the $DERIVE_i$ and $EXTRACT_i$ processes so that they interact through these buffers. This is necessary since the alphabet of each of these processes should contain the alphabet of the corresponding buffer process. Thus, each $EXTRACT_i(q)$ sub-process enters data on the left channel of its associated buffer, and may thus be defined as

$$EXTRACT_i(q) = (\underline{buf_i.left!q} \rightarrow SKIP)$$

Each corresponding $DERIVE_i$ sub-process inputs data from the right channel of its associated buffer. Its definition therefore changes to

$$DERIVE_i = \underline{buf_i.right?q} \rightarrow dOut_i!(q, \frac{d}{di}q) \rightarrow DERIVE_i$$

5.2 Putting everything together

A complete system that can be built from the preceding processes is designated $BRZBUFF(D, T)$, because it provides for the buffering just discussed. It is defined as follows:

$$BRZBUFF(D, T) = OUTER(D, T) \parallel BUFFERS \parallel DERIVE \parallel UPDATE$$

Thus, $BRZBUFF(\varnothing, \{E\})$ is a more efficient alternative to $BRZ(\varnothing, \{E\})$. It, too, is therefore a concurrent specification of the sequential algorithm given in figure 1.

Figure 2 depicts the major constituent processes of *BRZBUFF*. It should be clear from the diagram that each $EXTRACT_i$,$DERIVE_i$ pair is now connected via a buffer. The arrows in the diagram indicate the direction of information flow on the channels that connect the processes.

6 Conclusion

Although CSP has proved to be a convenient paradigm and notation for unravelling and articulating the concurrency inherent in the sequential algorithm, it has proven to be deficient in one respect: there does not seem to be a convenient mechanism for gracefully terminating the concurrent specification. As given above, the algorithm terminates when T is empty and further synchronisation is expected on the *input* channel. This means that the *OUTER* process does not terminate in a *SKIP*, but instead awaits further input on this channel, which never appears. Notwithstanding this deficiency, the problem can be easily overcome at the implementation level.

While we have carefully argued the reasons for constructing concurrent version as proposed above, there are of course, a large range of alternative design choices that could be made. These relate not only to overall architectural issues, but also to the level of more or less granularity in the concurrency, and whether the number of processors available should be explicitly taken into account.

Thus, we have already pointed out the scope for including even more buffering than we have. There might also be scope for a more fine-grained approach to updating sets or checking for similarity of regular expressions. At this stage, we have chosen to abstract away from these considerations, and leave their exploration to future research.

From an implementation point of view, it would be relatively easy to use a threaded language such as Java to do the implementation on a single processor platform. However, this does not appear to be particularly interesting, since the context switching required would undoubtedly render the concurrent version less efficient than its sequential counterpart. Instead, we are interested in implementing the concurrent version proposed above on one or more multiprocessor platforms. We expect this to be the immediate focus of our future research.

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] J. A. Brzozowski: *Derivatives of regular expressions.* Journal of the ACM, 11(4) 1964, pp. 481–494.

[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] R. McDougall: *Extreme software scaling.* ACM Queue, 3(7) September 2005, pp. 36–46.

[6] K. Olukoton and L. Hammond: *The future of microprocessors.* ACM Queue, 3(7) September 2005, pp. 26–29.

[7] H. Sutter: *A fundamental turn toward concurrency in software.* Dr. Dobb's Journal, 30(3) March 2005, pp. 16–20,22.

[8] H. Sutter and J. Larus: *Software and the concurrency revolution.* ACM Queue, 3(7) September 2005, pp. 54–62.

[9] D. Ziadi and J.-M. Champarnaud: *An optimal parallel algorithm to convert a regular expression into its Glushkov automaton.* Theoretical Computer Science, 215(1-2) February 1999, pp. 69–87.