RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

Testing Extended Regular Language Membership Incrementally by Rewriting

Grigore Roşu and Mahesh Viswanathan
Department of Computer Science
University of Illinois at Urbana-Champaign, USA

Abstract. In this paper we present lower bounds and rewriting algorithms for testing membership of a word in a regular language described by an extended regular expression. Motivated by intuitions from monitoring and testing, where the words to be tested (execution traces) are typically much longer than the size of the regular expressions (patterns or requirements), and by the fact that in many applications the traces are only available incrementally, on an event by event basis, our algorithms are based on an event-consumption idea: a just arrived event is "consumed" by the regular expression, i.e., the regular expression modifies itself into another expression discarding the event. We present an exponential space lower bound for monitoring extended regular expressions and argue that the presented rewriting-based algorithms, besides their simplicity and elegance, are practical and almost as good as one can hope. We experimented with and evaluated our algorithms in Maude.

1 Introduction

Regular expressions represent a compact and useful technique to specify patterns in strings. There are programming and/or scripting languages, such as Perl, which are mostly based on efficient implementations of pattern matching via regular expressions. Extended regular expressions (ERA), which add complementation ($\neg R$) to the usual union ($R_1 + R_2$), concatenation ($R_1 \cdot R_2$), and repetition ($R^\star$) operators, make the description of regular languages more convenient and more succinct. The membership problem for an extended regular expression R and a word $w = a_1 a_2 \ldots a_n$ is to decide whether w is in the regular language generated by R. The size of w is typically much larger than that of R.

Due to their convenience in specifying patterns, regular expressions, and implicitly the membership problem, have many applications and not only in computer science. For example, [14] suggests interesting applications in molecular biology. Monitoring and testing are other interesting application areas for regular expressions, because the execution of physical processes or computer programs can usually be abstracted by an external observer, or monitor, as a linear sequence of events. Since monitoring or testing of a process or program typically terminates after a period of time and a result of the monitoring/testing session is desired quickly, efficient implementations of the membership problem are of critical importance to these areas. Moreover, since monitoring sessions can be quite long, sometimes days or weeks, algorithms which do not need to store the execution trace or equivalent size information are typically preferred.

There has been some interest manifested recently in the software analysis community in using temporal logics in testing [9, 10]. The Temporal Rover tool

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

(TR) and its follower DB Rover [5] are already commercial; they are based on the idea of extending or instrumenting Java programs to enforce checking their execution trace against formulae expressed in temporal logics. The MaC tool [18] has developed its own language to express monitoring safety requirements, using an interval past time temporal logic at its core. In [21, 20] various algorithms to generate testing automata from temporal logic formulae are described. Java PathExplorer [7] is a runtime verification environment under current development at NASA Ames, whose logical monitoring part consists of checking execution traces against formulae expressed in both future time and past time temporal logics. [6, 8] present efficient algorithms for monitoring future time linear temporal logic formulae, while [11] gives a method to synthesize efficient monitors from past time temporal formulae. An interesting aspect of linear temporal logics in the context of monitoring/testing, is that they specify *patterns* for the execution traces of the monitored processes, which can also be specified by extended regular expressions of comparable or sometimes smaller size.

In this paper we focus on the membership problem for EREs. Previous work on the membership problem for regular expressions and their extensions [12, 19, 22, 17], have focussed on developing dynamic programming or automata based algorithms that run in time that is polynomial in both the size of the regular expression and the trace. These algorithms, however, suffer from a couple of drawbacks that make them unamenable as monitoring or testing algorithms. First, they are not incremental. They assume that the entire word is available when the algorithm is run. Second, the running time of these algorithms is at least quadratic in the size of the word. This is an unacceptably high overhead in monitoring and testing, because the word is usually enormous.

We, instead, investigate the membership problem in a model that is more appropriate for the context of monitoring and testing. More precisely, we assume that the ERE R to monitor is given apriori, but the letters a_1, a_2, ..., a_n forming the word w are received one by one, from the first (1) to the last (n). We often call the expression R a "requirement formula" and the letters in w "events". We also assume that w is large enough that one does not want to store it for future processing; therefore, each event has to be processed as it arrives. For that reason, we interchangeably call this problem the "monitoring" or the "incremental membership" problem. We give an exponential space lower bound by showing that any monitoring algorithm for EREs uses space that is $\Omega(2^{c\sqrt{m}})$ in the size m of the ERE, for some fixed constant c. Then, inspired by a related technique in [8] for future time linear temporal logic, we give a simple exponential space rewriting algorithm which solves the incremental membership problem in space $O(2^{m^2})$, thus giving an upper bound for the membership problem. In the end we give an improved version of the algorithm which we implemented and evaluated using Maude, which performs much better than the proved upper bound, thus opening the door for further interesting research in this direction.

Note that the simple-minded technique to first generate a nondeterministic (NFA) or a deterministic finite automaton (DFA) from the ERE and then to monitor against that NFA or DFA is not practical. This is because the size of

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

the NFA or DFA can be non-elementarily larger than the initial regular ERE, because negation involves an NFA-to-DFA translation, which implies an exponential blow-up; since negations can be nested, the size of such NFAs or DFAs could be highly exponential. Even if one would succeed in storing such an immense automaton, say a DFA, monitoring against it would still be highly exponential because a transition in a DFA requires time logarithmic in the total number of states (the next state needs to be at least read and each state label/name needs at least a logarithmic number of bits). ERE to (perhaps alternating) automata effective translations may well be possible, and we believe they are, but the simplistic ones are clearly too inconvenient to be considered.

2 Monitoring Extended Regular Expressions

In this section we define extended regular expressions (ERE) and languages formally, and give an exponential space lower bound for monitoring ERE.

2.1 Definitions

Extended regular expressions (ERE) define languages by inductively applying union ($+$), concatenation ($\cdot$), Kleene Closure ($\star$), intersection ($\cap$), and complementation ($\neg$). More precisely, for an alphabet Σ, an ERE over Σ is defined as follows, where $A \in \Sigma$: $R ::= \emptyset \mid \epsilon \mid A \mid R + R \mid R \cdot R \mid R^\star \mid R \cap R \mid \neg R$.

