Distributed Finite-State Runtime Monitoring with Aggregated Events*

Kevin Falzon[1], Eric Bodden[1], and Rahul Purandare[2]

[1] European Center for Security and Privacy by Design (EC-SPRIDE)
{kevin.falzon, eric.bodden}@ec-spride.de
[2] Department of Computer Science and Engineering
University of Nebraska-Lincoln
rpuranda@cse.unl.edu

Abstract. Security information and event management (SIEM) systems usually consist of a centralized monitoring server that processes events sent from a large number of hosts through a potentially slow network. In this work, we discuss how monitoring efficiency can be increased by switching to a model of *aggregated traces*, where monitored hosts buffer events into lossy but compact batches. In our trace model, such batches retain the number and types of events processed, but not their order.
We present an algorithm for automatically constructing, out of a regular finite-state property definition, a monitor that can process such aggregated traces. We discuss the resultant monitor's complexity and prove that it determines the set of possible next states without producing false negatives and with a precision that is optimal given the reduced information the trace carries.

1 Introduction

In this work, we consider a common scenario to which runtime monitoring is nowadays often applied, namely that of security information and event management (SIEM) systems [9]. Such systems, mainly designed for intrusion detection or the discovery of insider attacks, usually comprise a centralized monitoring server that processes events sent from a large number of hosts within a local company network. At peak times, these hosts might be slowed down significantly, as they block while trying to synchronously send off event information to an overloaded monitoring server [11].

We address this problem by proposing a trace model in which the monitored hosts can aggregate parts of the event stream, retaining the number and types of events processed, but not their order. Discarding ordering information allows event streams to be compressed effectively, whilst retaining event frequencies and types maintains a certain level of precision. In comparison to related work [5, 12], this trace model is not probabilistic and does not allow for "gaps" in the event stream—every occurring event is

* This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE. (www.ec-spride.de)
This is a revised edition of the version presented at RV2013. The definition of modulo, and consequently, the evaluation function, has been redefined. The transformation of constraint expressions under unbounded iteration has also been modified to correctly handle star depth.

indeed accounted for. The aggregated trace rather provides an *over*-approximation that implicitly includes all permutations of the original trace it represents.

As the main contribution of this paper, we define an algorithm to automatically derive, from a finite-state property definition, a runtime monitor that can process such aggregated traces. We prove that the current formulation of our algorithm produces monitors that are guaranteed not to miss property violations. As we also show, the monitor processes the compressed event stream in bounded time, while not producing more false positives than a naïve monitor that traverses the original property state machine using all possible permutations of the original event trace. Our algorithm can be easily parameterized with different acceptance conditions that can decrease the number of warnings further while allowing for some missed violations.

To summarize, this paper presents the following original contributions:

- a general trace model in which trace producers supply, for certain periods of time, aggregated information about events that occurred during those periods,
- an algorithm that automatically constructs a monitor for such traces from a finite-state property definition,
- a proof that the algorithm always converges and that the resulting monitor is guaranteed not to miss any actual violations and is optimally precise given the aggregated trace information it receives, and
- a complexity estimation for the resulting monitoring algorithm.

The remainder of this paper is structured as follows. Section 2 presents a motivating example, leading to a description of *constraint automata* in Section 3. Section 4 defines a process for translating finite state automata into constraint automata, and Section 5 regards complexity and implementation considerations. The paper discusses related work in Section 6 and concludes in Section 7.

2 Introductory Example

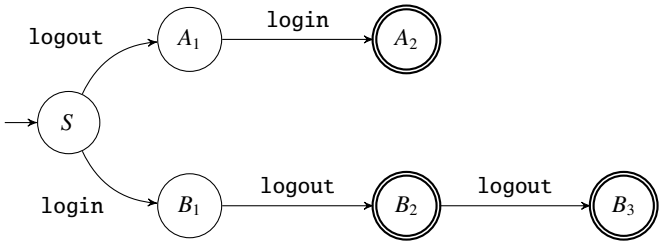


Fig. 1: A deterministic automaton.

Consider the automaton illustrated in Figure 1. We follow the model of a trace classifier, where different accepting states can cause different error messages, for instance as implemented through JavaMOP [8]. In this example, both the event sequences

`logout` · `login` and `login` · `logout` are in the language of the automaton, yet they lead to different states, which in our model means that they would be classified differently.

In this work, we consider the situation where monitored hosts in a distributed system wish to compress data to decrease the load on the network. One of the most effective ways to compress an event trace is to aggregate all occurring events in a data structure that captures the events' types and frequencies but discards their order. In our example, the host could retain the number of times that `login` and `logout` events were observed. The monitor would then face the challenge that, on receiving a compressed batch of two events, the next state would be uncertain (A_2 or B_2). Importantly, though, receiving a subsequent `logout` event would confirm that the automaton must be in state B_3, and would cause the automaton to converge onto a single state.

A naïve approach to processing such "compressed traces" with incomplete information about ordering constraints would be to define a special transition procedure over an unmodified finite-state property automaton. In such a model, on receiving an aggregated batch of events, one would be able to determine the possible next states by traversing the automaton with each legal permutation of events that satisfies the aggregated input, be it through brute-force generation of traces, or by using the automaton as a generator. Using the example illustrated in Figure 1, consider the case where the monitor has observed the following compressed batch of events:

$$\{\langle \texttt{logout}, 2\rangle, \langle \texttt{login}, 1\rangle\}$$

This signifies the arrival of two `logout` events and one `login` event, without any information on the order in which they were received. In a different scenario, one might consider aggregate traces that only record N, that is, the number of events that occurred, without even recording their type. For such a model, any state reachable within N steps of the current state is a potential next state. Preserving the type restricts the possible next state set to a subset of these states. In general, the larger the window and number of aggregated events, the greater the uncertainty of the end state, as the automaton will have potentially progressed to a greater depth.

The problem with determining the next state set via naïve traversal is that the running time will grow exponentially with the number of observed events. Thus, this work proposes an ahead-of-time automaton transformation of finite-state property automata into a data structure which, on being presented with a current state and an input of aggregated events, computes the set of possible and valid next states *efficiently*, within a time bounded by the size of the structure rather than that of the input.

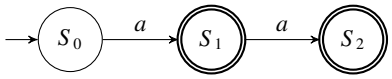
3 Constraint Automata

The following section introduces the notions of *constraints* and *constraint automata*, defining their structure, evaluation and traversal.

3.1 Overview

The transitions in a finite-state automaton determine the number, type and order of input elements required to move between states. In the scenario we consider, compressed inputs are *unordered*, containing information only pertaining to each element's frequency.

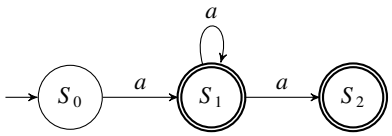
The problem of computing the set of next states can be reformulated in terms of reachability. Each state can be seen as having a set of associated conditions, or *constraints*, on the input. If a constraint evaluates to true, then the system can transition into that state. To compute a complete set of next states, one has to check these conditions for every reachable state.



