Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 113 (2005) 123–143
www.elsevier.com/locate/entcs

Simulation of Simultaneous Events in Regular Expressions for Run-Time Verification

Usa Sammapun Arvind Easwaran Insup Lee Oleg Sokolsky[1,2]

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA

Abstract

When specifying system requirements, we want a language that can express the requirements in the simplest and most intuitive form. Although the MaC system provides an expressive language, called MEDL, it is generally awkward to express certain features like temporal ordering of complex events, timing constraints, and frequencies of events which are inherent in safety properties. MEDL-RE extends the MEDL language to include regular expressions to easily specify timing dependencies and timing constraints. Due to simultaneous events generated by the MaC system, monitoring regular expressions by simulating DFAs would result in a potential problem. The DFA simulations would involve concurrent multi-path simulations and result in exponential running time. To handle simultaneous events inexpensively, we generate a dependency graph to identify possible simultaneous events. Further, we augment the original DFAs with alternative transitions, which will substitute for multi-path simulations.

Keywords: Runtime verification, DFA simulation, DFA, MEDL.

1 Introduction

The monitoring, checking and steering (MaC) framework [9,10,11] has been designed to ensure that the execution of a real-time system is consistent with its requirements at run-time. It provides a language, called MEDL, to specify safety properties based on LTL [13]. The safety properties include both

[1] This research was supported in part by NSF CCR-9988409, NSF CCR-0086147, NSF CCR-0209024, and ARO DAAD19-01-1-0473.
[2] Email: {usa,arvinde,lee,sokolsky}@saul.cis.upenn.edu

doi:10.1016/j.entcs.2004.01.030

computational and timing requirements. The safety properties are defined in terms of events, conditions, auxiliary variables, and auxiliary functions. Events are instantaneous incidents such as variable updates or the start/end of a method call. Conditions are propositions about the program that may be true or false for a duration of time. Those events and conditions can also be composed using connectives described in Section 2. Auxiliary variables are temporary storage, which allows us, for example, to count the number of occurrences of an event. Auxiliary functions return values and time stamps of events. The MEDL language provides an elegant and intuitive way to specify computational requirements. It, however, does not provide an intuitive way to specify timing requirements, such as temporal ordering of events with complex timing dependencies, timing constraints or counting of specific events in a time interval.

The extension of MEDL called MEDL-RE [14] adds the ability to specify ordering of events in the form of regular expressions (RE) over a customized set of events, which offers users with clearer and less error-prone specifications. In this paper, we propose an efficient simulation of the corresponding DFAs at runtime. By observing a sequence of events occuring in a target system, a DFA generated by MEDL-RE matches the sequence of events with a specific regular expression. However, because the composite events can be triggered simultaneously and cannot be temporally ordered in any way, the DFA must recognize these events for any ordering of the events. This means if a regular expression has some inherent ordering of simultaneous events, the DFA must accept all different permutations of such order. We refer to those permutations as *linearizations*. To build such a DFA, we augment the original DFA with alternative transitions to provide paths from one linearization to another. We then prove that the original DFA and the augmented DFA are equivalent. Only those DFAs whose underlying regular expressions have candidate simultaneous events in their relevant sets are augmented. The candidate simultaneous events can be statically detected by building and traversing a dependency graph described in Section 4.

The paper is organized as follows. Section 2 briefly explains an overview of the MaC framework. Section 3 introduces an extension MEDL-RE. Section 4 discusses the construction of the dependency graph. Section 5 presents and proves our augmented DFA algorithm. Section 6 presents related work. Lastly, section 7 concludes the paper.

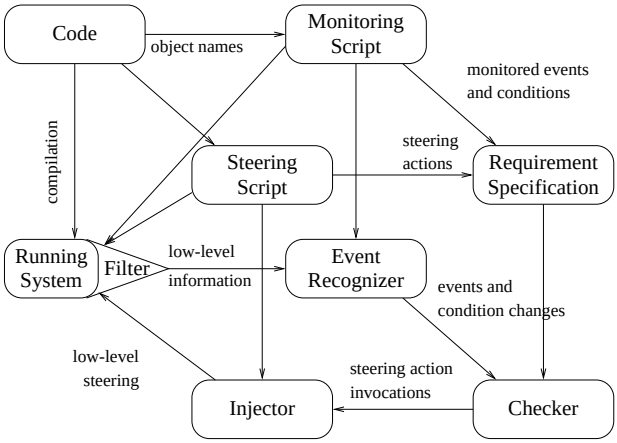

Fig. 1. Overview of the MaC architecture

2 MaC Overview

2.1 MaC Architecture

The MaC system has been developed to ensure that a program runs correctly with respect to its formal requirement. Fig. 1 shows the MaC architecture. The system works as follows. A user specifies a requirement of a target program in a formal language. Given a target program and the requirement, the MaC system inserts a collection of probes or a *filter* into the target program. During run-time, the execution of the probed target program is monitored and checked by the MaC system. An *event recognizer* detects primitive events and conditions from state information received from the *filter*. The primitive events are change of value (`update(object)`), entering method (`startM(method)`), and leaving method (`endM(method)`). The primitive conditions are boolean variables or boolean statements composed by primitive typed variables in the target program. These events and conditions are then sent to a *run-time checker*, which determines whether or not the current execution history satisfies the requirement specification. The execution history is captured from a sequence of events sent by the event recognizer. If the run-time checker detects any violation, it notifies the user and triggers an *injector* to take a steering action specified in a steering script and steer the target program back to a safe state.

