Electronic Notes in Theoretical Computer Science 89 No. 2 (2003)
URL: `http://www.elsevier.nl/locate/entcs/volume89.html` 20 pages

Generating Optimal Monitors for Extended Regular Expressions

Koushik Sen [1,3]

Department of Computer Science,
University of Illinois at Urbana Champaign,
USA.

Grigore Roşu [2,4]

Department of Computer Science,
University of Illinois at Urbana Champaign,
USA.

Abstract

Ordinary software engineers and programmers can easily understand regular patterns, as shown by the immense interest in and the success of scripting languages like Perl, based essentially on regular expression pattern matching. We believe that regular expressions provide an elegant and powerful specification language also for *monitoring requirements*, because an execution trace of a program is in fact a string of states. Extended regular expressions (EREs) add complementation to regular expressions, which brings additional benefits by allowing one to specify patterns that must *not* occur during an execution. Complementation gives one the power to express patterns on strings more compactly. In this paper we present a technique to generate *optimal monitors* from EREs. Our monitors are deterministic finite automata (DFA) and our novel contribution is to generate them using a modern coalgebraic technique called *coinduction*. Based on experiments with our implementation, which can be publicly tested and used over the web, we believe that our technique is more efficient than the simplistic method based on complementation of automata which can quickly lead to a highly-exponential state explosion.

---

[1] Supported in part by the Defense Advanced Research Projects Agency (the DARPA IPTO TASK Program, contract number F30602-00-2-0586 and the DARPA IXO NEST Program, contract number F33615-01-C-1907) and the ONR Grant N00014-02-1-0715.

[2] Supported in part by the joint NSF/NASA grant CCR-0234524.

[3] Email: `ksen@cs.uiuc.edu`

[4] Email: `grosu@cs.uiuc.edu`

1 Introduction

Regular expressions can express patterns in strings in a compact way. They proved very useful in practice; many programming/scripting languages like Perl, Python, Tcl/Tk support regular expressions as core features. Because of their power to express a rich class of patterns, regular expressions, are used not only in computer science but also in various other fields, such as molecular biology [18]. All these applications boast of very efficient implementation of regular expression pattern matching and/or membership algorithms. Moreover, it has been found that compactness of regular expressions can be increased non-elementarily by adding complementation ($\neg R$) to the usual union ($R_1 + R_2$), concatenation ($R_1 \cdot R_2$), and repetition (R^*) operators of regular expressions. These are known as *extended regular expressions* (EREs) and they proved very intuitive and succinct in expressing regular patterns.

Recent trends have shown that the software analysis community is inclining towards scalable techniques for software verification. Works in [12] merged temporal logics with testing, thereby getting the benefits of both worlds. The Temporal Rover tool (TR) and its follower DB Rover [5] are already commercial. In these tools the Java code is instrumented automatically so that it can check the satisfaction of temporal logic properties at runtime. The MaC tool [17,22] has been developed to monitor safety properties in interval past time temporal logics. In [24,25], various algorithms to generate testing automata from temporal logic formulae, are described. Java PathExplorer [10] is a runtime verification environment currently under development at NASA Ames. The Java MultiPathExplorer tool [29] proposes a technique to monitor all equivalent traces that can be extracted from a given execution, thus increasing the coverage of monitoring. [7,11] present efficient algorithms for monitoring future time temporal logic formulae, while [13] gives a technique to synthesize efficient monitors from past time temporal formulae. [27] uses rewriting to perform runtime monitoring of EREs.

An interesting aspect of EREs is that they can express safety properties compactly, like those encountered in testing and monitoring. By generating automata from logical formulae, several of the works above show that the safety properties expressed by different variants of temporal logics are subclasses of regular languages. The converse is *not* true, because there are regular patterns which cannot be expressed using temporal logics, most notoriously those related to counting; e.g., the regular expression $(0 \cdot (0 + 1))^*$ saying that every other letter is 0 does not admit an equivalent temporal logic formula. Additionally, EREs tend to be often very natural and intuitive in expressing requirements. For example, let us try to capture the safety property "it should not be the case that in any trace of a traffic light we see green and then immediately red at any point". The natural and intuitive way to express it in ERE is $\neg((\neg\emptyset) \cdot \texttt{green} \cdot \texttt{red} \cdot (\neg\emptyset))$, where $\emptyset$ is the empty ERE (no words), so $\neg\emptyset$ means "anything".

Previous approaches to ERE membership testing [14,23,31,21,16] have focussed on developing techniques that are polynomial in both the size of the word and the size of the formulae. The best known result in these approaches is described in [21] where they can check if a word satisfies an ERE in time $O(m \cdot n^2)$ and space $O(m \cdot m + k \cdot n^2)$, where m is the size of the ERE, n is the length of the word, and k is the number of negation/intersection operators. These algorithms, unfortunately, cannot be used for the purpose of monitoring. This is because they are not incremental. They assume the entire word is available before their execution. Additionally, their running time and space requirements are quadratic in the size of the trace. This is unacceptable when one has a long trace of events and wants to monitor a small ERE, as it is typically the case. This problem is removed in [27] where traces are checked against EREs through incremental rewriting. At present, we do not know if the technique in [27] is optimal or not.