The language defined by an expression R, denoted by $\mathcal{L}(R)$, is defined inductively as $\mathcal{L}(\emptyset) = \emptyset$, $\mathcal{L}(\epsilon) = \{\epsilon\}$, $\mathcal{L}(A) = \{A\}$, $\mathcal{L}(R_1 + R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2)$, $\mathcal{L}(R_1 \cdot R_2) = \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(R_1) \text{ and } w_2 \in \mathcal{L}(R_2)\}$, $\mathcal{L}(R^\star) = (\mathcal{L}(R))^\star$, $\mathcal{L}(R_1 \cap R_2) = \mathcal{L}(R_1) \cap \mathcal{L}(R_2)$, $\mathcal{L}(\neg R) = \Sigma^\star \setminus \mathcal{L}(R)$. Given an ERE, as defined above using union, concatenation, Kleene Closure, intersection and complementation, one can translate it into an equivalent expression that does not have any intersection operation, by applying De Morgan's Laws. The translation only results in a linear blowup in size. Therefore, in the rest of the paper we do not consider expressions containing intersection. More precisely, we only consider EREs of the form $R ::= R + R \mid R \cdot R \mid R^\star \mid \neg R \mid A \mid \epsilon \mid \emptyset$.

2.2 Monitoring

In this subsection we will show that any monitoring algorithm for extended regular expressions must use space that is exponential in the size of the regular expression describing the correctness property. We will give an example of a language for which a lot of information needs to be remembered in order for it to determine if a trace satisfies the property.

The language that will be used in proving the lower bound was first present in [3] to show the power of alternation. Since then this example also has been used to prove lower bounds on LTL model checking [15, 16]. Consider the language

$$L_k = \{\sigma\#w\#\sigma'\$w \mid w \in \{0,1\}^k \text{ and } \sigma, \sigma' \in \{0,1,\#\}^\star\}.$$

We will first show that the above language can be described using an ERE of size $\Theta(k^2)$. We will then show that any monitoring algorithm must keep track of all strings over $\{0,1\}$ of length k that appear between $\#$ symbols before the $\$$ in the trace, in order for it to decide membership in L_k. This will give us a space lower bound of 2^k for monitoring algorithms.

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

Proposition 1. *There is an ERE R_k such that $L(R_k) = L_k$ and $|R_k| = \Theta(k^2)$.*
Proof. The ERE will be a conjunction of the following two facts.
(a) There is exactly one \$ symbol in the trace, and
(b) There is a # symbols after which there is a string of length k over $\{0,1\}$ before the next #, such that for every i, the ith symbol after the # is exactly the same as the ith symbol after the \$.

In other words, R_k is the following extended regular expression.

$$R_k = (\neg\$)^\star\$(\neg\$)^\star \bigcap$$
$$(0+1+\#)^\star\#[\bigcap_{i=0}^{k}[((0+1)^i 0(0+1)^{k-i-1}\#(0+1+\#)^\star\$(0+1)^i 0(0+1)^{k-i-1})$$
$$+((0+1)^i 1(0+1)^{k-i-1}\#(0+1+\#)^\star\$(0+1)^i 1(0+1)^{k-i-1})]]$$

Observe that $|R_k| = \Theta(k^2)$.

In order to prove the space lower bound, the following equivalence relation on strings over $(0+1+\#)^\star$ is useful. For a string $\sigma \in (0+1+\#)^\star$, define $S(\sigma) = \{w \in (0+1)^k \mid \exists \lambda_1, \lambda_2.\ \lambda_1 \# w \# \lambda_2 = \sigma\}$. We will say that $\sigma_1 \equiv_k \sigma_2$ iff $S(\sigma_1) = S(\sigma_2)$. Now observe that the number of equivalence classes of $\equiv_k$ is 2^{2^k}; this is because for any $S \subseteq (0+1)^k$, there is a σ such that $S(\sigma) = S$. We are now ready to prove the space lower bound.

Theorem 1. *Any ERE monitoring algorithm requires space $\Omega(2^{c\sqrt{m}})$, where m is the size of the input ERE and c is some fixed constant.*
Proof. Since $|R_k| = \theta(k^2)$ by Proposition 1, it follows that there is some constant c' such that $|R_k| \leq c'k^2$ for all large enough k. Let c be the constant $1/\sqrt{c'}$. We will prove this lower bound result by contradiction. Suppose A is an ERE monitoring algorithm that uses less that $2^{c\sqrt{m}}$ space for any EREs of large enough size m. We will look at the behavior of the algorithm A on inputs of the form R_k. So $m = |R_k| \leq c'k^2$, and A uses less than 2^k space. Since the number of equivalence classes of $\equiv_k$ is 2^{2^k}, by pigeon hole principle, there must be two strings $\sigma_1 \not\equiv_k \sigma_2$ such that the memory of $A(R_k)$ after reading $\sigma_1\$$ is the same as the memory after reading $\sigma_2\$$. In other words, $A(R_k)$ will give the same answer on all inputs of the form $\sigma_1\$w$ and $\sigma_2\$w$. Now since $\sigma_1 \not\equiv_k \sigma_2$, it follows that $(S(\sigma_1) \setminus S(\sigma_2) \cup (S(\sigma_2) \setminus S(\sigma_1)) \neq \emptyset$. Take $w \in (S(\sigma_1) \setminus S(\sigma_2) \cup (S(\sigma_2) \setminus S(\sigma_1))$. Then clearly, exactly one out of $\sigma_1\$w$ and $\sigma_2\$w$ is in L_k, and so $A(R_k)$ gives the wrong answer on one of these inputs. Therefore, A is not a correct.

3 An Event Consuming Rewriting Algorithm

In this section we introduce a rewriting-based monitoring procedure. It is based on an event consumption idea, in the sense that an extended regular expression R and an event a produce another extended regular expression, denoted $R\{a\}$, with the property that for any trace w, $aw \in R$ if and only if $w \in R\{a\}$. The ERE $R\{a\}$ is also known as a "derivative" of "residual" in the literature (see [2, 1], where several interesting properties of derivatives are also presented). The intuition here is that in order to incrementally test for membership of an incoming sequence of events to a given ERE, one can "process" the events as they are available, by modifying accordingly the monitoring requirement expression.