2.2 MaC Languages

The MaC system provides three languages. The requirement specification or the Meta-Event Definition Language (MEDL), based on an extension of a

linear temporal logic (LTL) [13], allows us to express a large subset of safety properties of systems, including real-time properties. The monitoring script, expressed in the Primitive Event Definition Language (PEDL), is used to define what information is sent from a filter to the MaC system and how it is transformed into events and conditions used in MEDL. The steering script written in the Steering Action Definition Language (SADL) is used to specify actions to be invoked when violations occur. See [9,10] for PEDL and SADL details.

2.2.1 Events and Conditions

Events occur instantaneously during the system execution, whereas *conditions* represent information that hold for a duration of time. For example, an event denoting the call to method `init` occurs at the instant the control is passed to the method, while a condition `(angle < 30)` holds as long as the value of the variable `angle` does not exceed `30`. The syntax of events and conditions is shown below.

$$E ::= e \mid \texttt{start}(C) \mid \texttt{end}(C) \mid E \;\&\&\; E \mid E \mid\mid E \mid E \;\texttt{when}\; C$$
$$C ::= c \mid \texttt{defined}(C) \mid \texttt{[}E\texttt{,}E\texttt{)} \mid \texttt{!}C \mid C \;\&\&\; C \mid C \mid\mid C \mid C \Rightarrow C$$

The boolean connectives used in events and conditions are defined in the usual way. Events $\texttt{start}(c)$ and $\texttt{end}(c)$ are triggered when c becomes true and false, respectively. An event e `when` c is triggered when e is triggered and c is true. A condition $\texttt{defined}(c)$ is true when c is defined. A condition $[e_1, e_2)$ is true between the occurrence of events e_1 and e_2 where e_1 is included but e_2 is not. For formal semantics of events and conditions, see [9,11].

2.2.2 Meta Event Definition Language (MEDL)

MEDL includes events and conditions imported from PEDL, definitions of composite events and conditions, safety properties, auxiliary variables, and auxiliary functions. A safety property can be expressed as a condition or as an event called an alarm. A safety property condition must *always* be true during the execution whereas an alarm must *never* be raised. Auxiliary variables can be used to define events and conditions. Auxiliary variables allow us, for example, to count the number of occurrences of an event. Auxiliary functions are $\texttt{time}(e)$ and $\texttt{value}(e)$, which return the time stamp and the value of an event e, respectively.

3 Syntax and Semantics of MEDL-RE

This section describes the extension of the MEDL language to include the specification of the ordering of events by expressing them as a regular expres-

$$\begin{aligned}\Sigma_E &= \{\mathrm{E}\}\\ \Sigma_{R_1+R_2} &= \Sigma_{R_1} \cup \Sigma_{R_2}\\ \Sigma_{R_1\ .\ R_2} &= \Sigma_{R_1} \cup \Sigma_{R_2}\\ \Sigma_{R^*} &= \Sigma_R\end{aligned}$$

Fig. 2. A function Σ_R, which returns the relevant event set of R

sion and a frequency of events in a time interval.

3.1 Syntax

Let $\mathcal{R}$ be a set of regular expression names, e.g., R, R_1, R_2, etc. Let s be statements in the MEDL-RE, s_{MEDL} be the existing MEDL statements, $\bar{E}$ be a list of events, r be regular expressions, and C be the conditions described in Section 2.2.1. We define the syntax of the MEDL-RE extension as follows.

s	::= s_{MEDL}	Existing MEDL statements
	| `RE` R `{` $\bar{E}$ `} = <` r `>`	New MEDL-RE statement
r	::= E | r . r | $r + r$ | $r*$	Regular expressions
E	::= e | `start(`C`)` | `end(`C`)`	Existing events
	| E && E | E || E | E `when` C	
	| `startRE(`r`)` | `success(`r`)` | `fail(`r`)`	Regular expression events
f	::= `time(`E`)` | `value(`E`)` | `occur(`E, C`)`	Auxiliary functions

The MEDL-RE extension includes regular expressions, events `startRE`(r), `success`(r), `fail`(r), and an auxiliary function `occur`(E, C). The regular expressions, ranged over a set Σ of events, consists of events (E), concatenation $(r\ .\ r)$, union $(r+r)$, and Kleene star $(r*)$. The event `startRE`(r) indicates that we start observing the regular expression. The event `success`(r) indicates that we have found a sequence of events that specifies the regular expression, and the event `fail`(r) indicates that we have started observing but failed to finish finding such a sequence. For an event E and a condition C, the auxiliary function `occur`(E, C) returns a frequency or a number of occurrences of an event E during the time interval when C holds true.