A simple, straightforward, and practical approach is to generate optimal *deterministic finite automata* (DFA) from EREs [15]. This process involves the conversion of each negative sub-component of the ERE to a non-deterministic finite automaton (NFA), determinization of the NFA into a DFA, complementation of the DFA, and then its minimization. The algorithm runs in a bottom-up fashion starting from the innermost negative ERE sub components. This method, although generates the minimal automata, is too complex and cumbersome in practice. Its space requirements 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.

Our approach is to generate the minimal DFA from an ERE using coinductive techniques. In this paper, the DFA thus generated is called the *optimal monitor* for the given ERE. A number of related approaches for simple regular expressions can be found in [30]. Our algorithm extends these approaches for simple regular expressions to EREs; further, we make the algorithm efficient by intelligent book keeping. The complexity of our algorithm seems to be hard to evaluate, because it depends on the size of the minimal DFA associated to an ERE and we are not aware of any lower bound results in this direction. However, experiments are very encouraging. Our implementation, which is available for evaluation on the internet via a CGI server reachable from `http://fsl.cs.uiuc.edu/rv/`, rarely took longer than one second to generate a DFA, and it took only 18 minutes to generate the minimal 107 state DFA for the ERE in Example 5.3 which was used to show the exponential space lower bound of ERE monitoring in [27].

In a nutshell, in our approach we use the concept of derivatives of an ERE, as described in Subsection 2.2. For a given ERE one generates all possible derivatives of the ERE for all possible sequences of events. The size of this set of derivatives depends upon the size of the initial ERE. However, several of these derivative EREs can be equivalent to each other. One can check the equivalence of EREs using coinductive technique as described in Section 3, that generates a set of equivalent EREs, called *circularities*. In Section 4, we show how circularities can be used to construct an efficient algorithm that generates optimal DFAs from EREs. In Section 5, we describe an implementation of this algorithm and give performance analysis results. We also made available on the internet a CGI interface to this algorithm.

2 Extended Regular Expressions and Derivatives

In this section we recall extended regular expressions and their derivatives.

2.1 Extended Regular Expressions

Extended regular expressions (ERE) define languages by inductively applying union (+), concatenation (·), Kleene Closure ($_*$), intersection (∩), and complementation (¬). More precisely, for an alphabet E, whose elements are called *events* in this paper, an ERE over E is defined as follows, where $a \in E$:

$$R ::= \emptyset \mid \epsilon \mid a \mid R + R \mid R \cdot R \mid R^* \mid R \cap R \mid \neg R.$$

The language defined by an expression R, denoted by $\mathcal{L}(R)$, is defined inductively as

$$\begin{aligned}
\mathcal{L}(\emptyset) &= \emptyset, \\
\mathcal{L}(\epsilon) &= \{\epsilon\}, \\
\mathcal{L}(A) &= \{A\}, \\
\mathcal{L}(R_1 + R_2) &= \mathcal{L}(R_1) \cup \mathcal{L}(R_2), \\
\mathcal{L}(R_1 \cdot R_2) &= \{w_1 \cdot w_2 \mid w_1 \in \mathcal{L}(R_1) \text{ and } w_2 \in \mathcal{L}(R_2)\}, \\
\mathcal{L}(R^*) &= (\mathcal{L}(R))^*, \\
\mathcal{L}(R_1 \cap R_2) &= \mathcal{L}(R_1) \cap \mathcal{L}(R_2), \\
\mathcal{L}(\neg R) &= \Sigma^* \setminus \mathcal{L}(R).
\end{aligned}$$

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 Law: $R_1 \cap R_2 = \neg(\neg R_1 + \neg R_2)$. 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

$$R ::= R + R \mid R \cdot R \mid R^* \mid \neg R \mid a \mid \epsilon \mid \emptyset.$$

2.2 Derivatives

In this subsection we recall the notion of *derivative*, or "residual" (see [2,1], where several interesting properties of derivatives are also presented). It is based on the idea of "event consumption", 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 $w \in R\{a\}$.