State	Constraint
S_0	$\#_a^v = 0$
S_1	$\#_a^v = 1$
S_2	$\#_a^v = 2$

Fig. 2: A linear automaton with constraints on a trace v received at S_0.

Figure 2 illustrates a very simple linear automaton, and the conditions required for entering each other state when starting from S_0. The function $\#_a^v$ returns the frequency with which symbol a appears in trace v. The constraints defined are strict and unambiguous, referring to specific frequencies. Thus, for example, if one a is observed at S_0, then the system can only be in state S_1.

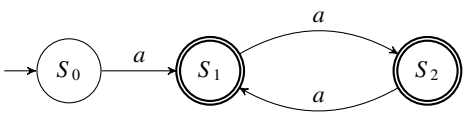


State	Constraint
S_0	$\#_a^v = 0$
S_1	$\#_a^v \geq 1$
S_2	$\#_a^v \geq 2$

Fig. 3: Modified constraints on introducing a loop at S_1.

Precision becomes an issue once *loops* enter the equation. Figure 3 illustrates an automaton similar to that shown in Figure 2, yet the addition of a self-loop has weakened the conditions, which can now only place a lower bound on the number of observed events. However, as can be seen in the example illustrated in Figure 4, loops do not always introduce ambiguity; in this example, the automaton goes to a unique state under any input despite the presence of loops.

In general, fixed sequences of transitions precisely define the number of elements that must be observed for the end state to be reached. Loops consume elements in multiples of the number of elements along their path. For instance, the self-loop on S_1 in Figure 3 consumes a symbols in multiples of one, while the loop formed between S_1 and S_2 in Figure 4 consumes a symbols in multiples of two. In the presence of loops, fixed sequences outside of loops will set a lower bound on the number of elements that must be consumed.



State	Constraint
S_0	$\#_a^v = 0$
S_1	$\#_a^v \geq 1 \wedge (\#_a^v - 1) \bmod 2 = 0$
S_2	$\#_a^v \geq 2 \wedge (\#_a^v - 2) \bmod 2 = 0$

Fig. 4: A flip-flop automaton. Following a mandatory single input, the automaton oscillates between S_1 and S_2, with the final state depending on whether an even or odd number of inputs has been received.

3.2 The Constraint Automaton Model

The examples illustrated defined constraints with respect to a single start state. By considering every state as an initial state, one may construct a *constraint automaton* that accepts aggregated event sets instead of single event inputs. A constraint automaton has a state for each state in the original automaton, and a transition (and consequently a constraint) for each pair of connected states. This allows the constraint automaton to deduce the next states from any configuration, facilitating the processing of sequences of compressed traces.

A constraint automaton is evaluated as a subset automaton, i.e., the "next state" is modelled as a set of states, as the loss of event ordering may lead to multiple valid traversals. In general, shorter compressed traces leave less room for ambiguity. Varying the degree of compression may help re-converge the automaton in instances where the current state set is large (note that a sequence of aggregated event sets that are all singletons is essentially equivalent to a regular trace).

Losing event ordering makes analysis inherently incomplete. More specifically, if a trace leads to a final state in a given finite-state automaton, then its compressed version may lead to multiple states in the derived constraint automaton. As will be seen in Section 6, while certain alternative approaches apply statistical methods to determine the most likely next state, a loss of information will generally introduce uncertainty. Given that this incompleteness cannot be eliminated, it is more relevant to reason in terms of *relative precision*, that is, if the permutations of a given trace lead to a certain set of next states in a finite state automaton, then that trace's compressed version must return *exactly the same set* of next states in the constraint automaton.

Before progressing any further, we first define the notion of an *aggregated event set*, and transforming a trace into such a set, as follows:

Definition 1 (Aggregated Event Set). *An aggregated event set for an alphabet Σ is a set of pairs mapping elements to frequencies, with one pair defined for each element in Σ. The set of possible aggregated events for alphabet Σ, denoted by $\mathcal{A}_\Sigma$, is defined as:*

$$\mathcal{A}_\Sigma \stackrel{def}{=} \{s \mid s \in 2^{\Sigma \times \mathbb{N}}, |s| = |\Sigma|, \\ c_1 \in s, c_2 \in s, c_1 = \langle a_1, n_1 \rangle, c_2 = \langle a_2, n_2 \rangle, (c_1 \neq c_2) \rightarrow (a_1 \neq a_2)\}$$

Definition 2 (Trace to Aggregated Count). *The aggregated event set for a trace v over events Σ, denoted by $\downarrow\#_\Sigma^v$, is defined as:*

$$\downarrow\#_\Sigma^v \stackrel{def}{=} \{\langle a, \#_a^v \rangle \mid a \in \Sigma\}$$

3.3 Regular Expressions as Constraints

Based on the defined constraint-automaton model, the next step is to devise a method for representing, deriving and evaluating the constraints. One possible constraint representation would be as *regular expressions*. For each pair of states in the original automaton, one would derive regular expressions that encompass all paths that lead from one state to the other. Several algorithms for deriving regular expressions from automata exist, such as the one described by Brzozowski [7]. Given any two states q and q' in an automaton, $\texttt{regex}_{\mathcal{D}}(q, q')$ will return a regular expression for the set of strings which, starting from q, lead to q', or $\perp$ if there are no paths between the two states.

To evaluate a compressed trace against a regular expression constraint, one would check whether the expression matches at least one legal permutation of the elements in the compressed trace. However, this procedure is computationally expensive, making regular expressions inadequate for representing constraints in a monitoring context.

3.4 Constraints and Constraint Expressions

Given the inadequacy of regular expressions for representing constraints, the remainder of this section details *constraint expressions*, which are more amenable to direct comparisons with compressed traces. Section 4 then describes a rewriting system for converting regular expressions into such constraint expressions.

Definition 3 (Modulus). *A* modulus *is a pair* $(m, k) \in \mathbb{N} \times \mathbb{N}$*, which we denote as* $m^{\cdot k}$.

Definition 4 (Basic Constraint). *A* basic constraint $\mathbb{C}$ *is a tuple of the form* $2^{M_{\mathbb{N}\cdot\Sigma}}$ *consisting of a set of* moduli*, a numerical* offset*, and the* symbol *being constrained.*

Example 1. The constraint on entering state S_1 from S_2 in the automaton illustrated in Figure 4 would be expressed as the following basic constraint:

$$\{2^{\cdot 0}\}_{1\cdot a}$$

For an observed aggregate input v, this constraint would be true when at least one a symbol has been observed (denoted by the subscript), and when $\#_a^v - 1$ is a multiple of 2. The precise evaluation procedure will be described in Section 3.5.

Definition 5 (Well-formed Constraint Vector). *A* constraint vector $\overrightarrow{\mathbb{C}}$ *is a set of basic constraints. For a constraint vector to be* well-formed *with respect to an automaton with alphabet* Σ*, it must contain exactly one basic constraint for each element of* Σ.