3.2 Semantics

Let Σ_R in Fig. 2 denote a set of events specified in a RE R and Σ_V denote a customized set of events specified as $\bar{E}$ in the new MEDL-RE statement shown in the syntax section. Then, a relevant event set of R is $\Sigma = \Sigma_R \cup \Sigma_V$. We modify the model M defined in [9] as follows. A model M is a tuple (S, τ, L_C, L_E, o), where $S = \{s_0, s_1, \ldots\}$ is a set of states, τ is a mapping from S to the time domain, L_C is a total function from $S \times \mathcal{C}$ to $\{true, false, \Lambda\}$ where $\mathcal{C}$ denotes a set of condition names and Λ denotes undefined, and L_E is a partial function from $S \times \mathcal{E}$ to a value domain where $\mathcal{E}$ denotes a set of

$$
\begin{array}{lll}
D_a(a) & = \epsilon & \\
D_a(b) & = \Lambda & \\
D_a(\Lambda) & = \Lambda & \\
D_a(\epsilon) & = \Lambda & \\
D_a(R_1 + R_2) & = D_a(R_1) + D_a(R_2) & \\
D_a(R*) & = (D_a(R)) \ . \ R^* & \\
D_a(R_1 \ . \ R_2) & = (D_a(R_1)) \ . \ R_2 & \text{if } E(R) = \text{False} \\
 & \quad (D_a(R_1)) \ . \ R_2 + D_a(R_2) & \text{if } E(R) = \text{True}
\end{array}
$$

Fig. 3. A derivative of a regular expression R with respect to a ($D_a(R)$)

$$
\begin{array}{ll}
E(a) & = \text{False} \\
E(\Lambda) & = \text{False} \\
E(\epsilon) & = \text{True} \\
E(R_1 + R_2) & = E(R_1) \vee E(R_2) \\
E(R_1 \ . \ R_2) & = E(R_1) \wedge E(R_2) \\
E(R^*) & = \text{True}
\end{array}
$$

Fig. 4. A function $E(R)$, which tests whether $\epsilon \in R$

event names. For all e_k where $L_E(s_i, e_k)$ is defined, there is an order $o(s_i, e_k)$ such that at time $\tau(s_i)$, an order for each occurrence e_k is distinct. The order o is a total and injective function that maps e_k and s_i to an ordered set of positive integers and $o(s_{i-1}, e_k) < o(s_i, e_l)$ for all i and any k, l.

The semantics of MEDL-RE is defined using a derivative of a RE [5]. For any RE R and any alphabet a, a *derivative of R with respect to a*, denoted by $D_a(R)$, is the RE, where $D_a(R) = \{x \in \Sigma^* | ax \in R\}$. Fig. 3 and Fig. 4 show the semantics of a derivative of a RE and the function $E(R)$ from Σ^* to *boolean*, which tests whether $\epsilon \in R$, respectively, where R_1, R_2 are regular expressions, $a, b \in \Sigma$ and $a \neq b$.

Besides the derivatives, we also need to define a function $\texttt{FIRST}(R)$ and a function $\Phi^o_M(R)$. $\texttt{FIRST}(R)$ returns a set containing all events that can appear as the first event in the RE R. Formally, $\texttt{FIRST}(R) = \{a \in \Sigma \cup \{\epsilon\} \mid ax \in R \text{ where } x \in \Sigma^*\}$. $\Phi^o_M(R)$ represents the remainder of the RE R at an order o after a sequence of derivatives. We define $\Phi^o_M(R)$ as follows.

$$
\begin{array}{ll}
\Phi_M^{o(s_i,e)}(R) = R & \text{if } M, \tau(s_i) \models e \text{ where } e \in \texttt{FIRST}(R) \\
\Phi_M^{o(s_i,e)}(R) = D_e(\Phi_M^{o(s_i,e)-1}(R)) & \text{if } M, \tau(s_i) \models e
\end{array}
$$

We also define a language $\mathcal{L}$ (R) of R as follows.

$$
\begin{array}{ll}
\mathcal{L}(\emptyset) = \emptyset & \mathcal{L}(\epsilon) = \{\epsilon\} \\
\mathcal{L}(a) = \{a\} & \mathcal{L}(R_1 + R_2) = \mathcal{L}(R_1) \cup \mathcal{L}(R_2) \\
\mathcal{L}(R_1.R_2) = \{x_1.x_2 | x_1 \in \mathcal{L}(R_1) \wedge x_2 \in \mathcal{L}(R_2)\} & \mathcal{L}(R^*) = (\mathcal{L}(R))^*
\end{array}
$$

Using $\texttt{FIRST}(R)$ and $\Phi^o_M(R)$, we define the $\texttt{startRE}(R)$ event, $\texttt{success}(R)$ event, and the $\texttt{fail}(R)$ event.

$$
\begin{array}{lll}
M, t & \models & \texttt{startRE}(R) \text{ iff } M, t \models e \text{ where } e \in \texttt{FIRST}(R) \\
M, t & \models & \texttt{success}(R) \text{ iff } M, t \models e \text{ and } \epsilon \in \texttt{FIRST}(D_e(\Phi_M^{o(s_i,e)-1}(R))) \text{ where t} = \tau(s_i) \\
M, t & \models & \texttt{fail}(R) \quad \text{iff } M, t \models e \text{ and } D_e(\Phi_M^{o(s_i,e)-1}(R)) = \Lambda \text{ where t} = \tau(s_i)
\end{array}
$$

The event $\texttt{startRE}(R)$ is triggered when an occuring event is one of the events

that can appear as the first event in the RE R. The event `success`(R) is triggered when an occuring event e causes the derivative of the remainder to contain empty string, and the event `fail`(R) is triggered when an occuring event e causes the derivative of the remainder to become undefined.