In the rest of the paper assume defined the typical operators on EREs and consider that the operator $\_ + \_$ is associative and commutative and that the operator $\_ \cdot \_$ is associative. In other words, reasoning is performed modulo the equations:

$$\begin{aligned}(R_1 + R_2) + R_3 &= R_1 + (R_2 + R_3),\\ R_1 + R_2 &= R_2 + R_1,\\ (R_1 \cdot R_2) \cdot R_3 &= R_1 \cdot (R_2 \cdot R_3).\end{aligned}$$

We next consider an operation $\_\{\_\}$ which takes an ERE and an event, and give several equations which define its operational semantics recursively, on the structure of regular expressions:

$$\begin{aligned}(R_1 + R_2)\{a\} &= R_1\{a\} + R_2\{a\} && (1)\\ (R_1 \cdot R_2)\{a\} &= (R_1\{a\}) \cdot R_2 + \texttt{if } (\epsilon \in R_1) \texttt{ then } R_2\{a\} \texttt{ else } \emptyset \texttt{ fi} && (2)\\ (R^*)\{a\} &= (R\{a\}) \cdot R^* && (3)\\ (\neg R)\{a\} &= \neg(R\{a\}) && (4)\\ b\{a\} &= \texttt{if } (b == a) \texttt{ then } \epsilon \texttt{ else } \emptyset \texttt{ fi} && (5)\\ \epsilon\{a\} &= \emptyset && (6)\\ \emptyset\{a\} &= \emptyset && (7)\end{aligned}$$

The right-hand sides of these equations 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 equations:

$$\texttt{if } (\mathit{true}) \texttt{ then } R_1 \texttt{ else } R_2 \texttt{ fi} = R_1 \qquad (8)$$

$$\texttt{if } (\mathit{false}) \texttt{ then } R_1 \texttt{ else } R_2 \texttt{ fi} = R_2 \qquad (9)$$

We assume a set of equations that properly define boolean expressions and reasoning. 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 defined via the following equations:

$$\epsilon \in (R_1 + R_2) = (\epsilon \in R_1) \vee (\epsilon \in R_2) \tag{10}$$

$$\epsilon \in (R_1 \cdot R_2) = (\epsilon \in R_1) \wedge (\epsilon \in R_2) \tag{11}$$

$$\epsilon \in (R^*) = \mathit{true} \tag{12}$$

$$\epsilon \in (\neg R) = \mathit{not}(\epsilon \in R) \tag{13}$$

$$\epsilon \in b = \mathit{false} \tag{14}$$

$$\epsilon \in \epsilon = \mathit{true} \tag{15}$$

$$\epsilon \in \emptyset = \mathit{false} \tag{16}$$

The 16 equations above are natural and intuitive. [27] shows that these equations, when regarded as rewriting rules are terminating and ground Church-Rosser (modulo associativity and commutativity of $\_ + \_$ and modulo associativity of $\_ \cdot \_$), so they can be used as a functional procedure to calculate derivatives. Due to the fact that the 16 equations defining the derivatives can generate useless terms, in order to keep EREs compact we also propose defining several *simplifying equations*, including at least the following:

$$\emptyset + R = R,$$

$$\emptyset \cdot R = \emptyset,$$

$$\epsilon \cdot R = R,$$

$$R + R = R.$$

The following result (see, e.g., [27] for a proof) gives a simple procedure, based on derivatives, to test whether a word belongs to the language of an ERE:

Theorem 2.1 *For any ERE $\mathcal{R}$ and any events a, a_1, a_2, ..., a_n in A, the following hold:*

1) $a_1a_2...a_n \in \mathcal{L}(R\{a\})$ if and only if $aa_1a_2...a_n \in \mathcal{L}(R)$; and

2) $a_1a_2...a_n \in \mathcal{L}(R)$ if and only if $\epsilon \in R\{a_1\}\{a_2\}...\{a_n\}$.

3 Hidden Logic and Coinduction

We use circular coinduction, defined rigorously in the context of hidden logics and implemented in the BOBJ system [26,8,9], to test whether two EREs are equivalent, that is, if they have the same language. Since the goal of this paper is to translate an ERE into a minimal DFA, standard techniques for checking equivalence, such as translating the two expressions into DFAs and then comparing those, do not make sense in this framework. A particularly appealing aspect of circular coinduction in the framework of EREs is that it does not only show that two EREs are equivalent, but also generates a larger set of equivalent EREs which will all be used in order to generate the target DFA.

Hidden logic is a natural extension of algebraic specification which benefits of a series of generalizations in order to capture various natural notions of behavioral equivalence found in the literature. It distinguishes *visible* sorts for data from *hidden* sorts for states, with states *behaviorally equivalent* if and only if they are indistinguishable under a formally given set of experiments. To keep the presentation simple and self contained, in this section we define an oversimplified version of hidden logic together with its associated circular coinduction proof rule, still general enough to support defining and proving EREs behaviorally equivalent.