Example 2. A *well-formed constraint vector* for an automaton with $\Sigma \stackrel{\text{def}}{=} \{a, b\}$.

$$\left\{\{2^{\cdot 0}\}_{1\cdot a}, \emptyset_{3\cdot b}\right\}$$

Definition 6 (Constraint Expression). *A* constraint expression $\hat{\mathbb{C}}$ *is a logical formula of the form*

$$\hat{\mathbb{C}} := \overrightarrow{\mathbb{C}} \mid \hat{\mathbb{C}} \underline{\vee} \hat{\mathbb{C}} \mid \hat{\mathbb{C}} \underline{\cdot} \hat{\mathbb{C}} \mid \hat{\mathbb{C}}^{\underline{n}} \mid \hat{\mathbb{C}}^{\underline{*}} \mid \perp$$

The empty constraint expression is represented by $\bot$, and $\hat{\mathbb{C}} \veebar \bot \equiv \bot \veebar \hat{\mathbb{C}} \equiv \hat{\mathbb{C}} \cdot \bot \equiv \bot \cdot \hat{\mathbb{C}} \equiv \hat{\mathbb{C}}$, whereas $\bot^{*} \equiv \bot^{n} \equiv \bot$

Definition 7 (Disjunctive Constraint Expression). *A* Disjunctive Constraint Expression *(DCE) is a constraint expression consisting solely of well-formed constraint vectors and $\veebar$ operators.*

Example 3. Two examples of constraint expressions on an automaton with $\Sigma \stackrel{\text{def}}{=} \{a, b\}$, the latter being a DCE.

$$\left\{\{5^{\cdot 0}\}_{2\cdot a}, \emptyset_{3\cdot b}\right\}^{*} \tag{1}$$

$$\left\{\{5^{\cdot 0}\}_{2\cdot a}, \emptyset_{3\cdot b}\right\} \veebar \left\{\emptyset_{3\cdot a}, \{2^{\cdot 1}\}_{2\cdot b}\right\} \tag{2}$$

3.5 Evaluating Constraints

The function $\texttt{eval}_v(\dot{\mathbb{C}})$ evaluates a DCE $\dot{\mathbb{C}}$ on an aggregated event input v, and is defined as follows:

$$\begin{aligned}
\texttt{eval}_v(\bot) &\stackrel{\text{def}}{=} \mathit{false} \\
\texttt{eval}_v(\overrightarrow{\mathbb{C}}) &\stackrel{\text{def}}{=} \exists ks \in \mathbb{N}^{*}.\ \forall M_{i\cdot a} \in \overrightarrow{\mathbb{C}}.\ (M = \emptyset \wedge \#_a^v = i) \vee \\
&\qquad (M = \left\{m_1^{\cdot z_1}, m_2^{\cdot z_2}, \ldots m_n^{\cdot z_n}\right\} \wedge \textstyle\sum_{j=1}^{n} ks_{z_j} \times M_j = \#_a^v - i) \\
\texttt{eval}_v(\overrightarrow{\mathbb{C}}_1 \veebar \overrightarrow{\mathbb{C}}_2 \veebar \ldots \veebar \overrightarrow{\mathbb{C}}_n) &\stackrel{\text{def}}{=} \texttt{eval}_v(\overrightarrow{\mathbb{C}}_1) \vee \texttt{eval}_v(\overrightarrow{\mathbb{C}}_2) \vee \ldots \vee \texttt{eval}_v(\overrightarrow{\mathbb{C}}_n)
\end{aligned}$$

Example 4. $\texttt{eval}_{\{\langle a,9\rangle,\langle b,10\rangle\}}(\{5^{\cdot 1}, 2^{\cdot 2}\}_{0\cdot a}\{3^{\cdot 1}, 2^{\cdot 3}\}_{3\cdot b})$ is true, as $5k_1 + 2k_2 = 9$ and $3k_1 + 2k_3 = 7$ for $k_1 = 1$, $k_2 = 2$ and $k_3 = 2$.

3.6 Traversing a Constraint Automaton

Algorithm 1 details a general approach to traversing a constraint automaton. The algorithm is designed for online use, with the blocking `nextInputEventCount`() function returning aggregated event counts collected by the monitoring system. Naturally, this can readily be adapted for offline inputs.

When traversing an automaton, the algorithm must evaluate the constraint expression for each transition leaving the current state (line 9). Multiple constraint expressions may evaluate to *true* simultaneously, which results in a set of possible next states. Thus, the current automaton state must be modelled as a set of states, with the automaton potentially being in any of those states. This non-determinism may arise even if the original automaton was deterministic. For example, while the automaton illustrated in Figure 1 is deterministic, it has branches that accept two traces with differing order but equal event frequencies. Its constraint automaton (Figure 5) is thus afflicted by ambiguity, as an aggregated input of two events would lead to the automaton potentially being in either A_2 or B_2. A subsequent event would cause the automaton to converge onto B_3. As we detail in Section 5, determinizing the constraint automaton would not help, as the

Algorithm 1 On-line Traversal of a Constraint Automaton $\langle Q, q_0, \Sigma, F, \Gamma \rangle$

1: $Current \leftarrow \{q_0\}$ ▷ Start at initial state
2: **loop** ▷ Perpetual loop
3: **if** $(Current \cap F \neq \emptyset)$ **then** ▷ Check if potentially in a final state
4: `reportError`$(Current)$ ▷ Report current state set
5: **end if**
6: $v \leftarrow$ `nextInputEventCount`() ▷ Get next map of aggregated events
7: $next_Q \leftarrow \emptyset$ ▷ Reset next state set
8: **for all** $c \in Current$ **do** ▷ Determine next states for each current state
9: $next_Q \leftarrow next_Q \cup \left\{c' \mid (c, \dot{\mathbb{C}}, c') \in \Gamma, \texttt{eval}_v(\dot{\mathbb{C}})\right\}$
10: **end for**
11: $Current \leftarrow next_Q$ ▷ Update with computed next states
12: **end loop**

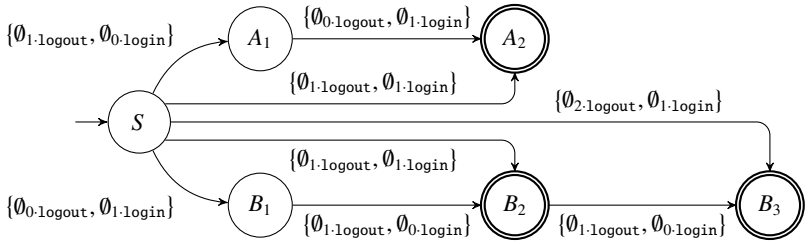


Fig. 5: The constraint automaton derived from the automaton in Figure 1.

resulting deterministic automaton would still require a transition function that evaluates the same set of transition constraints.

The size of *Current* may grow as well as shrink, the latter occurring when parallel traversals converge onto a state, or when members of the set do not lead to valid next states under the observed input. As presented, the algorithm never halts, instead reporting an error whenever *Current* contains *some* final state. This policy over-approximates error states, which may lead to false alarms. To reduce them, one may consider using other policies, such as reporting errors only when *Current* is composed entirely of final states. Alternatively, one may augment the automaton with probabilistic information, terminating based on the likelihood that the system is in an actual error state.