Next, we define a frequency function `occur`(e, c). A function `occur`(e, c) returns the number of occurrences of an event e during the time interval that a condition c holds true. `occur`(e, c) is defined as follows. Let $f(s_i, e, c)$ denote a frequency of e in c at time $\tau(s_i)$. At time $\tau(s_i)$, `occur`$(e, c) = f(s_i, e, c)$.

$$\begin{array}{rll} f(s_i,e,c) = \Lambda & & \text{if } M,\tau(s_i) \not\models c \\ = 0 & & \text{if } M,\tau(s_i) \models start(c) \\ = f(s_{i-1},e,c) & & \text{if } M,\tau(s_i) \models c \text{ and } M,\tau(s_i) \not\models e \\ = f(s_{i-1},e,c)+1 & & \text{if } M,\tau(s_i) \models c \text{ and } M,\tau(s_i) \models e \end{array}$$

3.3 Examples

Two examples of requirements that need such timing dependencies are

(i) *an event a must not occur three times in a row,* and

(ii) *the ordered events w,x,y,z can occur out of order for less than ten times when a condition c holds true.*

Events are `a`, `b`, `w`, `x`, `y`, and `z` where events `a` and `b` are not related to events `w`, `x`, `y`, and `z`, and vice versa. To specify those requirements in the existing MEDL, we need a few auxiliary variables to keep track of when and how many times events `a`, `w`, `x`, `y`, and `z` occur. The fragment of MEDL below shows how we specify the above requirements in the original MEDL.

```
var aCount, wxyzCount;
alarm a3 = a when aCount == 3;
event wxyz = ( y||z when [w,x) ) ||
             ( z when [w, end([x,y))) ) when c;
property wxyz10 = wxyzCount < 10;
a -> { aCount' = aCount+1; }
b -> { aCount' = 0; }
wxyz -> { wxyzCount' = wxyzCount+1; }
```

The `aCount` variable keeps track of how many times an event `a` occurs. Whenever `a` occurs, we increment it and whenever `b` occurs, we reset it. The alarm `a3` alerts users when `aCount` becomes 3. The event `wxyz` is triggered only when `c` holds true and when `y` or `z` occurs between `w` and `x` or when `z` occurs between `x` and `y`. The `wxyzCount` variable stores the number of times the event `wxyz` has occured. The property `wxyz10` alerts users when `wxyzCount` is greater than 10.

Consider when we need to specify the ordering of more than four events, the event such as `wxyz` can get very complicated because too many cases need to be considered. This shows that writing requirements using the existing MEDL is error-prone and difficult to understand. By adding regular expressions to the language, an order of events can be expressed more intuitively. The below fragment of the specification shows how to specify the example requirements in the extended MEDL-RE.

```
RE a3RE {b} = <a.a.a>;
RE wxyzRE {} = <w.x.y.z>;
alarm a3 = success(a3RE);
property wxyz10 = occur(fail(wxyzRE),c) < 10;
```

The regular expression `a3RE` and `wxyzRE` denote the ordering of events `a` occurring three times in a row and the ordering of the event `w` followed by the events `x`, `y`, and `z`. The relevant set `{b}` in `a3RE` indicates that if an event `b` occurs between two events `a`, then this sequence would fail to match `a3RE`. However, the MEDL-RE would ignore an event `x` if it occurs between two events `a` and would not fail the sequence. The relevant set `{}` in `wxyzRE` works similarly. The alarm `a3` alerts users when we successfully match the RE `a3RE` while the property `wxyz10` alerts users when the RE `wxyzRE` fails more than ten times. The requirements now are much simpler and easier to understand than the original MEDL.

4 Implementation

An important property of monitoring is its ability to monitor target systems as efficiently and quickly as possible using minimal system resources. Hence, using a deterministic finite automaton (DFA) to monitor a RE is preferred over a non-deterministic finite automaton (NFA). Several existing algorithms have been proposed to efficiently construct and minimize a DFA from a given RE. We have chosen the algorithm by Aho, Sethi and Ullman [2] because it generates a DFA directly without generating an NFA. The empirical result by Watson [15] also suggests that this DFA construction is efficient.

Simulation of a DFA involves the following steps. Identify a RE that has the current event in its relevant set. Then compare the current event with all the possible transitions from the current state of the DFA and take the appropriate one. If the current event initiates the simulation of a DFA, then it would trigger a `startRE` event. If the automaton moves to a final state, a `success` event is triggered under certain conditions as specified in Section 5. Similarly, the event causing the automaton to get stuck triggers a `fail` event. However, the problem in DFA simulation arises when there are simultaneous events generated by one primitive event or condition. Since there is no inherent

order amongst these simultaneous events, the DFA simulation must evaluate multiple paths simultaneously for all possible permutations of these events. This is expensive in terms of resource utilization and may also be non-viable for certain applications especially in real time systems.