3.1 Algebraic Preliminaries

The reader is assumed familiar with basic equational logic and algebra in this section. We recall a few notions in order to just make our notational conventions precise. An S-sorted signature Σ is a set of sorts/types S together with operational symbols on those, and a Σ-algebra A is a collection of sets $\{A_s \mid s \in S\}$ and a collection of functions appropriately defined on those sets, one for each operational symbol. Given an S-sorted signature Σ and an S-indexed set of variables Z, let $T_\Sigma(Z)$ denote the Σ-term algebra over variables in Z. If $V \subseteq S$ then $\Sigma\restriction_V$ is a V-sorted signature consisting of all those operations in Σ with sorts entirely in V. We may let $\sigma(X)$ denote the term $\sigma(x_1, ..., x_n)$ when the number of arguments of σ and their order and sorts are not important. If only one argument is important, then to simplify writing we place it at the beginning; for example, $\sigma(t, X)$ is a term having σ as root with only variables as arguments except one, and we do not care which one, which is t. If t is a Σ-term of sort s' over a special variable $*$ of sort s and A is a Σ-algebra, then $A_t :\ A_s \to A_{s'}$ is the usual interpretation of t in A.

3.2 Behavioral Equivalence, Satisfaction and Specification

Given disjoint sets V, H called *visible* and *hidden sorts*, a *hidden* (V, H)-*signature*, say Σ, is a many sorted $(V \cup H)$-signature. A *hidden subsignature of* Σ is a hidden (V, H)-signature Γ with $\Gamma \subseteq \Sigma$ and $\Gamma\restriction_V = \Sigma\restriction_V$. The *data signature* is $\Sigma\restriction_V$. An operation of visible result not in $\Sigma\restriction_V$ is called an *attribute*, and a hidden sorted operation is called a *method*.

Unless otherwise stated, the rest of this section assumes fixed a hidden signature Σ with a fixed subsignature Γ. Informally, Σ-algebras are universes of possible states of a system, i.e., "black boxes," where one is only concerned with behavior under experiments with operations in Γ, where an experiment is an observation of a system attribute after perturbation; this is formalized below.

A Γ-*context for sort* $s \in V \cup H$ is a term in $T_\Gamma(\{* : s\})$ with one occurrence of $*$. A Γ-context of visible result sort is called a Γ-*experiment*. If c is a context for sort h and $t \in T_{\Sigma,h}$ then $c[t]$ denotes the term obtained from c by substituting t for $*$; we may also write $c[*]$ for the context itself.

Given a hidden Σ-algebra A with a hidden subsignature Γ, for sorts $s \in (V \cup H)$, we define Γ-*behavioral equivalence of* $a, a' \in A_s$ by $a \equiv_\Sigma^\Gamma a'$ iff $A_c(a) = A_c(a')$ for all Γ-experiments c; we may write $\equiv$ instead of $\equiv_\Sigma^\Gamma$ when Σ and Γ can be inferred from context. We require that all operations in Σ are compatible with $\equiv_\Sigma^\Gamma$. Note that behavioral equivalence is the identity on visible sorts, since the trivial contexts $* : v$ are experiments for all $v \in V$. A major result in hidden logics, underlying the foundations of coinduction, is that Γ-behavioral equivalence is the largest equivalence which is identity on visible sorts and which is compatible with the operations in Γ.

Behavioral satisfaction of equations can now be naturally defined in terms of behavioral equivalence. A hidden Σ-algebra A Γ-*behaviorally satisfies* a Σ-equation $(\forall X)\ t = t'$, say e, iff for each $\theta : X \to A$, $\theta(t) \equiv_\Sigma^\Gamma \theta(t')$; in this case we write $A \models\!\!\!\equiv_\Sigma^\Gamma e$. If E is a set of Σ-equations we then write $A \models\!\!\!\equiv_\Sigma^\Gamma E$ when A Γ-behaviorally satisfies each Σ-equation in E. We may omit Σ and/or Γ from $\models\!\!\!\equiv_\Sigma^\Gamma$ when they are clear.

A *behavioral* Σ-*specification* is a triple (Σ, Γ, E) where Σ is a hidden signature, Γ is a hidden subsignature of Σ, and E is a set of Σ-sentences equations. Non-data Γ-operations (i.e., in $\Gamma - \Sigma\restriction_V$) are called *behavioral*. A Σ-algebra A *behaviorally satisfies* a behavioral specification $\mathcal{B} = (\Sigma, \Gamma, E)$ iff $A \models\!\!\!\equiv_\Sigma^\Gamma E$, in which case we write $A \models\!\!\!\equiv \mathcal{B}$; also $\mathcal{B} \models\!\!\!\equiv e$ iff $A \models\!\!\!\equiv \mathcal{B}$ implies $A \models\!\!\!\equiv_\Sigma^\Gamma e$.