4 Constructing a Constraint Automaton from a Property FSA

The following defines the process of deriving constraint automata from finite state properties. The transformation is performed in two phases. In the first phase, *regular expressions* are constructed for every pair of states in the property. In the second phase, each regular expression is subsequently transformed into a constraint on frequencies.

4.1 Translating Regular Expressions into Constraint Expressions

The process of translating a regular expression into a constraint expression involves two steps. The first step transforms the regular expression into an *initial constraint*

expression, and is performed by applying the *regex-to-constraint expression* operator $\xrightarrow{\Sigma}$, defined as follows.

Definition 8 (Regex-To-CE). *Given a regular expression $\mathcal{R}$, one can derive a constraint expression $\hat{\mathbb{C}}$, whose vectors are well-formed with respect to an alphabet Σ. This is denoted by $\mathcal{R} \xrightarrow{\Sigma} \hat{\mathbb{C}} \stackrel{def}{=} \hat{\mathbb{C}} = [\![\mathcal{R}]\!]_\Sigma$, where $[\![]\!]_\Sigma$ is defined as:*

$$[\![\mathcal{R}_1\mathcal{R}_2]\!]_\Sigma \rightarrow [\![\mathcal{R}_1]\!]_\Sigma \mathbin{\underline{:}} [\![\mathcal{R}_2]\!]_\Sigma \qquad [\![\mathcal{R}_1 \mid \mathcal{R}_2]\!]_\Sigma \rightarrow [\![\mathcal{R}_1]\!]_\Sigma \mathbin{\underline{\vee}} [\![\mathcal{R}_2]\!]_\Sigma$$
$$[\![\mathcal{R}^n]\!]_\Sigma \rightarrow [\![\mathcal{R}]\!]_\Sigma^{\underline{n}} \qquad [\![\mathcal{R}^*]\!]_\Sigma \rightarrow [\![\mathcal{R}]\!]_\Sigma^{\underline{*}}$$
$$[\![a^n]\!]_\Sigma \rightarrow \{\emptyset_{n\cdot a}\} \cup \bigcup_{e \in \Sigma \setminus \{a\}} \emptyset_{0\cdot e}$$

The transformation replaces operators from the regex domain into that of constraint expressions, and transforms alphabetic symbols into well-formed constraint vectors.

Example 5. $a^2b \xrightarrow{\{a,b\}} (\{\emptyset_{2\cdot a}\} \cup \{\emptyset_{0\cdot b}\}) \mathbin{\underline{:}} (\{\emptyset_{1\cdot b}\} \cup \{\emptyset_{0\cdot a}\}) \equiv \{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \mathbin{\underline{:}} \{\emptyset_{0\cdot a}, \emptyset_{1\cdot b}\}$

4.2 From Constraint Expressions to DCEs

As can be seen in Example 5, the constraint expression produced by $\xrightarrow{\Sigma}$ will not necessarily be a DCE (in this case, because it contains a concatenation operator), yet the constraint evaluation function described in Section 3.5 is only defined for DCEs. The remainder of this section defines the $\twoheadrightarrow$ operator, which must be repeatedly applied to a constraint expression until a DCE is obtained.

Definition 9 (Distribution of Concatenation over Disjunction). *We define $\twoheadrightarrow$ such that concatenation* distributes *over disjunctions of expressions:*

$$\hat{\mathbb{C}} \mathbin{\underline{:}} \left(\hat{\mathbb{C}}_1 \mathbin{\underline{\vee}} \hat{\mathbb{C}}_2 \mathbin{\underline{\vee}} \ldots \mathbin{\underline{\vee}} \hat{\mathbb{C}}_n\right) \twoheadrightarrow \left(\hat{\mathbb{C}} \mathbin{\underline{:}} \hat{\mathbb{C}}_1\right) \mathbin{\underline{\vee}} \left(\hat{\mathbb{C}} \mathbin{\underline{:}} \hat{\mathbb{C}}_2\right) \mathbin{\underline{\vee}} \ldots \mathbin{\underline{\vee}} \left(\hat{\mathbb{C}} \mathbin{\underline{:}} \hat{\mathbb{C}}_n\right)$$

Definition 10 (Concatenating Constraints). *Two constraint vectors can be concatenated by adding the corresponding basic constraints' offsets and modulo sets:*

$$\overrightarrow{\mathbb{C}}_1 \mathbin{\underline{:}} \overrightarrow{\mathbb{C}}_2 \twoheadrightarrow \left\{(m_1 \cup m_2)_{(n_1+n_2)\cdot a} \mid a \in \Sigma, m_{1\,n_1\cdot a} \in \overrightarrow{\mathbb{C}}_1, m_{2\,n_2\cdot a} \in \overrightarrow{\mathbb{C}}_2\right\}$$

Definition 11 (Bounded Repetition). *The reduction of expressions repeated for a fixed number of times is defined for a single constraint vector and a disjunction of constraint expressions, as follows:*

$$\overrightarrow{\mathbb{C}}^{\underline{k}} \twoheadrightarrow \left\{m_{(n\times k)\cdot a} \mid m_{n\cdot a} \in \overrightarrow{\mathbb{C}}\right\}$$
$$\left(\hat{\mathbb{C}}_1 \mathbin{\underline{\vee}} \hat{\mathbb{C}}_2 \mathbin{\underline{\vee}} \ldots \mathbin{\underline{\vee}} \hat{\mathbb{C}}_n\right)^{\underline{k}} \twoheadrightarrow \bigvee_{i_1+i_2+\ldots+i_n=k} \hat{\mathbb{C}}_1^{\underline{i_1}} \mathbin{\underline{:}} \hat{\mathbb{C}}_2^{\underline{i_2}} \mathbin{\underline{:}} \cdots \mathbin{\underline{:}} \hat{\mathbb{C}}_n^{\underline{i_n}}$$

Definition 12 (Unbounded Repetition). *The reduction of expressions within unbounded repetition is defined for a single constraint vector and a disjunction of constraint expressions, as follows:*

$$\left(\hat{\mathbb{C}}_1 \veebar \hat{\mathbb{C}}_2 \veebar \ldots \veebar \hat{\mathbb{C}}_n\right)^{*} \twoheadrightarrow \hat{\mathbb{C}}_1^{*} \dot{:} \hat{\mathbb{C}}_2^{*} \dot{:} \ \cdots \ \dot{:} \hat{\mathbb{C}}_n^{*}$$

$$\overrightarrow{\mathbb{C}}^{*} \twoheadrightarrow \left\{(m \cup \texttt{toMod}(n,k))_{n\cdot a} \mid m_{n\cdot a} \in \overrightarrow{\mathbb{C}} \wedge \exists m'_{n'\cdot a'} \in \overrightarrow{\mathbb{C}}.\, m' \neq \emptyset\right\} \veebar$$

$$\left\{\texttt{toMod}(n,k)_{0\cdot a} \mid m_{n\cdot a} \in \overrightarrow{\mathbb{C}}\right\} \qquad [k = \texttt{idx}()]$$

When defining $\twoheadrightarrow$ over expressions containing unbounded repetition, we assume the existence of a function `idx`() that returns a natural number which is unique for every single application of $\twoheadrightarrow$. The function `toMod`() is defined as:

$$\texttt{toMod}(n,k) \mapsto \emptyset \text{ if } n = 0; \left\{n^{\cdot k}\right\} \text{ otherwise}$$

4.3 Building the Constraint Automaton

Based on the previous definitions, we can now define a construction for transforming a finite-state automaton into a constraint automaton.