4.1 Causes of Simultaneous Events

The MaC language consists of high-level or composite events and conditions, and low-level or primitive events and conditions described in Section 2.1 and 2.2. When primitive events or conditions are detected, the filter sends them to the event recognizer in the order as they occur. These primitive events and conditions can trigger composite events or change the value of conditions. While the primitive events and conditions are ordered, we cannot order the composite events triggered by one primitive event or conditions because all of the composite events occur at the same time.

Since REs can be defined on composite events that can occur individually and simultaneously, we need a way to recognize the events that can occur simultaneously. If two events can occur simultaneously, then their order is insignificant and their concatenation can be permuted without changing the meaning. To handle simultaneous events inexpensively, we propose the following steps. During the static phase, we determine if events used in the REs could occur simultaneously. If such events exist, we augment the original DFA with alternative transitions. At runtime, we try to take a step using the original transitions. If it is not possible, we try to take a step using the alternative transitions only if there exists a set of events from among the relevant set of this RE that have occured simultaneously.

4.2 Detecting Simultaneous Events

To detect possible simultaneous events statically, we construct a dependency graph from the syntax of events and conditions using connectives described in Section 2. The dependency graph $G = (V, E)$ is a directed graph where the vertices V are connectives, events, and conditions, and the edges E represent dependency between them. Let e_1, e_2 be events or conditions. Then, $(e_1, e_2) \in E$ if and only if e_1 is used to compose e_2. For example, if $e_3 = e_1||e_2$, then $(e_1, ||)$, $(e_2, ||)$, and $(||, e_3)$ are in E. Fig. 5 shows an example monitoring script, and Fig. 6 shows the corresponding dependency graph. In the figures, there are two primitive events `update(A.x)`, `update(A.y)` and two primitive conditions `A.c1`, `A.c2`. All of them are vertices at the bottom of Fig. 6. Since the event `update(A.x)` and the condition `A.c1` are used to compose a connective `when`, edges (`update(A.x)`, `when`) and (`A.c1`, `when`) are in E. Since the event `e1` is

```
e1 = update(A.x) when A.c1;
e2 = update(A.x) when A.c1 && A.c2;
e3 = update(A.y) when A.c2;
RE test {} = e1 . e2 . e3;
```

Fig. 5. Monitoring script

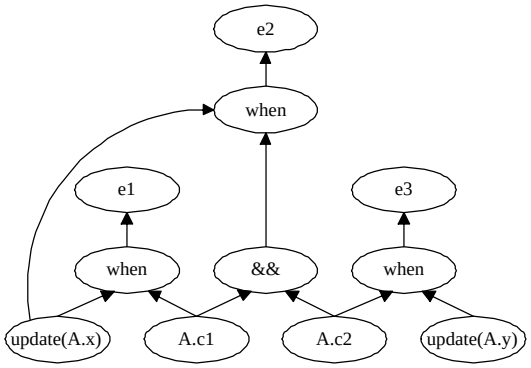

Fig. 6. Dependency graph

composed of the connective `when`, an edge (`when`, `e1`) is also in E. The events `e2`, `e3` are constructed similarly.

We denote that an event e_2 depends on an event e_1 if and only if there exists a path from an event e_1 to an event e_2 in G. We also denote that if there is a path to each of e_1 and e_2 from a common vertex in the graph, then they can occur simultaneously. Thus, a set of events that can be reached from the same primitive event in the dependency graph can all occur simultaneously. For example, in Fig. 6, the events `e1`, `e2` depend on the same primitive event `update(A.x)`, and therefore, they can occur simultaneously. To detect dependency, we traverse the graph in a bottom-up fashion using BFS.

However, the analysis of the dependency graph can yield a false positive result. This means that simultaneous events obtained from the graph might never occur in the actual system. For example, consider the events in Fig. 6. Based on the dependency graph, the events `e1` and `e2` can occur simultaneously because both depend on the event `update(A.x)`. Suppose that the conditions `A.c1` and `A.c2` are boolean variables in the target program and they are never true at the same time. Then, events `e1` and `e2` never occur simultaneously.

There are two special cases. The first case is when an event is composed using an *and* connective (e_1 && e_1). The event with the && connective is triggered if and only if both of its two arguments e_1 and e_2 occur simultaneously. Therefore, if there exists an && connective along the path, the two arguments of the && connective must depend on one common vertex. Otherwise, the event with the && connective will never be triggered and therefore, can be

eliminated from possible simultaneous events. The second case is when an event is composed using a *when* connective (e `when` c), which consists of an event side e and a condition side c. The event with the `when` connective is triggered if and only if the event e occurs when the condition c is true. Assume the two events we are detecting are e_1 and e_2, and e_1 is composed using a `when` connective. If the common vertex of e_1 and e_2 lies on the event side of the `when` connective in e_1, then e_1 and e_2 can occur simultaneously. If it lies only on the condition side, then e_1 and e_2 cannot occur simultaneously. For example, the events `e2` and `e3` in Fig. 6 cannot occur simultaneously. This is the case because `update(A.x)` and `update(A.y)` cannot occur simultaneously based on our assumption that all primitive events are always ordered. However, the events `e1` and `e2` can occur simultaneously if both `A.c1` and `A.c2` are true at the same time.

A set of possible simultaneous events obtained by the above process can be used to identify REs that would require augmentation. Only those regular expressions that have a subset of simultaneous events in their relevant sets need to be augmented.