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

The rewriting systems in this paper are all considering that the operator $\_+\_$ is associative and commutative and that the operator $\_\cdot\_$ is associative. In other words, rewriting is performed modulo the equations:

$$(R_1 + R_2) + R_3 \equiv R_1 + (R_2 + R_3),$$
$$R_1 + R_2 \equiv R_2 + R_1,$$
$$(R_1 \cdot R_2) \cdot R_3 \equiv R_1 \cdot (R_2 \cdot R_3).$$

3.1 Rewriting Rules

We next consider an operation $\_\{\_\}$ which takes an extended regular expression and an event, and give seven rewriting rules which define its operational semantics recursively, on the structure of the regular expression:

$$(R_1 + R_2)\{a\} \to R_1\{a\} + R_2\{a\} \quad (1)$$
$$(R_1 \cdot R_2)\{a\} \to (R_1\{a\}) \cdot R_2 + \texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi} \quad (2)$$
$$(R^\star)\{a\} \to (R\{a\}) \cdot R^\star \quad (3)$$
$$(\neg R)\{a\} \to \neg(R\{a\}) \quad (4)$$
$$b\{a\} \to \texttt{if } (b = a) \texttt{ then } \epsilon \texttt{ else } \emptyset \texttt{ fi} \quad (5)$$
$$\epsilon\{a\} \to \emptyset \quad (6)$$
$$\emptyset\{a\} \to \emptyset \quad (7)$$

The right-hand sides of these rules use operations which we describe next. "`if` $(\_)$ `then` $\_$ `else` $\_$ `fi`" takes a boolean term and two EREs as arguments and has the expected meaning defined by two rewriting rules:

$$\texttt{if } (\mathit{true}) \texttt{ then } R_1 \texttt{ else } R_2 \texttt{ fi} \to R_1 \quad (8)$$
$$\texttt{if } (\mathit{false}) \texttt{ then } R_1 \texttt{ else } R_2 \texttt{ fi} \to R_2 \quad (9)$$

We assume a set of rewriting rules that properly evaluate boolean expressions. Boolean expressions include the constants *true* and *false*, as well as the usual connectors $\_ \wedge \_$, $\_ \vee \_$, and *not*. Testing for empty trace membership (which is used by (2)) can be efficiently implemented via the following rewriting rules:

$$\epsilon \in (R_1 + R_2) \to (\epsilon \in R_1) \vee (\epsilon \in R_2) \quad (10)$$
$$\epsilon \in (R_1 \cdot R_2) \to (\epsilon \in R_1) \wedge (\epsilon \in R_2) \quad (11)$$
$$\epsilon \in (R^\star) \to \mathit{true} \quad (12)$$
$$\epsilon \in (\neg R) \to \mathit{not}(\epsilon \in R) \quad (13)$$
$$\epsilon \in b \to \mathit{false} \quad (14)$$
$$\epsilon \in \epsilon \to \mathit{true} \quad (15)$$
$$\epsilon \in \emptyset \to \mathit{false} \quad (16)$$

The 16 rules defined above are natural and intuitive. Since the memory of our monitoring algorithm will consist of an ERE and since our main consideration here is memory, we pay special attention to the *size* of an ERE. The following three rules keep the size of the ERE generated by the other rules small. For that reason, we call them "simplifying rules". The latter may seem backwards at first sight. Its crucial role in maintaining EREs small will become clearer later:

$$R + \emptyset \to R \quad (17)$$
$$R + R \to R \quad (18)$$
$$R_1 \cdot R + R_2 \cdot R \to (R_1 + R_2) \cdot R \quad (19)$$

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

The sizes of the right-hand sides of these three rules are smaller (by at least 2) than their corresponding left-hand sides.

Let $\mathcal{R}$ denote the rewriting system defined above. Some notions and notations are needed before we can state the important results. Let $\equiv_{\mathcal{C}}$ denote the congruence relation generated by the set $\mathcal{C}$ containing the three equations just before Subsection 3.1 (associativity of $\_+\_$ and $\_\cdot\_$ and commutativity of $\_+\_$). Then the rewriting relation *modulo* $\mathcal{C}$ generated by the rules above, written $\rightarrow_{\mathcal{R}/\mathcal{C}}$, is the relation $\equiv_{\mathcal{C}};\rightarrow_{\mathcal{R}};\equiv_{\mathcal{C}}$, where semicolon denotes composition of binary relations and $\rightarrow_{\mathcal{R}}$ is the ordinary (non-AC) relation generated by $\mathcal{R}$. We say that $\mathcal{R}$ is *terminating modulo* $\mathcal{C}$ if and only if $\rightarrow_{\mathcal{R}/\mathcal{C}}$ is terminating, and that it is *ground Church-Rosser modulo* $\mathcal{C}$ if and only if $\leftrightarrow^{\star}_{\mathcal{R}\cup\mathcal{C}}$ is contained in $\rightarrow^{\star}_{\mathcal{R}/\mathcal{C}};\equiv_{\mathcal{C}};\leftrightarrow^{\star}_{\mathcal{R}/\mathcal{C}}$ on all ground terms (concrete EREs in our case). The typical technique to show termination modulo some equations is to define a weight function of terms, assigning a natural number to each term, and then show that this map is invariant with respect to equations and decreasing with respect to the rewriting rules.