Definition 13 (Regular Expression to Constraint Expression). *Given a regular expression $\mathcal{R}$, $C_\Sigma(\mathcal{R})$ will return a DCE $\dot{\mathbb{C}}$ whose vectors are well-formed with respect to Σ, such that $\mathcal{R} \xrightarrow{\Sigma} \hat{\mathbb{C}} \twoheadrightarrow^{*} \dot{\mathbb{C}}$.*

Definition 14 (FSA to CA). *Given a finite-state automaton $\mathcal{D} \stackrel{def}{=} \langle Q, q_0, \Sigma, F, \Gamma\rangle$ with Q states, initial state $q_0 \in Q$, alphabet Σ, final states $F \subseteq Q$, and $\Gamma \subseteq Q \times \Sigma \times Q$, one can construct a constraint automaton $\mathcal{CA} \stackrel{def}{=} \langle Q, q_0, \Sigma, F, \Gamma'\rangle$, where*

$$\Gamma' \stackrel{def}{=} \left\{(q, \dot{\mathbb{C}}, q') \mid q, q' \in Q, \mathcal{R} = \texttt{regex}_{\mathcal{D}}(q, q'), \mathcal{R} \neq \perp, \dot{\mathbb{C}} = C_\Sigma(\mathcal{R})\right\}$$

The construction considers each pair of states, deriving the regular expressions and converting them into constraint expressions. Each state in $\mathcal{CA}$ will thus have a transition to every other state with the corresponding constraint expression, provided that a path between those states exists in $\mathcal{D}$.

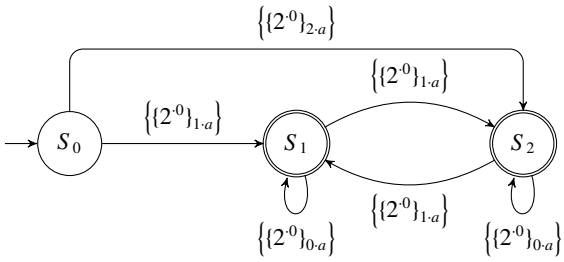


Fig. 6: Unambiguous constraint automaton for the flip-flop defined in Figure 4.

4.4 Examples

Example 6. The following example shows the derivation of a constraint from state S_0 to S_2 in Figure 7, which involves a repeated set of identical transitions, equivalent to the bounded iteration of a group of regular expressions related via concatenation. As with multinomial expansion, raising a DCE with m terms to a power n will result in a constraint expression of $\binom{n+m-1}{n}$ terms. In this example, the terms are reduced further, yet in general, bounded iteration will produce long DCEs.

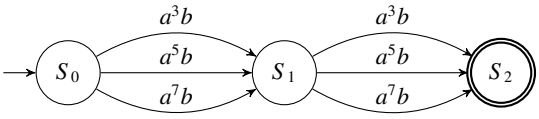
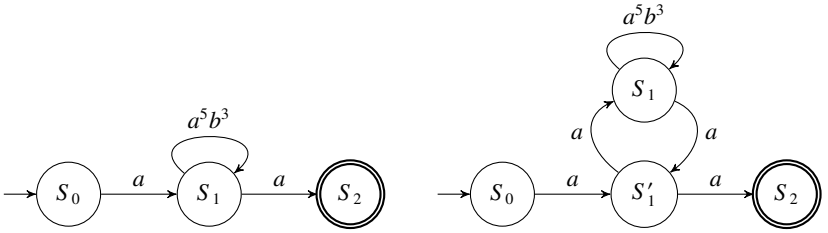


Fig. 7: Automaton for $(a^3b \mid a^5b \mid a^7b)^2$

$$
\begin{aligned}
& (a^3b \mid a^5b \mid a^7b)^2 && \text{Regex-to-CE, Bounded Repetition,} \\
\xrightarrow{\Sigma}\ & (\{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\})^{\underline{2}} && \text{Concatenation (8, 11, 10)} \\
\twoheadrightarrow\ & \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{2}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{2}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{2}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\vee}} \\
& \{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\}^{\underline{0}} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}^{\underline{1}} && \text{Bounded Repetition (11)} \\
\twoheadrightarrow\ & \{\emptyset_{14\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{10\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{6\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \\
& (\{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\}) \mathbin{\underline{\vee}} \\
& (\{\emptyset_{3\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}) \mathbin{\underline{\vee}} && \text{Bounded Repetition,} \\
& (\{\emptyset_{5\cdot a}, \emptyset_{1\cdot b}\} \mathbin{\underline{\cdot}} \{\emptyset_{7\cdot a}, \emptyset_{1\cdot b}\}) && \text{Power}^{\underline{0}} \text{ elimination (11)} \\
\twoheadrightarrow\ & \{\emptyset_{14\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{10\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{6\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \\
& \{\emptyset_{8\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{10\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{12\cdot a}, \emptyset_{2\cdot b}\} && \text{Concatenation (10)} \\
=\ & \{\emptyset_{14\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{10\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{6\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \\
& \{\emptyset_{8\cdot a}, \emptyset_{2\cdot b}\} \mathbin{\underline{\vee}} \{\emptyset_{12\cdot a}, \emptyset_{2\cdot b}\} && \text{Removal of duplicates}
\end{aligned}
$$

Example 7. The following example shows the derivation of a constraint from state S_0 to S_2 in Figure 8a, involving a loop sandwiched between two compulsory single-element transitions, thus demonstrating the use of modulo sets.

$$
\begin{aligned}
& a(a^5b^3)^*a && \\
\xrightarrow{\Sigma}\ & \{\emptyset_{1\cdot a}, \emptyset_{0\cdot b}\} \mathbin{\underline{\cdot}} \{\emptyset_{5\cdot a}, \emptyset_{3\cdot b}\}^{\underline{*}} \mathbin{\underline{\cdot}} \{\emptyset_{1\cdot a}, \emptyset_{0\cdot b}\} && \text{Regex-to-CE (8)} \\
\twoheadrightarrow\ & \{\emptyset_{1\cdot a}, \emptyset_{0\cdot b}\} \mathbin{\underline{\cdot}} \left\{\left\{5^{\cdot 0}\right\}_{0\cdot a}, \left\{3^{\cdot 0}\right\}_{0\cdot b}\right\} \mathbin{\underline{\cdot}} \{\emptyset_{1\cdot a}, \emptyset_{0\cdot b}\} && \text{Unbounded Repetition (12)} \\
\twoheadrightarrow\ & \left\{\left\{5^{\cdot 0}\right\}_{2\cdot a}, \left\{3^{\cdot 0}\right\}_{0\cdot b}\right\} && \text{Concatenation (10)}
\end{aligned}
$$



(a) Automaton for $a(a^5b^3)^*a$ (b) Automaton for $a(a(a^5b^3)^*a)^*a$

Fig. 8: Example automata showing loops and nested repetition

Example 8. The final example shows the derivation of a constraint from state S_0 to S_2 in Figure 8b, which showcases a nested repetition, demonstrating the effect of unbounded iteration on non-empty modulo sets.