5 Algorithm for Augmenting and Simulating DFA with Simultaneous Events

Based on the information of simultaneous events provided in Section 4, we incorporate additional information into the DFA statically. We claim that this additional information can assist the simultaneous simulation of multiple paths in the DFA efficiently. The following algorithm is proposed to incorporate this additional information into the DFA.

Let the DFA be a minimal DFA generated using the algorithm for DFA generation and minimization as described in Section 4. We call this DFA as DFA_{org} and the augmented DFA generated by the algorithm as DFA_{aug}. We construct DFA_{aug} under the following premises.

(i) The set of simultaneous events generated by the event recognizer does not constitute multiple occurrences of the same event. This assumption is valid because the MaC system does not record the number of occurrences of an event at any given time instant. It only records the presence or absence of the event occurrence.

(ii) DFA_{org} consists of a single accepting state. Since we detect only the shortest sequence accepted by DFA_{org}, we can remove all outgoing transitions from all the accepting states. The minimization procedure would then merge those states into one.

Let $DFA_{org} = (Q, \Sigma, T, q_0, q_f)$ where Q is a set of finite states, Σ is a relevant event set of R as described in Section 3.2, $T : Q \times \Sigma \rightarrow Q$ is a partial transition function, $q_0 \in Q$ is the start state, and $q_f \in Q$ is the single accepting state. Given DFA_{org}, the algorithm constructs $DFA_{aug} = (Q, \Sigma, T, q_0, q_f, D, U, ES, T_{alt})$ where Q, Σ, T, q_0, and q_f are obtained from DFA_{org}, $D : T \rightarrow \mathbb{N} \cup \{0\}$ is the *distance* annotation of a transition indicating its longest distance from q_f, $U : T \rightarrow \mathbb{N} \cup \{0\}$ is the *unique distance* function that maps each transition of a state to a unique natural number or zero, $ES : Q \rightarrow \mathcal{P}(\Sigma^*)$ is an *event string* annotation of a state, and $T_{alt} : Q \times \Sigma \rightarrow Q$ is a partial *alternative transition* function. We denote $T(q, e) = \Lambda$ iff $(q, e) \notin dom(T)$ and denote $\langle q, e, q' \rangle \in T$ iff $T(q, e) = q'$. D, U, ES, and T_{alt} are described in further details in the following sections. The algorithm for DFA_{aug} construction has two phases: Annotation of transitions and states, and generation of alternative transitions.

5.1 Phase 1 : Annotation of Transitions and States

During this phase, we annotate each transition $t \in T$ of DFA_{org} with a numeric value denoted as $D(t)$, such that the value

(i) equals the distance of the transition t from the accepting state q_f. This distance is equal to the number of transitions to be traversed to reach q_f from the current state using this transition t.

(ii) denotes the maximum of all such distances.

Transitions that constitute cycles or self loops are ignored during this phase. This distance generation phase of the algorithm is shown in Fig. 8. This annotation can be done by traversing the DFA using a BFS algorithm starting at q_f.

This distance information $D(t)$ will assist in ordering of transitions for simulation at runtime. We will traverse one of the outgoing transitions from the current state based on the distance values associated with the transitions, i.e., lower distance value first. The rationale behind this choice is that transitions that are closer to the accepting state should be attempted before those that are farther away from it. In case of a conflict between transitions (i.e., two transitions having the same distance value), we pick transitions based on the ordering of corresponding events in the relevant set. In Fig. 8 the start state S has two outgoing transitions both numbered 3. We will order these transitions based on the ordering of events $e1$ and $e2$ in the relevant set. Let this unique ordering of transitions t be called $U(t)$.

During this phase, we also annotate each state of DFA_{aug} with additional information, indicating the set of possible sequences of input events that could