Theorem 2. *$\mathcal{R}$ is terminating and ground Church-Rosser modulo $\mathcal{C}$; let $nf_{\mathcal{R}/\mathcal{C}}(R)$ be the normal form of R in $\mathcal{R}$ modulo $\mathcal{C}$. Furthermore, for a given extended regular expression R and a given event a, $\mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R\{a\})) = \{w \mid aw \in R\}$.*

Proof. Let γ be a function to natural numbers defined inductively as follows on terms of sort extended regular expression:

$$\begin{aligned}
&\gamma(R\{a\}) = (\gamma(R)+1)^2,\\
&\gamma(R_1 + R_2) = \gamma(R_1 \cdot R_2) = \gamma(R_1) + \gamma(R_2) + 1,\\
&\gamma(R^\star) = \gamma(\neg R) = \gamma(R) + 1,\\
&\gamma(b) = \gamma(\epsilon) = \gamma(\emptyset) = 1,
\end{aligned}$$

and on terms of sort bool:

$$\begin{aligned}
&\gamma(\epsilon \in R) = 2 \cdot \gamma(R),\\
&\gamma(B_1 \wedge B_2) = \gamma(B_1 \vee B_2) = \gamma(B_1) + \gamma(B_2) + 1,\\
&\gamma(not(B)) = \gamma(B) + 1,\\
&\gamma(true) = \gamma(false) = 1.
\end{aligned}$$

Let us now define a binary relation $\succ$ on extended regular expression terms as $R \succ R'$ if and only if $\gamma(R) > \gamma(R')$. It can easily be seen that $\succ$ is well-founded and that $\gamma(R) = \gamma(R')$ for each associativity or commutativity equation $R = R'$ in $\mathcal{C}$. We claim that $\succ$ includes the rewriting relation $\rightarrow_{\mathcal{R}/\mathcal{C}}$. It suffices to show that $\succ$ includes the relation $\rightarrow_{\mathcal{R}}$, which can be simply tested on each of the rewriting rules in $\mathcal{R}$ above. For example, rule (2) can be tested as follows:

$$\begin{array}{ll}
\gamma((R_1 \cdot R_2)\{a\}) > \gamma((R_1\{a\}) \cdot R_2 + \texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi}), & \text{iff}\\
(\gamma(R_1 \cdot R_2) + 1)^2 > \gamma((R_1\{a\}) \cdot R_2) + \gamma(\texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi}) + 1, & \text{iff}\\
(\gamma(R_1) + \gamma(R_2) + 2)^2 > (\gamma(R_1)+1)^2 + \gamma(R_2) + 2 \cdot \gamma(R_1) + (\gamma(R_2)+1)^2 + 3, & \text{iff}\\
2 \cdot \gamma(R_1) \cdot \gamma(R_2) + \gamma(R_2) > 1. &
\end{array}$$

For simplicity, assume that rule (5) is replaced by a finite set of rules $b\{a\} \rightarrow \emptyset$ for each different a, b in the alphabet and $a\{a\} \rightarrow \epsilon$ for each a. We therefore can conclude that $\mathcal{R}$ is terminating modulo $\mathcal{C}$. Due to space limitations, the Church-Rosser property of $\mathcal{R}$ modulo $\mathcal{C}$ will be shown elsewhere. However, since $\mathcal{R}$ is *not*

left-linear (see rules (18) and (19)), one *cannot* apply the classical critical pair completion procedure by Huet in [13].

We next show that for any extended regular expression R and any event a, $\mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R\{a\})) = \{w \mid aw \in R\}$. First notice that for any two extended regular expressions (without containing the operation $\_\{\_\}$) R and R', it is the case that $\mathcal{L}(R) = \mathcal{L}(R')$ whenever $R \rightarrow_{\mathcal{R}/\mathcal{C}} R'$; this is because the rules (17), (18) and (19) in R and all the equations in $\mathcal{C}$, the only which can be applied, are all valid properties of regular languages. In particular, $\mathcal{L}(R) = \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R)$ for any extended regular expression R. We can now start showing our main result inductively, on the structure of the extended regular expression:

$$\begin{aligned}
\mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}((R_1 + R_2)\{a\})) &= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R_1\{a\} + R_2\{a\})) \\
&= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R_1\{a\}) + nf_{\mathcal{R}/\mathcal{C}}(R_2\{a\})) \\
&= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R_1\{a\})) \cup \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R_2\{a\})) \\
&= \{w \mid aw \in \mathcal{L}(R_1)\} \cup \{w \mid aw \in \mathcal{L}(R_2)\} \\
&= \{w \mid aw \in \mathcal{L}(R_1 + R_2)\}.
\end{aligned}$$

Before we continue, note that for any ground or concrete ERE R, the normal form of $\epsilon \in R$ in $\mathcal{R}$ modulo $\mathcal{C}$ is either *true* or *false*. Moreover, it follows that $nf_{\mathcal{R}/\mathcal{C}}(\epsilon \in R) = \mathit{true}$ if and only if $\epsilon \in \mathcal{L}(R)$, which implies that $nf_{\mathcal{R}/\mathcal{C}}(\texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi})$ is either $nf_{\mathcal{R}/\mathcal{C}}(R_2\{a\})$ when $\epsilon \in \mathcal{L}(R_1)$ or $\emptyset$ when $\epsilon \notin \mathcal{L}(R_1)$. Then

$$\begin{aligned}
\mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}((R_1 \cdot R_2)\{a\})) &= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}((R_1\{a\}) \cdot R_2 + \\
&\qquad \texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi})) \\
&= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(nf_{\mathcal{R}/\mathcal{C}}(R_1\{a\}) \cdot R_2 + \\
&\qquad nf_{\mathcal{R}/\mathcal{C}}(\texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi}))) \\
&= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(nf_{\mathcal{R}/\mathcal{C}}(R_1\{a\}) \cdot R_2 + \\
&\qquad nf_{\mathcal{R}/\mathcal{C}}(R_2\{a\}))) \text{ when } \epsilon \in \mathcal{L}(R_1), \text{ or} \\
&\quad \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(nf_{\mathcal{R}/\mathcal{C}}(R_1\{a\}) \cdot R_2 + \emptyset)) \text{ when } \epsilon \notin \mathcal{L}(R_1) \\
&= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R_1\{a\})) \cdot \mathcal{L}(R_2) \cup \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R_2\{a\})) \\
&\quad \text{when } \epsilon \in \mathcal{L}(R_1), \text{ or} \\
&\quad \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R_1\{a\})) \cdot \mathcal{L}(R_2) \text{ when } \epsilon \notin \mathcal{L}(R_1) \\
&= \{w \mid aw \in \mathcal{L}(R_1)\} \cdot \mathcal{L}(R_2) \cup \{w \mid aw \in \mathcal{L}(R_2)\} \\
&\quad \text{when } \epsilon \in \mathcal{L}(R_1), \text{ or} \\
&\quad \{w \mid aw \in \mathcal{L}(R_1) \cdot \mathcal{L}(R_2)\} \text{ when } \epsilon \notin \mathcal{L}(R_1) \\
&= \{w \mid aw \in \mathcal{L}(R_1 \cdot R_2)\}.
\end{aligned}$$