$a(a(a^5b^3)^*a)^*a$

$$\xrightarrow{\Sigma} \{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \mathbin{:} (\{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \mathbin{:} \{\emptyset_{5\cdot a}, \emptyset_{3\cdot b}\}^{*}_{-})^{*}_{-} \qquad \text{Regex-to-CE, Catenation (8, 10)}$$

$$\twoheadrightarrow \{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \mathbin{:} (\{\{5\}_{2\cdot a}, \{3\}_{0\cdot b}\})^{*}_{-} \qquad \text{Result of Example 7}$$

$$\twoheadrightarrow \{\emptyset_{2\cdot a}, \emptyset_{0\cdot b}\} \mathbin{:} \left(\left\{\left\{2^{\cdot 1}\right\}_{0\cdot a}, \emptyset_{0\cdot b}\right\} \veebar \left\{\left\{2^{\cdot 1}, 5^{\cdot 0}\right\}_{2\cdot a}, \left\{3^{\cdot 0}\right\}_{0\cdot b}\right\}\right) \qquad \text{Unbounded Repetition (12)}$$

$$\twoheadrightarrow \left\{\left\{2^{\cdot 1}\right\}_{2\cdot a}, \emptyset_{0\cdot b}\right\} \veebar \left\{\left\{2^{\cdot 1}, 5^{\cdot 0}\right\}_{4\cdot a}, \left\{3^{\cdot 0}\right\}_{0\cdot b}\right\} \qquad \text{Concatenation (10)}$$

5 Computational Complexity

The purpose of constraint automata is to determine the precise set of next states for unordered input traces *efficiently*. Thus, it is important to analyze the computational cost of using the involved structures.

Consider the conversion of an input automaton $\mathcal{D}$, with states Q and an alphabet Σ, into a constraint automaton $\mathcal{CA}$. The size of the resultant constraint automaton is influenced by three factors, namely (i) the *connectivity* of $\mathcal{D}$, with a fully connected automaton leading to an out-degree of $|Q|$ for each state in $\mathcal{CA}$, thus requiring a maximum of $|Q|$ constraint expressions to be evaluated with each step in the constraint automaton, (ii) the number of *choice operators* in the regular expressions derived from the automaton, which affects the number of constraint vectors in the derived constraint expressions, and (iii) the number of *cycles* in the regular expressions, which will cause basic constraints' modulo-set sizes to grow.

A sparsely-connected input automaton would tend to have fewer outgoing transitions per state (as fewer states would be reachable from other states), whereas a densely-connected automaton containing many loops of differing length would increase the size of constraints' modulo sets. As the constraint vectors in the automaton must be well-formed, they will each contain $|\Sigma|$ basic constraints.

Furthermore, as the current state is a set of possible states, the set of next states would have to be computed whilst taking each current state into consideration. The size of the current state set can be at most $|Q|$, which would only occur when the automaton

is potentially in any state. Hence, the number of operations performed when computing the next state, assuming the worst-case scenario of a fully-connected aggregate automaton and a full current state set, is:

$$\underbrace{|Q|}_{\substack{\text{current}\\\text{states}}} \times \underbrace{|Q|}_{\substack{\text{constraint}\\\text{expressions}}} \times \underbrace{vecs}_{\substack{\text{vectors}\\\text{/CE}}} \times \underbrace{|\Sigma|}_{\substack{\text{basic}\\\text{constraints}\\\text{/CV}}} \times \left(\underbrace{mods!}_{\substack{\text{modulo}\\\text{set size}\\\text{/BC}}} + \underbrace{1}_{\substack{\text{offset}\\\text{comparison}}} \right)$$

As noted earlier, the magnitude of *mods* and *vecs* is dependent on the form of the extracted regular expressions, specifically the number of cycles and choice operators, respectively. Evaluating the modulo set requires finding solutions to linear equations, which has a worst-case running time depending on the input and number of terms. As will be discussed in Section 6, in practice, one can lower the average running time by ordering the evaluation of constraints based on their weakness, and by using more sophisticated techniques for solving equations.

Implementation Considerations

The use of a set of current states could be eliminated by making the automaton deterministic, yet this generally results in an exponential growth in states, increasing the number of constraint expressions to be evaluated at every state by an equivalent degree. By maintaining a dynamic set of current states, one can thus reduce the average traversal time, as only the outgoing transitions from potential current states are evaluated.

In this work, we have opted to use regular expressions to produce an initial constraint automaton so as to modularize the transformation stages. It is possible that some performance gains may be obtained by generating constraint expressions directly from the original automaton. More specifically, this may allow the detection of subexpressions that are shared across constraints, facilitating the caching and reuse of partial results during constraint evaluation. Optimizations could also extract common subexpressions among constraint expressions emanating from states, rather than evaluating each outgoing transition in isolation. Such considerations could give rise to interesting future work.

A system implementing constraint automata would most likely benefit from changing the representation from the one used into one that is more amenable to comparisons. For example, constraints could be organized in a tree structure based on their offset values, speeding up evaluation by excluding branches which do not meet the minimum.

6 Related Work

Instrumentation is recognized as a source of overhead in runtime verification. This overhead can be reduced by decreasing the amount of instrumentation, or *sampling*, that is performed. Statistical methods can then be employed to infer the most probable sequence of state transitions that occurred during the time in which sampling was suspended. For instance, Stoller et al. [12] consider the scenario where instrumentation is suspended for some period of time, leaving a gap in the sequence of observed events.

Their approach focused on reconstructing the missed events via probabilistic models, which estimate the next state based on traces that were observed previously. This approach differs from that explored in this work, as the gaps are devoid of information, not even specifying the number of missed events. In contrast, our work only considers unordered traces, and still requires that the number and type of events be logged.

Bodden et al. [5,6] provide an implementation of efficient time-triggered automata, which consider gaps of events during monitoring. The approach explored can report false positives (but not false negatives) if continuously monitoring "skip" events that prevent an error state from being reached.

Bartocci et al. [2] extend the concept of probabilistically monitoring gaps in events, and introduce the notion of *criticality levels*, which vary based on the probability that a system reaches an error state. Criticality levels can then be used to determine the degree of instrumentation performed, with the system increasing sampling to determine the precise system state. A similar concept could be integrated into the construction examined in this work. For example, sampling could be increased on detecting that the current state set contains a final state.

Another approach, adopted by Basin et al. [3], is to handle the uncertainty brought about by incomplete traces using a three-valued logic, whereby the property's evaluation function is modified to also reason about indeterminate results.

Bauer et al. [4] present a multi-valued logic that is able to express not only whether a violation has taken place, but also whether a violation would occur if the trace terminated right now. One could easily combine such acceptance conditions with our approach.