EREs can be very naturally defined as a behavioral specification. The enormous benefit of doing so is that the behavioral inference, including most importantly coinduction, provide a *decision procedure* for equivalence of EREs. [8] shows how standard regular expressions (without negation) can be defined as a behavioral specification, a BOBJ implementation, and also how BOBJ with its circular coinductive rewriting algorithm can prove automatically several equivalences of regular expressions. Related interesting work can also be found in [28]. In this paper we extend that to general EREs, generate minimal observer monitors, and also give several other examples.

Example 3.1 A behavioral specification of EREs defines a set of two visible sorts $V = \{Bool,\ Event\}$, one hidden sort $H = \{Ere\}$, one behavioral attribute $\epsilon \in \_ :\ Ere \to Bool$ and one behavioral method, the derivative, $\_\{\_\} :\ Ere \times Event \to Ere$, together with all the other operations in Subsection 2.1 defining EREs, including the events in E which are defined as visible constants of sort *Event*, and all the equations in Subsection 2.2. We call it the *ERE behavioral specification* and let $\mathcal{B}_{ERE}$ denote it.

Since the only behavioral operators are the test for ϵ membership and the derivative, it follows that the experiments have exactly the form $\epsilon \in *\{a_1\}\{a_2\}...\{a_n\}$, for any events a_1, a_2, ..., a_n. In other words, an experiment consists of a series of derivations followed by an ϵ membership test, and therefore two regular expressions are *behavioral equivalent* if and only if they cannot be distinguished by such experiments. Notice that the above reasoning applies within *any algebra* satisfying the presented behavioral specification. The one we are interested in is, of course, the *free* one, whose set carriers contain exactly the extended regular expressions as presented in Subsection 2.1, and the operations have the obvious interpretations. We informally call it the *ERE algebra*.

Letting $\equiv$ denote the behavioral equivalence relation generated on the ERE algebra, then Theorem 2.1 immediately yields the following important result.

Theorem 3.2 *If R_1 and R_2 are two EREs then $R_1 \equiv R_2$ if and only if $\mathcal{L}(R_1) = \mathcal{L}(R_2)$.*

This theorem allows us to prove equivalence of EREs by making use of behavioral inference in the ERE behavioral specification, from now on simply referred to by $\mathcal{B}$, including (especially) circular coinduction. The next section shows how circular coinduction works and how it can be used to show EREs equivalent.

3.3 Circular Coinduction as an Inference Rule

In the simplified version of hidden logics defined above, the usual equational inference rules, i.e., reflexivity, symmetry, transitivity, substitution and congruence [26] are all sound for behavioral satisfaction. However, equational reasoning can derive only a very limited amount of interesting behavioral equalities. For that reason, *circular coinduction* has been developed as a very powerful automated technique to show behavioral equivalence. We let $\Vdash$ denote the relation being defined by the equational rules plus circular coinduction, for deduction from a specification to an equation.

Before we present circular coinduction formally, we give the reader some intuitions by duality to structural induction. The reader who is only interested in using the presented procedure or who is not familiar with structural induction, can skip this paragraph. Inductive proofs show equality of terms $t(x), t'(x)$ over a given variable x (seen as a constant) by showing $t(\sigma(x))$ equals $t'(\sigma(x))$ for all σ in a basis, while circular coinduction shows terms t, t' behaviorally equivalent by showing equivalence of $\delta(t)$ and $\delta(t')$ for all behavioral operations δ. Coinduction applies behavioral operations at the top, while structural induction applies generator/constructor operations at the bottom. Both induction and circular coinduction assume some "frozen" instances of t, t' equal when checking the inductive/coinductive step: for induction, the terms are frozen at the bottom by replacing the induction variable by a constant, so that no other terms can be placed beneath the induction variable, while for coinduction, the terms are frozen at the top, so that they cannot be used as subterms of other terms (with some important but subtle exceptions which are not needed here; see [9]).

Freezing terms at the top is elegantly handled by a simple trick. Suppose every specification has a special visible sort b, and for each (hidden or visible) sort s in the specification, a special operation $[\_] : \ s \rightarrow b$. No equations are assumed for these operations and no user defined sentence can refer to them; they are there for technical reasons. Thus, with just the equational inference rules, for any behavioral specification $\mathcal{B}$ and any equation $(\forall X)\ t = t'$, it is necessarily the case that $\mathcal{B} \ \Vdash (\forall X)\ t = t'$ iff $\mathcal{B} \ \Vdash (\forall X)\ [t] = [t']$. The rule below preserves this property. Let the sort of t, t' be hidden; then

Circular Coinduction:

$$\frac{\mathcal{B} \cup \{(\forall X)\ [t] = [t']\} \ \Vdash (\forall X, W)\ [\delta(t, W)] = [\delta(t', W)], \text{for all appropriate } \delta \in \Gamma}{\mathcal{B} \ \Vdash (\forall X)\ t = t'}$$

We call the equation $(\forall X)\ [t] = [t']$ added to $\mathcal{B}$ a **circularity**; it could just as well have been called a coinduction hypothesis or a co-hypothesis, but we find the first name more intuitive because from a coalgebraic point of view, coinduction is all about finding circularities.