Similarly, the inductive property follows for repetition and complement:

$$\begin{aligned}
\mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R^\star\{a\})) &= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R\{a\} \cdot R^\star)) \\
&= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R\{a\}) \cdot R^\star) \\
&= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R\{a\})) \cdot \mathcal{L}(R^\star) \\
&= \{w \mid aw \in R\} \cdot \mathcal{L}(R^\star) \\
&= \{w \mid aw \in R^\star\},
\end{aligned}$$

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

and

$$\begin{aligned}\mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}((\neg R)\{a\})) &= \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(\neg(R\{a\}))) \\ &= \mathcal{L}(\neg nf_{\mathcal{R}/\mathcal{C}}(R\{a\})) \\ &= \Sigma^\star \setminus \mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R\{a\})) \\ &= \Sigma^\star \setminus \{w \mid aw \in R\} \\ &= \{w \mid aw \notin R\} \\ &= \{w \mid aw \in \neg R\}.\end{aligned}$$

The remaining proofs, when R is a singleton, ϵ or $\emptyset$, are trivial. Thus we conclude that $\mathcal{L}(nf_{\mathcal{R}/\mathcal{C}}(R\{a\})) = \{w \mid aw \in R\}$ for any extended regular expression R and any event a.

From now on in the paper, we let $R\{a\}$ also (ambiguously) denote the term $nf_{\mathcal{R}/\mathcal{C}}(R\{a\})$, and consider that the rewrites in $\mathcal{R}$ modulo $\mathcal{C}$ are always applied automatically.

3.2 The Algorithm

We can now introduce our rewriting based algorithm for incrementally testing membership of words or traces to extended regular languages:

Algorithm $\mathcal{A}(R, a_1, a_2, ..., a_n)$
INPUT: An ERE R and events a_1, a_2, ..., a_n received incrementally
OUTPUT: *true* if and only if $a_1a_2...a_n \in \mathcal{L}(R)$; `false` otherwise
1. **let** R' **be** R
2. **let** i be 1
3. **while** $i \leq n$ **do**
4. **wait** until a_i is available
5. **let** R' **be** $nf_{\mathcal{R}/\mathcal{C}}(R'\{a_i\})$
6. **if** $R' = \emptyset$ **then return** *false*
7. **if** $R' = \neg(\emptyset)$ **then return** *true*
8. **let** i **be** $i + 1$
9. **return** $(\epsilon \in R')$; calculated using $\mathcal{R}$ (modulo $\mathcal{C}$ or not)

Therefore, a local ERE R' is updated after receiving each of the events a_i. If R' ever becomes empty (step 6) then, by Theorem 2, there is no way for the remaining events to make the whole trace into an accepting one, so the algorithm returns *fail* and the remaining events are not processed anymore. Similarly, if R' becomes the total language (step 7), then also by Theorem 2 it follows that any continuation will be accepted so the algorithm safely returns *true*. Step 8 finally tests whether the empty word is in R' after all the events have been processed, which, by Theorem 2 again, tells whether the sequence $a_1a_2...a_n$ is in the language of R.

3.3 Analysis

We will now show that the space and time requirements of our rewriting algorithm are not much worse than the lower bounds proved in the previous section.

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

Our rewriting algorithm keeps track of one extended regular expression which it modifies every time it receives a new event from the trace. We will prove that the size of this regular expression is bounded, no matter how many events are processed, and this will give us the desired bounds.

For an extended regular expression R, define the function size as follows:

$$\text{size}(R) = \max_{n,a_1,a_2,\ldots a_n} |R\{a_1\}\{a_2\}\cdots\{a_n\}|$$

So size(R) is the maximum size that R can grow to for any sequence of events.

Proposition 2. $\max_{|R|=m} \text{size}(R) \leq 2^{m^2}$.

Proof. Before presenting a proof of the bounds, we introduce some notation that will be useful in the proof. For a regular expression R, we will denote by $\overline{R\{a_1\}\{a_2\}\cdots\{a_n\}}$ the regular expression (actually its normal form in $\mathcal{R}$):

$$R\{a_1\}\{a_2\}\cdots\{a_n\} + R\{a_2\}\{a_3\}\cdots\{a_n\} + \cdots R\{a_n\}.$$

In addition, we define the following functions:

$$\begin{array}{l}
\overline{\text{size}}(R) = \max_{n,a_1,a_2,\ldots a_n} |\overline{R\{a_1\}\{a_2\}\cdots\{a_n\}}|, \\
\text{diff}(R) = \max_{n,a_1,a_2,\ldots a_n} |\{R\{a_i\}\{a_{i+1}\}\cdots\{a_n\} \mid 1 \leq i \leq n\}|, \\
\overline{\text{diff}}(R) = \max_{n,a_1,a_2\ldots a_n} |\{\overline{R\{a_i\}\{a_{i+1}\}\cdots\{a_n\}} \mid 1 \leq i \leq n\}|.
\end{array}$$

So $\overline{\text{size}}(R)$ measures the maximum size the expression $\overline{R}$ can grow to, $\text{diff}(R)$ measures the number of syntactically different terms in $\overline{R}$, and finally $\overline{\text{diff}}(R)$ is similar to $\text{diff}(R)$ but defined for $\overline{R}$