The choice of algorithm when generating regular expressions from a finite-state automaton will affect the size and complexity of the resultant expressions [10]. Bounded iteration with choice produces constraint expressions with multiple constraint vectors. In broad terms, unbounded iteration will cause the offset value to be added to the modulo set. Subsequent nesting of an iterated expression within unbounded iterations with no offset will have no further effect on the constraint expression's size. Ideally, the algorithm employed would thus minimize the number of choice operators in the output expressions. By isolating the inefficient component of the constraints into modulo sets, one may choose to apply existing results and libraries addressing the *Satisfiability Modulo Theories* problem [1] to speed up the computation.

7 Conclusion

We have presented a trace model that allows for the monitoring of distributed systems by compressing partial event streams before they are sent to the monitoring server. We described an algorithm for constructing ahead-of-time a monitor that can deal with compressed event streams in such a way that it provably recognizes property violations without false negatives. We have further shown that the resulting automaton is as precise as possible, and has a complexity low enough to promise performance gains in practice.

References

1. Barrett, C., Stump, A., Tinelli, C.: The satisfiability modulo theories library (smt-lib) (Apr 2013), http://smtlib.org/
2. Bartocci, E., Grosu, R., Karmarkar, A., Smolka, S., Stoller, S., Zadok, E., Seyster, J.: Adaptive runtime verification. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification, Lecture Notes in Computer Science, vol. 7687, pp. 168–182. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-35632-2_18
3. Basin, D., Klaedtke, F., Marinovic, S., Zlinescu, E.: Monitoring compliance policies over incomplete and disagreeing logs. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification, Lecture Notes in Computer Science, vol. 7687, pp. 151–167. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-35632-2_17
4. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In: Sokolsky, O., TaÅran, S. (eds.) Runtime Verification, Lecture Notes in Computer Science, vol. 4839, pp. 126–138. Springer Berlin Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-77395-5_11
5. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. In: 7th workshop on Runtime Verification at the 6th International Conference on Aspect-Oriented Software Development, Vancouver, Canada. LNCS, vol. 4839, pp. 22–37. Springer (Mar 2007), http://www.bodden.de/pubs/bhl+07collaborative.pdf
6. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative runtime verification with tracematches. Oxford Journal of Logics and Computation (Nov 2008), http://www.bodden.de/pubs/bhl+08collaborative.pdf
7. Brzozowski, J.A.: Derivatives of regular expressions. vol. 11, pp. 481–494. ACM, New York, NY, USA (Oct 1964), http://doi.acm.org/10.1145/321239.321249
8. Chen, F., Roşu, G.: Mop: an efficient and generic runtime verification framework. In: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications. pp. 569–588. OOPSLA '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1297027.1297069
9. Miller, D., Pearson, B.: Security information and event management (SIEM) implementation. McGraw-Hill (2011)
10. Neumann, C.: Converting deterministic finite automata to regular expressions (Mar 2005), http://neumannhaus.com/christoph/papers/2005-03-16.DFA_to_RegEx.pdf
11. Steffens, S.: P3 consulting, personal communication. http://www.p3-consulting.de/
12. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime verification with state estimation. In: Proceedings of the Second international conference on Runtime verification. pp. 193–207. RV'11, Springer-Verlag, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-29860-8_15

A Proofs

Theorem 1 (Convergence). *Given an arbitrary finite state automaton $\mathcal{D}$ with Q states, the transformation will always converge onto a constraint automaton whose constraint expressions are all DCEs.*

Proof. By definition, the function `regex` returns a regular expression of bounded size. As Q is finite, and `regex` is computed pairwise for each state, the constraint automaton

will contain at most $|Q|^2$ transitions. The task is thus to show that the conversion of regular expressions into constraints always converges onto a DCE.

A regular expression $\mathcal{R}$ with N atoms of the form a^k, P power operators, and C concatenation operators, will produce an initial constraint expression $\hat{\mathbb{C}}$ under $\xrightarrow{\Sigma}$ with an equal number of P and C operators in the constraint expression domain, and each of the N atoms will be replaced with a constraint vector. The number of disjunctions in the expression is not directly relevant for the purposes of convergence.

The transformation $\twoheadrightarrow$ is defined for all constraint expression operators, only stopping once a constraint expression is a DCE. In addition, $\twoheadrightarrow$ will itself produce a constraint expression, thus showing that the system will always progress while there are operators left to be reduced. The next step is thus to demonstrate that repeated applications of $\twoheadrightarrow$ will reduce P and C to zero. This is done by case analysis on each constituent definition of the operator, as follows:

- **Concatenation** of two vectors will reduce C by 1.
- **Distributing** $\underline{\cdot}$ **over** $\underline{\vee}$ has the immediate effect of increasing C by the number of disjuncts, yet the composite expressions will subsequently undergo concatenation, giving an overall reduction in C of 1.
- **Bounded Repetition** on constraint vectors will reduce P. Bounded repetition on a disjunctions of constraint expressions will itself produce a disjunction of constraint expressions. The index is transferred to the individual, simpler disjuncts. These disjuncts would either consist of constraint vectors, in which case P would be reduced, or further disjunctions. Yet in the latter case, the nesting depth must be finite (due to the expression being finite).
- **Unbounded Repetition** of a constraint vector leads to an immediate reduction of 1 from P. For a disjunction of constraint expressions, the transformation will result in each disjunct being raised to an unbounded power, and hence an initial increase in P and C by the number of disjuncts. Yet if the sub-terms are constraint vectors, then P and C will be reduced, as per the previous definitions. If, alternatively, the terms are themselves disjunctions of expressions, then the terms would be further expanded until concatenations of basic vectors are reached, at which point P and C will be reduced.

Theorem 2 (Equal Aggregate Event Vectors). *Given that Perms(s) returns the set of permutations of a trace s, if* $s \in \Sigma^*$*, then* $\forall p \in Perms(s).\ \downarrow\#^p_\Sigma = \downarrow\#^s_\Sigma$

Proof. As the alphabet Σ is fixed for all the permutations of s, the function will always return a set containing Σ maps, with one for each element of Σ. Since the permutation operation only affects the order of symbols within a trace, the symbol counts remain unaffected.

Theorem 3 (Equivalent Evaluations of Regex and Constraint Expressions). *The constraint automaton cannot produce false negatives, and is also maximally precise given the reduced information it receives, i.e., won't produce more false warnings than a solution based on the explicit automaton traversal using all string permutations. It holds that* $\forall s \in \Sigma^*.\ \left[\texttt{eval}_{\downarrow\#^s_\Sigma}(C_\Sigma(\mathcal{R})) \leftrightarrow (\exists p \in Perms(s).\ \texttt{match}(p, \mathcal{R}))\right]$

Each direction of the bi-implication is proven separately, and is presented as two proofs. Prior to the proofs, we state the following lemmas:

Lemma 1 (Permutations of a regular expression). *Given that Perms($\mathcal{R}$) returns all permutations of the sub-expressions of a regular expression $\mathcal{R}$ such that any sub-expressions $\mathcal{R}_1$ and $\mathcal{R}_2$ can be reordered using $\mathcal{R}_1\mathcal{R}_2 \equiv \mathcal{R}_2\mathcal{R}_1$ and $\mathcal{R}_1|\mathcal{R}_2 \equiv \mathcal{R}_2|\mathcal{R}_1$, it holds that $\exists p \in Perms(s).\, \texttt{match}(p, \mathcal{R}) \leftrightarrow \exists p \in Perms(s).\, \exists r \in Perms(R).\, \texttt{match}(p, r)$*