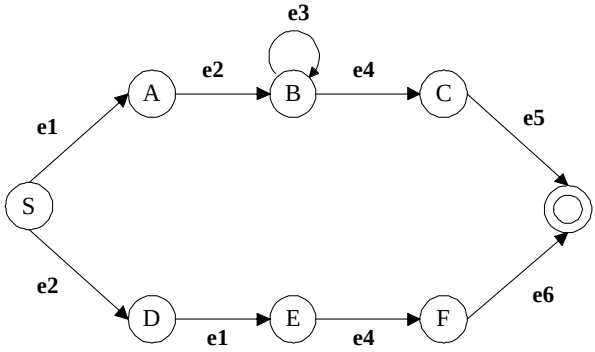

Fig. 7. Original minimized DFA_{org}

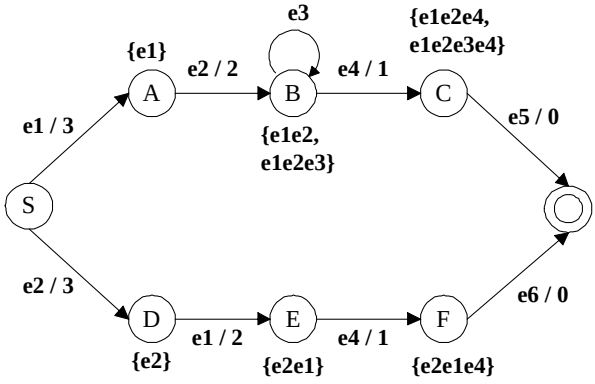

Fig. 8. Distance D and event string ES annotations of DFA_{aug}

be encountered to reach this state starting from the start state q_0. Each such input sequence is called the *event string* of that state. Please note that there can be multiple event strings for each state. This annotation step is shown in Fig. 8. The *event string* set for each state q, called $ES(q)$, is equal to the concatenation of the set of event strings of all its predecessor states with the corresponding transitions. All states that have self loops on an event e have the event e appended to each of their event strings after the event strings are replicated. As shown in Fig. 8, the state B has a self loop on event $e3$. Its set of event strings are then given by the event string $\{e1e2\}$ for the normal transition and the event string $\{e1e2e3\}$ for the self loop. In case there is a cycle *abca*, then *abc* is appended to each of the event strings of the states that constitute the cycle after replication of the event strings. Please note that determination of event strings proceeds to the next state only after it is completely determined for the present state, i.e., all self loops for the current state must be resolved before proceeding to the next state. Cycles would be

```
1  for each state q ∈ Q − {q0, qf}
2    for each event string es ∈ ES(q)
3      for each event e ∈ Σ such that T(q, e) = Λ
4        for each state q1 ∈ Q − {q0} such that
5        max_{∀ei∈Σ,T(q1,ei)≠Λ}(U(T(q1, ei))) ≥ min_{∀ei∈Σ,T(q,ei)≠Λ}(U(T(q, ei)))
6          for each event string es1 ∈ ES(q1)
7            if (Hashed(es1) == Hashed(es))
8              then Add ⟨q, e, T(q1, e)⟩ to Talt (if T(q1, e) ≠ Λ)
9            else if (Hashed(es) == Hashed(Remove(e, es1)))
10             then Add ⟨q, e, q1⟩ to Talt
```

Fig. 9. Algorithm to generate alternative transitions T_{alt}

an exception here because we cannot identify cycles until we proceed ahead and return back to the state. Even in this case, the algorithm will proceed to states following the cycle only after all the event strings for all the states in the cycle have been determined.

5.2 Phase 2: Generation of Alternative Transitions.

Before we proceed, we need to define the terms *linearization* and *equivalence of linearization.*

Linearization *Linearization is a total order of a set of events. If this set consists of events that have occured simultaneously then the ordering of such simultaneous events in the set is one linearization of this set. A different ordering of these simultaneous events then constitutes a different linearization of this set.*

Equivalence *Two linearizations are equivalent if and only if they are different permutations of the same set of events.*

During this phase, we add alternative transitions to appropriate states in DFA_{org}. The alternative transitions take the DFA from a state representing one linearization to another state representing an equivalent linearization for some set of simultaneous events. Let $Remove(e, es)$ be an event string obtained by removing one occurrence of e from an event string es. Let $Hashed(es)$ be a hash value associated with an event string es where the ordering of events in es is insignificant. This means two event strings have the same hash value if and only if they are equivalent linearizations. For example, $Hashed(abc) = Hashed(cab)$ but $Hashed(aabc) \neq Hashed(abc)$.

The algorithm shown in Fig. 9 compares the event strings of two states to see whether they are different linearizations of the same input sequence. If so, we say that the two event strings are equivalent linearizations of the

input sequence, and we could add alternative transitions from one state to another without changing the meaning of the DFA. Since we always choose a transition t from a state q with the lowest $U(t)$ first, the order of paths taken is unique. Thus, we need to add alternative transitions only from a state with lower $U(t)$ to the states with higher $U(t)$. This means that if q and q_1 are states, then we add alternative transitions from q to q_1 if and only if the maximum value of $U(t_1)$ among all transitions t_1 of q_1 is greater than the minimum value of $U(t)$ among all transitions t of q (lines 4–5). When adding the alternative transitions annotated with an event e from q to q_1, we ensure that the existing transitions in q do not have a transition annotated with the event e (line 3). Thus, the resulting finite automata is still deterministic.

There are two different cases which the algorithm handles. Assume $es \in ES(q)$ and $es_1 \in ES(q_1)$ are two event strings such that q has lower $U(t)$ than q_1. The first case is when some event strings of the two states q and q_1 have different but equivalent linearizations. For example, $es = abc$ and $es_1 = bca$. Here the algorithm adds an alternative transition on the current event e from the state q to the state represented by $T(q_1, e)$ (lines 7–8). This transition is added for each event e such that there are no transitions currently present from q on any of these events. This case is shown in Fig. 10. Here, q=F, q_1=C, es=$e2e1e4$, es_1=$e1e2e4$ and e=$e5$. The algorithm then adds a transition on $e5$ to $T(C, e5)$ which is the accepting state q_f.

The second case is when $Remove(e, es_1)$ and es represent equivalent linearizations for the two states q and q_1. For example, if $es = ba$, $es_1 = aeb$, and the current event is e, then the removal of e from es_1 returns an event string which is equivalent to es. Here the algorithm adds an alternative transition on the current event e from q to q_1 (lines 9–10). Again the transitions are added only for those events that currently do not have transitions from q. This case is shown in Fig. 11. Here, q=F, q_1=C, es=$e2e1e4$, es_1=$e1e2e3e4$ and e=$e3$. The algorithm then adds a transition on $e3$ to C.