Using the above functions, we will be able to give bounds on the size of size, inductively. We first make some important observations regarding the expression $|R\{a_1\}\{a_2\}\cdots\{a_n\}|$ based on its form:

$$\begin{array}{rl}
|(R_1 + R_2)\{a_1\}\cdots\{a_n\}| &= |R_1\{a_1\}\cdots\{a_n\} + R_2\{a_1\}\cdots\{a_n\}| \\
&\leq |R_1\{a_1\}\cdots\{a_n\}| + |R_2\{a_1\}\cdots\{a_n\}| + 1 \\
|(R_1 \cdot R_2)\{a_1\}\cdots\{a_n\}| &\leq |(R_1\{a_1\}\cdots\{a_n\})\cdot R_2 + R_2\{a_n\} + \cdots + R_2\{a_1\}\cdots\{a_n\}| \\
&\leq |(R_1\{a_1\}\cdots\{a_n\})| + 1 + |R_2| + |\overline{R\{a_1\}\cdots\{a_n\}}| + 1 \\
|(R_1^\star)\{a_1\}\cdots\{a_n\}| &\leq |(\overline{R_1\{a_1\}\cdots\{a_n\}})\cdot R_1^\star| \\
&= |\overline{R_1\{a_1\}\cdots\{a_n\}}| + |R_1^\star| + 1 \\
|(\neg R_1)\{a_1\}\cdots\{a_n\}| &= |\neg(R_1\{a_1\}\cdots\{a_n\})| \\
&= |R_1\{a_1\}\cdots\{a_n\}| + 1
\end{array}$$

The only observation that needs some explanation is the one corresponding to $R_1^\star$. Observe that, $R_1^\star\{a_1\}\cdots\{a_n\}$ will get rewritten, in the worst case, as

$$(R_1\{a_1\}\cdots\{a_n\})\cdot R_1^\star + (R_1\{a_2\}\cdots\{a_n\})\cdot R_1^\star + \cdots (R_1\{a_n\})\cdot R_1^\star$$

which after simplification using the rule (19) will be $(\overline{R_1\{a_1\}\cdots\{a_n\}})\cdot R_1^\star$. Note that, in making the above observations, we make use of the fact that $\mathcal{R}$ is ground Church-Rosser modulo $\mathcal{C}$ (see Theorem 2).

Based on these observations, we can give an inductive bound on size:

$$\begin{array}{l}\mathrm{size}(R_1 + R_2) \leq \mathrm{size}(R_1) + \mathrm{size}(R_2) + 1,\\ \mathrm{size}(R_1 \cdot R_2) \leq \mathrm{size}(R_1) + |R_2| + \overline{\mathrm{size}}(R_2) + 2,\\ \mathrm{size}(R_1^\star) \leq \overline{\mathrm{size}}(R_1) + |R_1^\star| + 1,\\ \mathrm{size}(\neg R_1) \leq \mathrm{size}(R_1) + 1.\end{array}$$

We are now ready to give bounds on $\overline{\mathrm{size}}$. Observe that:

$$\begin{array}{l}\overline{\mathrm{size}}(R_1 + R_2) \leq \overline{\mathrm{size}}(R_1) + \overline{\mathrm{size}}(R_2) + 1,\\ \overline{\mathrm{size}}(R_1 \cdot R_2) \leq \overline{\mathrm{size}}(R_1) + |R_2| + \overline{\mathrm{size}}(R_2) + 2,\\ \overline{\mathrm{size}}(R_1^\star) \leq \overline{\mathrm{size}}(R_1) + |R_1^\star| + 1,\\ \overline{\mathrm{size}}(\neg R_1) \leq \mathrm{diff}(R_1) \cdot \mathrm{size}(R_1) + 2\mathrm{diff}(R_1).\end{array}$$

The reasons for the above inequalities is similar to those for size. The only case that needs explanation is the one for $\neg R_1$. Observe that $(\neg R_1)\{a_1\}\cdots\{a_n\} + (\neg R_1)\{a_2\}\cdots\{a_n\} + \cdots + (\neg R_1)\{a_n\}$ is the same as $\neg(R_1\{a_1\}\cdots\{a_n\}) + \cdots + \neg(R_1\{a_n\})$. So based on how many of the terms $R_1\{a_i\}\cdots\{a_n\}$ are different, we can bound $\overline{\mathrm{size}}(\neg R_1)$.

Finally, we give the bounds on the function diff and $\overline{\mathrm{diff}}$ based on a similar reasoning:

$$\begin{array}{l}\mathrm{diff}(R_1 + R_2) \leq \mathrm{diff}(R_1) \cdot \mathrm{diff}(R_2),\\ \mathrm{diff}(R_1 \cdot R_2) \leq \mathrm{diff}(R_1) \cdot \overline{\mathrm{diff}}(R_2),\\ \mathrm{diff}(R_1^\star) \leq \overline{\mathrm{diff}}(R_1),\\ \mathrm{diff}(\neg R_1) \leq \mathrm{diff}(R_1).\end{array}$$

To complete the analysis, observe that $\overline{\mathrm{diff}}(R) \leq \mathrm{diff}(R)$.

If we take $(\max_{|R|=m} \overline{\mathrm{diff}}(R))$ and $(\max_{|R|=m} \mathrm{diff}(R))$ to be bounded by 2^m, and $(\max_{|R|=m} \overline{\mathrm{size}}(R))$ and $(\max_{|R|=m} \mathrm{size}(R))$ to be bounded by 2^{m^2}, then we see that all of the inequalities are satisfied. Hence the proposition follows.

Theorem 3. *The monitoring algorithm based on rewriting uses space $O(2^{2m^2})$ and time $O(n \cdot 2^{2m^2})$; time is measured in number or rewriting steps.*

Proof. The space needed by the algorithm consists of the space needed to store the evolving ERE. By the proposition above, we know that, after simplification, such an ERE will never be larger than $O(2^{m^2})$, where m is the size of the initial ERE. However, before simplification, the stored ERE first suffers an increase in size. We claim that, regardless of the order in which rewrite rules are applied, the size of the intermediate term obtained by deriving a given ERE of size M will never grow larger than M^2. This is indeed true, because if one analyzes the rewriting rules which can increase the size of the term, namely rules (1)–(7) and (10)–(11), then one can see that the worst case scenario is given by a recurrence $S(M_1 + M_2 + 1) \leq S(M_1) + S(M_2) + M_1 + M_2 + c$, where c is some (small) constant; this recurrence implies $S(M) = O(M^2)$. Therefore, the space needed by our rewriting algorithm is $O(2^{2m^2})$.