Lemma 2 (Regex decomposition). *Decomposed strings match sub-expressions, that is, $\forall s \in \Sigma^*.\, \texttt{match}(s, \mathcal{R}_1\,\mathcal{R}_2) \rightarrow \exists p_1 p_2 = s.\, \texttt{match}(p_1, \mathcal{R}_1) \wedge \texttt{match}(p_2, \mathcal{R}_2)$, and $\forall s \in \Sigma^*.\, \texttt{match}(s, \mathcal{R}_1|\mathcal{R}_2) \rightarrow \texttt{match}(s, \mathcal{R}_1) \vee \texttt{match}(s, \mathcal{R}_2)$*

Proof (Evaluation → Match). Every basic constraint, constraint vector and DCE can be reconstructed into a regular expression, as follows:

$$\texttt{Rec}(\{m_1, m_2, \ldots, m_k\}_{n\cdot a}) = a^n(a^{m_1^*}a^{m_2^*}\ldots a^{m_k^*})^*$$
$$\texttt{Rec}(\{\mathbb{C}_1, \mathbb{C}_2, \ldots, \mathbb{C}_k\}) = \texttt{Rec}(\mathbb{C}_1)\ \texttt{Rec}(\mathbb{C}_2)\ \ldots\ \texttt{Rec}(\mathbb{C}_k)$$
$$\texttt{Rec}(\overrightarrow{\mathbb{C}}_1 \veebar \overrightarrow{\mathbb{C}}_2 \veebar \ldots \veebar \overrightarrow{\mathbb{C}}_k) = \texttt{Rec}(\overrightarrow{\mathbb{C}}_1) \mid \texttt{Rec}(\overrightarrow{\mathbb{C}}_2) \mid\ \ldots\ \mid \texttt{Rec}(\overrightarrow{\mathbb{C}}_k)$$

If $\dot{\mathbb{C}}$ is a DCE which holds for $\downarrow\#^s_\Sigma$, then one can rebuild a string $a^{\#^s_a}b^{\#^s_b}\ldots z^{\#^s_i}$, for $a, b \ldots z \in \Sigma$, that will contain a symbol for each element in Σ with a frequency equal to its value in the aggregated input. Given the definition of Rec(), it follows that $\exists p \in Perms(a^{\#^s_a}b^{\#^s_b}\ldots z^{\#^s_i}).\, \texttt{match}(p, \texttt{Rec}(\dot{\mathbb{C}}))$. By using Lemma 1, the fact that $Perms(a^{\#^s_a}b^{\#^s_b}\ldots z^{\#^s_i}) = Perms(s)$, and by substituting $\dot{\mathbb{C}}$ with $C_\Sigma(\mathcal{R})$, the statement to be proven can be reformulated as:

$$\exists p \in Perms(s).\, \texttt{match}(p, \texttt{Rec}(C_\Sigma(\mathcal{R}))) \rightarrow \exists p \in Perms(s).\, \exists r \in Perms(\mathcal{R}).\, \texttt{match}(p, r)$$

The relation will be demonstrated through case analysis on the different forms of regular expressions. Given that Lemma 2 holds, it is sufficient to show that the operators hold on the basic elements of regular expressions and then induce on the length of the regular expression.

$\mathcal{R}$	$C_\Sigma(\mathcal{R})$	Rec($C_\Sigma(\mathcal{R})$)	
a^nb^m	$\{\emptyset_{n\cdot a}, \emptyset_{m\cdot b}\}$	a^nb^m	$\in Perms(R)$
$a^n\|b^m$	$\{\emptyset_{n\cdot a}, \emptyset_{0\cdot b}\} \veebar \{\emptyset_{0\cdot a}, \emptyset_{m\cdot b}\}$	$a^n\|b^m$	$\in Perms(R)$
a^{n^*}	$\{\{n^{\cdot z}\}_{0\cdot a}\}$	$a^0a^{n^*}$	$\in Perms(R)$
$(a^nb^m)^*$	$\{\{n^{\cdot z_1}\}_{0\cdot a}, \{m^{\cdot z_1}\}_{0\cdot b}\}$	$(a^nb^m)^*$	$\in Perms(R)$
$(a^n\|b^m)^*$	$\{\{n^{\cdot z_1}\}_{0\cdot a}, \{m^{\cdot z_2}\}_{0\cdot b}\}$	$(a^n)^*(b^m)^*$	Matches $\mathcal{R}$ for permutation of s
$(a^{n^*}\|b^{m^*})^k$	$\{\{n^{\cdot z_1}\}_{0\cdot a}, \{m^{\cdot z_2}\}_{0\cdot b}\}$	$(a^n)^*(b^m)^*$	Matches $\mathcal{R}$ for permutation of s

Proof (Match → Evaluation). Recall that $C_\Sigma()$ is evaluated in two phases, first transforming the input into an initial constraint expression (Definition 8), and then iteratively reducing the expression into a DCE. The initial transformation simply changes the operators into those of the constraint-expression domain, whilst replacing alphabetic symbols into basic constraints. It is evident that $\forall a \in \Sigma.\, s \in \Sigma^*.\, \forall n \in \mathbb{N}.\, \texttt{match}(s, a^n) \leftrightarrow$

$\texttt{eval}_{\downarrow \#^s_{\vec{\Sigma}}}(\emptyset_{n \cdot a})$, since the LHS will only be true for sequences of as of length n, which also holds for the RHS. The task is thus to demonstrate that expressions obtained by reductions using $\twoheadrightarrow$ will also evaluate to true. This is done via case analysis on the operators, as follows:

- **Concatenation** of two constraint vectors is performed by summing the offsets and performing a union of modulo sets for every basic constraint. The result will thus consume at least the same amount of input events as the constituent vectors would in isolation.
- **Distributing** $\underline{\cdot}$ **over** $\underline{\vee}$ preserves truth due to the design of the `eval()` procedure, as each disjunct will also include the concatenated outer disjunct.
- **Bounded Repetition** of a constraint vector $\overrightarrow{\mathbb{C}}$ raised to a power k is equivalent to concatenating a sequence of k consecutive $\overrightarrow{\mathbb{C}}$ vectors. The modulo sets thus remains unchanged, whilst the offsets of each basic constraint are multiplied by a factor k. Bounded repetition of a disjunction of constraint is performed by expanding the disjunction as a *multinomial*, discarding the coefficients. This produces a disjunction of sequences with every combination of the terms, essentially unravelling the loop.
- **Unbounded Repetition** resolves to a concatenation of zero or more constraint expressions. If a single constraint vector is being raised to a power, then its offset is added to its modulo set, thus matching when the original expression is taken one or more times. In the case of an unbounded repetition of a DCE, each disjunct can be taken an unbounded number of times, forming a sequence of constraint expressions. Equivalent sequences can be formed by changing the disjunctions into concatenations, whilst simultaneously placing each concatenated expression in an unbounded loop.