Theorem 3.3 *The usual equational inference rules together with* Circular Coinduction *are sound. That means that if* $\mathcal{B} \ \Vdash (\forall X)\ t = t'$ *and* $sort(t, t') \neq b$, *or if* $\mathcal{B} \ \Vdash (\forall X)\ [t] = [t']$, *then* $\mathcal{B} \ \models\!\equiv (\forall X)\ t = t'$.

Example 3.4 Suppose that we want to show that the EREs $(a + b)^*$ and $(a^* b^*)^*$ admit the same language. By Theorem 3.2, we can instead show that $\mathcal{B}_{ERE} \models\!\equiv\ (\forall \emptyset)\ (a + b)^* = (a^* b^*)^*$. Notice that a and b are treated as constant events here; one can also prove the result when a and b are variables, but one would need to first make use of the theorem of hidden constants [26]. To simplify writing, we omit the empty quantifier of equations. By the Circular Coinduction rule, one generates the following three proof obligations

$$\mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^* b^*)^*]\} \Vdash [\epsilon \in (a + b)^*] = [\epsilon \in (a^* b^*)^*],$$
$$\mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^* b^*)^*]\} \Vdash [(a + b)^* \{a\}] = [(a^* b^*)^* \{a\}],$$
$$\mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^* b^*)^*]\} \Vdash [(a + b)^* \{b\}] = [(a^* b^*)^* \{b\}].$$

The first proof task follows immediately by using the equations in $\mathcal{B}$ as rewriting rules, while the other two tasks reduce to

$$\mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^* b^*)^*]\} \Vdash [(a + b)^*] = [a^* (a^* b^*)^*],$$
$$\mathcal{B}_{ERE} \cup \{[(a + b)^*] = [(a^* b^*)^*]\} \Vdash [(a + b)^*] = [b^* (a^* b^*)^*].$$

By applying Circular Coinduction twice, after simplifying the two obvious proof tasks stating the ϵ membership, one gets the following four proof obligations

$$\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash [(a+b)^*]\{a\} = [a^*(a^*b^*)^*\{a\}],$$
$$\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash [(a+b)^*]\{b\} = [a^*(a^*b^*)^*\{b\}],$$
$$\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash [(a+b)^*]\{a\} = [b^*(a^*b^*)^*\{a\}],$$
$$\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash [(a+b)^*]\{b\} = [b^*(a^*b^*)^*\{b\}],$$

which, after simplification translate into

$$\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash [(a+b)^*] = [a^*(a^*b^*)^*],$$
$$\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash [(a+b)^*] = [b^*(a^*b^*)^*],$$
$$\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash [(a+b)^*] = [a^*(a^*b^*)^*],$$
$$\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash [(a+b)^*] = [b^*(a^*b^*)^*],$$

Again by applying circular coinduction we get

$$\begin{aligned}
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*],\ [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash& \\
[(a+b)^*] = [a^*(a^*b^*)^*],& \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [a^*(a^*b^*)^*],\ [(a+b)^*] = [b^*(a^*b^*)^*]\} \Vdash& \\
[(a+b)^*] = [b^*(a^*b^*)^*],& \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*],\ [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash& \\
[(a+b)^*] = [a^*(a^*b^*)^*],& \\
\mathcal{B}_{ERE} \cup \{[(a+b)^*] = [(a^*b^*)^*], [(a+b)^*] = [b^*(a^*b^*)^*],\ [(a+b)^*] = [a^*(a^*b^*)^*]\} \Vdash& \\
[(a+b)^*] = [b^*(a^*b^*)^*],&
\end{aligned}$$

which now follow all immediately. Notice that BOBJ uses the newly added (to $\mathcal{B}_{ERE}$) equations as rewriting rules when it applies its circular coinductive rewriting algorithm, so the proof above is done slightly differently, but entirely automatically.

Example 3.5 Suppose now that one wants to show that $\neg(a^*b) \equiv \epsilon + a^* + (a+b)^*b(a+b)(a+b)^*$. One can also do it entirely automatically by circular coinduction as above, generating the following list of circularities:

$$\begin{aligned}
[\neg(a^*b)] &= [\epsilon + a^* + (a+b)^*b(a+b)(a+b)^*], \\
[\neg(\epsilon)] &= [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^*], \\
[\neg(\emptyset)] &= [(a+b)^*b(a+b)(a+b)^* + (a+b)^*], \\
[\neg(\emptyset)] &= [(a+b)^*b(a+b)(a+b)^* + (a+b)(a+b)^* + (a+b)^*].
\end{aligned}$$