The number of rewrites needed to process one event is also $O(2^{2m^2})$. Note first that the number of rewrites for a test $\epsilon \in R$ is $|R|$. Then one can easily give

a recurrence for the number of rewrites to push an event to leaves; for example, in the case of concatenation this is $N((R_1R_2)\{a\}) \leq N(R_1) + |R_1| + N(R_2) + 1$. Therefore, there are $O(2^{2m^2})$ applications of rules (1)–(16). Since each of the remaining rules, the simplifying ones, decrease the size of the term by 2 and the maximum size of the term is $O(2^{2m^2})$, it follows that the total number of rewrites needed to process an event is indeed $O(2^{2m^2})$.

The above results can be improved if one considers only regular expressions, instead of extended regular expressions. Applying the same rewrite algorithm to expressions that do not have negations, we can use the very same analysis to observe that the rewrite algorithm uses space $O(m^2)$ and running time $O(n \cdot m^2)$

Theorem 4. *The monitoring algorithm based on rewriting, when applied to expressions not containing any negation, use space $O(m^2)$ and time $O(n \cdot m^2)$.*

4 Implementation, Evaluation and Conclusion

We have implemented in Maude [4] several improved versions of the rewriting-based algorithm in Section 3. In this section we present an implementation which worked best on our test suits. Space/time analysis seems hard to do rigorously and is not given for this implementation, but the given experimental data suggest that the $O(2^{m^2})$ space upper bound proved in Subsection 3.3 is more of a theoretical importance than practical. We hope to calculate the exact worst-case complexity of the next rewriting procedure soon, but for now are happy to present it as a procedure for monitoring extended regular expressions which performs very well in practice. The usual operations on extended regular expressions can be defined in a functional module (`fmod ... endfm`) as follows:

```
fmod ERE is
  sorts Event Ere .
  subsort Event < Ere .
  op _+_ : Ere Ere -> Ere [assoc comm prec 60] .
  op __  : Ere Ere -> Ere [assoc prec 50] .
  ops (_*) (~_)  : Ere -> Ere .
  ops epsilon empty : -> Ere .
endfm
```

Precedences were given to some operators to avoid writing parentheses: the lower the precedence the tighter the binding.

10 rules for ϵ-membership and for simplifying extended regular expressions were given in Section 3 (rules (10)-(19)). These rules were shown to keep the size of any evolving extended regular expression lower than $O(2^{m^2})$, where m is its initial size. Driven by practical experiments, we have decided to define a partial ERE inclusion operator, called `_in_`, using 22 rewriting rules (some of them conditional) which correctly extends the needed (total) ϵ-membership in Section 3. Together with other 10 simplifying rules, ERE inclusion is defined in the following module:

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

```
fmod SYMPLIFY-ERE is including ERE .
  vars R R' R1 R2 R1' R2' : Ere .  vars A B : Event .
  eq empty R = empty .               eq R empty = empty .
  eq epsilon R = R .                 eq R epsilon = R .
  eq ~ ~ R = R .                     eq R * * = R * .
  eq epsilon * = epsilon .           eq empty * = empty .
 ceq R1 + R2 = R2 if R1 in R2 .      eq R1 R + R2 R = (R1 + R2) R .

  op _in_ : Ere Ere -> Bool .
  eq empty in R = true .             eq epsilon in A = false .
  eq A in B = (A == B) .             eq R in R = true .
  eq epsilon in (R1 + R2) = epsilon in R1 or epsilon in R2 .
  eq A in (R1 + R2) = A in R1 or A in R2 .
 ceq R in (R1 + R2) = true if R in R1 .
  eq (R1 + R2) in R = R1 in R and R2 in R .
  eq epsilon in (R1 R2) = epsilon in R1 and epsilon in R2 .
  eq A in (R1 R2) =
       A in R1 and epsilon in R2 or A in R2 and epsilon in R1 .
 ceq (R1 R2) in (R1' R2') = true if (R1 in R1') /\ (R2 in R2') .
  eq epsilon in (R *) = true .
 ceq R1 in (R *) = true if R1 in R .
 ceq (R1 R2) in (R *) = true if (R1 in (R *)) /\ (R2 in (R *)) .
  eq R in (~ empty) = true .
  eq R in (~ epsilon) = not (epsilon in R) .
  eq R in (~ A) = not (A in R) .
  eq epsilon in (~ R) = not (epsilon in R) .
  eq A in (~ R) = not(A in R) .
  eq (~ R) in (~ R') = R' in R .
  eq R in empty = R == empty .
  eq R in epsilon = R == empty or R == epsilon .
endfm
```

The module above therefore adds 32 equational constraints to the EREs defined syntactically in the module `ERE` (included with the Maude keyword `including`). Maude executes these equations as (conditional) rewrite rules. The major simplifying rule in `SIMPLIFY-ERE` is the 5th on the left column, which properly generalizes rule (18) in Section 3; this was the rule motivating the definition of the ERE partial inclusion.

We can now define the event consuming operator, _{_}, together with its associated seven rules (1)-(7) from Section 3:

```
fmod CONSUME-EVENT is protecting SYMPLIFY-ERE .
  vars R1 R2 R : Ere .  vars A B C : Event .
  op _{_} : Ere Event -> Ere [prec 45] .
  eq (R1 + R2){A} = R1{A} + R2{A} .
  eq (R1 R2){A} =
       R1{A} R2 + if (epsilon in R1) then R2{A} else empty fi .
  eq (R *){A} = R{A} (R *) .
  eq (~ R){A} = ~ (R{A}) .
```

```
  eq B{A} = if B == A then epsilon else empty fi .
  eq epsilon{A} = empty .
  eq empty{A} = empty .
endfm
```

The conditional operator `if_then_else_fi`, whose semantics was given by the rules (8)-(9) in Section 3, is part of the builtin `BOOL` module in Maude.

One can now use the rewriting procedure above by either launching Maude `reduce` commands directly, such as:

```
red (A(A + B)*)* {A} .
red ((A + B)((C + A)* (A B *)*)*)* {A} .
red ((A + B)((C + A)* (A B *)*)*)* {B} .
red ((A + B)((C + A)* (A B *)*)*)* {C} .
```

which give the following expected answers,

```
==========================================
reduce in CONSUME-EVENT : (A (A + B) *) *{A} .
rewrites: 14 in 0ms cpu (0ms real) (~ rewrites/second)
result Ere: (A + B) * (A (A + B) *) *
==========================================
reduce in CONSUME-EVENT : ((A + B) ((A + C) * (A B *) *) *) *{A} .
rewrites: 32 in 0ms cpu (0ms real) (~ rewrites/second)
result Ere: ((A + C) * (A B *) *) * ((A + B) ((A + C) * (A B *)*)*)*
==========================================
reduce in CONSUME-EVENT : ((A + B) ((A + C) * (A B *) *) *) *{B} .
rewrites: 32 in 0ms cpu (0ms real) (~ rewrites/second)
result Ere: ((A + C) * (A B *) *) * ((A + B) ((A + C) * (A B *)*)*)*
==========================================
reduce in CONSUME-EVENT : ((A + B) ((A + C) * (A B *) *) *) *{C} .
rewrites: 31 in 0ms cpu (0ms real) (~ rewrites/second)
result Ere: empty
```

or by calling it from a different place (procedure, thread, process) where the algorithm in Subsection 3.2 is implemented – it is worth mentioning that this algorithm can also be implemented directly in Maude, using its *loop mode* feature [4] which is specially designed to process events interactively.

We have tested the event consuming procedure above on several extended regular expressions and several sequences of events, and the results were quite encouraging. We were not able to notice any measurable running time on meaningful formulae that one would want to enforce in real software monitoring applications. In order to do proper worst-case measurements, we have implemented (also by rewriting in Maude) another procedure which takes as input a natural number m and does the following:

1. Generates all extended regular expressions of size m over 0 and 1;
2. For each such expression R, it calculates the number size(R) (see Subsection 3.3) by exhaustively generating the *set* of all the extended regular expressions $R\{a_1\}\{a_2\}\cdots\{a_n\}$ for all n and $a_1, a_2, ..., a_n \in \{0, 1\}$; by Proposition 2, this set is finite;

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

3. It returns the largest of size(R) for all R above.

This algorithm is obviously very inefficient[1]. We were only able to run it for all $m \leq 12$ in less than 24 hours, generating the following table:

m	1	2	3	4	5	6	7	8	9	10	11	12
$\max_{\|R\|=m}(\text{size}(R))$	1	2	6	8	18	24	39	51	57	77	92	108

Since the space requirements of our rewriting monitoring procedure is given by the size of the current formula, the table above gives us a measure of the space needed in the worst case by our rewriting algorithm. It shows for example that an extended regular expression of size 12, in the worst possible case grows to size 108, which is of course infinitely better than the upper bound that we were able to prove for the simplified algorithm, namely $2^{12^2} = 22,300,745,198,530,623,141,535,718,272,648,361,505,980,416$. This tells us that there is plenty of room for further research in finding better rewriting based algorithms and better upper bounds for space requirements than the ones we were able to find in Section 3. The improved rewriting procedure presented in this section can be such a significantly better membership algorithm, but proving it seems to be hard.

It is worth mentioning that, even if one removes the auxiliary rewriting rules from the module above and keeps only the 19 rules presented in the previous section, the size of the evolving ERE still stays smaller that 2^m. This stimulates us to conclude with the following:

Conjecture. *The rewriting-based algorithm presented in Section 3 runs in space $O(2^m)$ and time $O(n2^m)$, where m is the size of the ERE and n is the size of the event trace. Moreover, these are the lower bounds for the membership problem.*

References

1. V.M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Journal of Theoretical Computer Science*, 155(2):291–319, 1996.
2. V.M. Antimirov and P.D. Mosses. Rewriting extended regular expressions. *Journal of Theoretical Computer Science*, 143(1):51–72, 1995.
3. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.
4. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. SRI International, January 1999, http://maude.csl.sri.com.
5. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *Lecture Notes in Computer Science*, pages 323–330. Springer, 2000.
6. D. Giannakopoulou and K. Havelund. Automata-Based Verification of Temporal Properties on Running Programs. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 412–416. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.

[1] We are, however, happy to provide it on request.

RTA'03, Lecture Notes in Computer Science (LNCS) 2706, pp 499-514. 2003

7. K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In *Proceedings of Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001.
8. K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings, International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. Institute of Electrical and Electronics Engineers, 2001. Coronado Island, California.
9. K. Havelund and G. Roşu. *Runtime Verification 2001*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2001. Proceedings of a *Computer Aided Verification (CAV'01)* satellite workshop.
10. K. Havelund and G. Roşu. *Runtime Verification 2002*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002. Proceedings of a *Computer Aided Verification (CAV'02)* satellite workshop.
11. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002.
12. S. Hirst. A new algorithm solving membership of extended regular expressions. Technical report, The University of Sydney, 1989.
13. G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
14. J.R. Knight and E.W. Myers. Super-pattern matching. *Algorithmica*, 13(1/2):211–243, 1995.
15. O. Kupferman and M. Y. Vardi. Freedom, Weakness, and Determinism: From linear-time to branching-time. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 81–92, 1998.
16. O. Kupferman and M. Y. Vardi. Model Checking of Safety Properties. In *Proceedings of the Conference on Computer-Aided Verification*, 1999.
17. O. Kupferman and S. Zuhovitzky. An Improved Algorithm for the Membership Problem for Extended Regular Expressions. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, 2002.
18. I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime Assurance Based on Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.
19. G. Myers. A four russians algorithm for regular expression pattern matching. *Journal of the ACM*, 39(4):430–448, 1992.
20. T. O'Malley, D. Richardson, and L. Dillon. Efficient Specification-Based Oracles for Critical Systems. In *In Proceedings of the California Software Symposium*, 1996.
21. D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *Proceedings of the Fourteenth International Conference on Software Engineering, Melbourne, Australia*, pages 105–118, 1992.
22. H. Yamamoto. An automata-based recognition algorithm for semi-extended regular expressions. In *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, pages 699–708, 2000.