Example 3.6 One can also show by circular coinduction that concrete EREs satisfy systems of guarded equations. This is an interesting but unrelated subject, so we do not discuss it in depth here. However, we show how easily one can prove by coinduction that a^*b is the solution of the equation $R = a \cdot R + b$. This equation can be given by adding a new ERE constant r to $\mathcal{B}_{ERE}$, together with the equations $\epsilon \in r = \mathit{false}$, $r\{a\} = r$, and $r\{b\} = \epsilon$. Circular Coinduction applied on the goal $r = a^*b$ generates the proof tasks:

$$\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} \Vdash [\epsilon \in r] = [\epsilon \in a^*b],$$
$$\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} \Vdash [r\{a\}] = [a^*b\{a\}],$$
$$\mathcal{B}_{ERE} \cup \{[r] = [a^*b]\} \Vdash [r\{b\}] = [a^*b\{b\}],$$

which all follow immediately.

The following says that circular coinduction provides a decision procedure for equivalence of EREs.

Theorem 3.7 *If R_1 and R_2 are two EREs, then $\mathcal{L}(R_1) = \mathcal{L}(R_2)$ if and only if $\mathcal{B}_{ERE} \Vdash R_1 = R_2$. Moreover, since the rules in $\mathcal{B}_{ERE}$ are ground Church-Rosser and terminating,* circular coinductive rewriting*[8,9], which iteratively rewrites proof tasks to their normal forms followed by a one step coinduction if needed, gives a decision procedure for ERE equivalence.*

4 Generating Minimal DFA Monitors by Coinduction

In this section we show how one can use the set of circularities generated by applying the circular coinduction rules in order to generate a minimal DFA from any ERE. This DFA can then be used as an optimal monitor for that ERE. The main idea here is to associate states in DFA to EREs obtained by deriving the initial ERE; when a new ERE is generated, it is tested for equivalence with all the other already generated EREs by using the coinductive procedure presented in the previous section. A crucial observation which significantly reduces the complexity of our procedure is that, once an equivalence is proved by circular coinductive rewriting, the entire set of circularities accumulated represent equivalent EREs. These can be used to later quickly infer the other equivalences, without having to generate the same circularities over and over again.

Since BOBJ does not (yet) provide any mechanism to return the set of circularities accumulated after proving a given behavioral equivalence, we were unable to use BOBJ to implement our optimal monitor generator. Instead, we have implemented our own version of coinductive rewriting engine for EREs, which is described below.

We are given an initial ERE R_0 over alphabet A and from that we want to generate the equivalent minimal DFA $D = (S, A, \delta, s_0, F)$, where S is the set of states, $\delta : S \times A \to S$ is the transition function, s_0 is the initial state, and $F \subseteq S$ is the set of final states. The coinductive rewriting engine explicitly accumulates the proven circularities in a set. The set is initialized to an empty set at the beginning of the algorithm. It is updated with the accumulated circularities whenever we prove equivalence of two regular expressions in the algorithm. The algorithm maintains the set of states S in the form of non-equivalent EREs. At the beginning of the algorithm S is initialized with a single element, which is the given ERE R_0. Next, we generate all the derivatives of the initial ERE one by one in a depth first manner. A derivative $R_x = R\{x\}$ is added to the set S, if the set does not contain any ERE equivalent to the derivative R_x. We then extend the transition function by setting $\delta(R, x) = R_x$. If an ERE R' equivalent to the derivative already exists in the set S, we extend the transition function by setting $\delta(R, x) = R'$. To check if an ERE equivalent to the derivative R_x already exists in the set S, we sequentially go through all the elements of the set S and try to prove its equivalence with R_x. In testing the equivalence we first add the set of circularities to the initial $\mathcal{B}$. Then we invoke the coinductive procedure. If for some ERE $R' \in S$, we are able to prove that $R' \equiv R_x$ i.e $\mathcal{B} \cup Eq_{\text{all}} \cup Eq_{\text{new}} \Vdash \mathrm{R}' = \mathrm{R}_x$, then we add the new equivalences Eq_{new}, created by the coinductive procedure, to the set of circularities. Thus we reuse the already proven equivalences in future proofs.

The derivatives of the initial ERE R_0 with respect to all events in the alphabet A are generated in a depth first fashion. The pseudo code for the whole algorithm is given in Figure 1.

```
dfs($R$)
begin
  foreach $x \in A$ do
    $R_x \leftarrow R\{x\}$;
    if $\exists R' \in S$ such that $\mathcal{B} \cup Eq_{\text{all}} \cup Eq_{\text{new}} \Vdash \mathrm{R}' = \mathrm{R}_x$ then
      $\delta(R, x) = R'$;   $Eq_{\text{all}} \leftarrow Eq_{\text{all}} \cup Eq_{\text{new}}$
    else $S \leftarrow S \cup \{R_x\}$;   $\delta(R, x) = R_x$;   dfs($R_x$); fi
  endfor
end
```

Fig. 1. ERE to minimal DFA generation algorithm

In the procedure **dfs** the set of final states F consists of the EREs from S which contain ϵ. This can be tested efficiently using the equations (10-16) in Subsection 2.2. The DFA generated by the procedure **dfs** may now contain some states which are non-final and from which the DFA can never reach a final state. We remove these redundant states by doing a breadth first search in backward direction from the final states. This can be done in time linear in the size of the DFA.

Theorem 4.1 *If D is the DFA generated for a given ERE R by the above algorithm then*

1) $\mathcal{L}(D) = \mathcal{L}(R)$,

2) D is the minimal DFA accepting $\mathcal{L}(R)$.

Proof:

1) Suppose $a_1a_2\ldots a_n \in \mathcal{L}(R)$. Then $\epsilon \in R\{a_1\}\{a_2\}\ldots\{a_n\}$. Let $R_i = R\{a_1\}\{a_2\}\ldots\{a_i\}$; then $R_{i+1} = R_i\{a_{i+1}\}$. To prove that $a_1a_2\ldots a_n \in \mathcal{L}(D)$, we use induction to show that for each $1 \leq i \leq n$, $R_i \equiv \delta(R, a_1a_2\ldots a_i)$. For the base case if $R_1 \equiv R\{a_1\}$ then **dfs** extends the transition function by setting $\delta(R, a_1) = R$. Therefore, $R_1 \equiv R = \delta(R, a_1)$. If $R_1 \not\equiv R$ then **dfs** extends δ by setting $\delta(R, a_1) = R_1$. So $R_1 \equiv \delta(R, a_1)$ holds in this case also. For the induction step let us assume that $R_i \equiv R' = \delta(R, a_1a_2\ldots a_i)$. If $\delta(R', a_{i+1}) = R''$ then from the **dfs** procedure we can see that $R'' \equiv R'\{a_{i+1}\}$. However, $R_i\{a_{i+1}\} \equiv R'\{a_{i+1}\}$. So $R_{i+1} \equiv R'' = \delta(R', a_{i+1}) = \delta(R, a_1a_2\ldots a_{i+1})$. Also notice $\epsilon \in R_n \equiv \delta(R, a_1a_2\ldots a_n)$; this implies that $\delta(R, a_1a_2\ldots a_n)$ is a final state and hence $a_1a_2\ldots a_n \in \mathcal{L}(D)$.

   Now suppose $a_1a_2\ldots a_n \in \mathcal{L}(D)$. The proof that $a_1a_2\ldots a_n \in \mathcal{L}(R)$ goes in a similar way by showing that $R_i \equiv \delta(R, a_1, a_2\ldots a_i)$.

2) If DFA D is not minimal then there exists at least two states p and q in D such that p and q are *equivalent* [15] i.e. $\forall w \in A^* : \delta(p, w) \in F$ iff $\delta(q, w) \in F$, where F is the set of final states. This means, if R_1 and R_2 are the EREs associated with p and q respectively in **dfs** then $\mathcal{L}(R_1) = \mathcal{L}(R_2)$ i.e. $R_1 \equiv R_2$. But **dfs** ensures that no two EREs representing the states of the DFA are equivalent. So we get a contradiction.

$\square$

5 Implementation and Evaluation

We have implemented the coinductive rewriting engine in the rewriting specification language Maude 2.0 [4]. The interested readers can download the implementation from the website `http://fsl.cs.uiuc.edu/rv/`. The operations on extended regular languages that are supported by our implementation are `~_` for negation, `_ *` for Kleene Closure, `_ _` for concatenation, `_ & _` for intersection, and `_ + _` for union in increasing order of precedence. Here, the intersection operator `_ & _` is a syntactic sugar and is translated to an ERE containing union and negation using De Morgan's Law:

```
eq R1 & R2 = ~ (~ R1 + ~ R2) .
```

To evaluate the performance of the algorithm we have generated the minimal DFA for all possible EREs of size up to 9. Surprisingly, the size of any DFA for EREs of size up to 9 did not exceed 9. Here the number of states gives the size of a DFA. The following table shows the performance of our procedure for the worst EREs of a given size. The code is executed on a Pentium 4 2.4GHz, 4 GB RAM linux machine.

Size	ERE	no. of states in DFA	Time (ms)	Rewrites
4	¬ (a b)	4	< 1	863
5	(a ¬ b) *	4	< 1	1370
6	¬ ((a ¬ b) *)	4	1	1453
7	¬ (a ¬ a a)	6	1	2261
8	¬ ((a ¬ b) * b)	7	1	3778
9	¬ (a ¬ a b) b	9	5	9717

Example 5.1 In particular, for the ERE ¬ (a ¬ a b) b the generated minimal DFA is given in Figure 2.