After completion of this phase, we can delete the event string information from all the states except the ones with $D(t)$ values of 1. We call these states the penultimate states. Their event strings will be used to verify the acceptance of the input sequence. This process will be explained in the next section.

5.3 DFA Simulation at Runtime

At runtime, we simulate DFA_{aug} without the alternative transitions as long as there is no occurrence of simultaneous events. On the first occurrence of simultaneous events, we activate all the alternative transitions. From the current state, we then take the path which is closest to the accepting state,

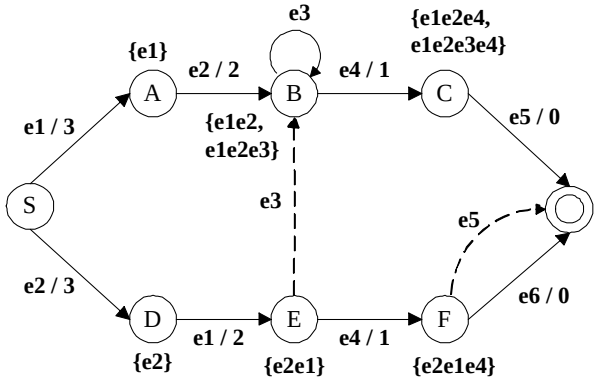

Fig. 10. Case 1 of the algorithm in Fig. 9

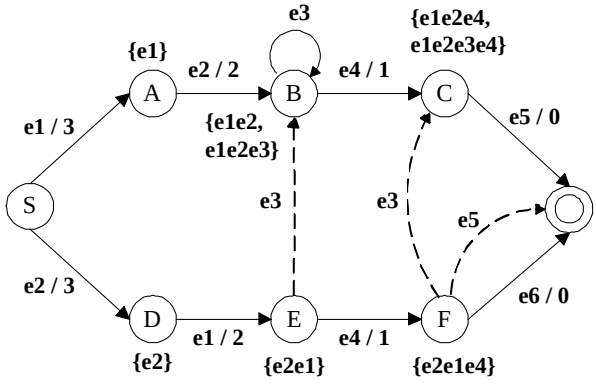

Fig. 11. Case 2 of the algorithm in Fig. 9

i.e., the path with the lowest $U(t)$. Next, we simulate DFA_{aug} normally taking into consideration the alternative transitions and $U(t)$.

Because events that occur simultaneously can also occur individually, some linearization that DFA_{aug} accepts might not be accepted by DFA_{org}. When DFA_{aug} reaches the accepting state, we verify by comparing the input sequence and the event strings in the penultimate states that have an original transition on the last event in the input sequence to the accepting state. Because DFA_{org} is minimal, only one such penultimate state exists. If they match, DFA_{aug} accepts that input sequence. Otherwise, DFA_{aug} rejects.

To be able to verify, we need to keep the history of all events seen from the start state. The history also keeps information about which events have occured simultaneously and which have occured individually. We do so by storing the input sequence as an array of simultaneous sets. To handle Kleene star, we have a flag to indicate whether or not each state has been visited.

```
1  i = 0, j = 0
2  for each e_i ∈ ES_{e_last}
3    if e_i ∈ h_j
4      then Remove e_i from h_j
5    else
6      then Reject this input string
7    if h_j is empty
8      then j = j + 1
9  Accept this input string (if it has not been rejected)
```

Fig. 12. Verifying Algorithm

We then keep only events that take the simulation from one state to another unvisited state. This way, the resulting history string will resemble the way event strings are calculated and simplify our verification algorithm. Let

H = A history of the input sequence.
e_{last} = The last event in the input sequence.
$ES_{e_{last}}$ = A set of event strings associated with the penultimate state, which has a transition in DFA_{org} on e_{last} to the accepting state.
e_i = A set of events in an event string where i is the position in the event string.
h_j = A set of simultaneous events $h_j \in H$ where h_j occurs before h_{j+1}.

The verifying algorithm in Fig. 12 is straightforward. For each $e_i \in ES_{e_{last}}$ (line 2), we check if e_i occurs in the current simultaneous set (line 3). If not, the input string is rejected (line 6). When the current set is completely parsed without being rejected, we move to the next set (lines 7–8). If we reach the end of $ES_{e_{last}}$ without being rejected, we accept the input sequence (line 9). For example, let the input sequence recorded when the DFA accepted be $\{e1e2, e4, e5\}$ where events $e1$ and $e2$ have occured simultaneously. Since the last event in the sequence, i.e., $e_{last} = e5$, we have $ES_{e_{last}} = \{e1e2e4, e1e2e3e4\}$. Now since event string $\{e1e2e4\}$ in $ES_{e_{last}}$ matches with the recorded input sequence $\{e1e2, e4\}$, DFA_{aug} will accept this input sequence.

5.4 Analysis of the Algorithm

This augmentation algorithm has the following running times during each phase. For phase 1, we run BFS twice, and thus the running time is $O(s + t)$ where $s = |Q|$ is the number of states , and $t = |dom(T)|$ is the number of transitions. For phase 2, the running time is $O(s^2 \cdot e \cdot es^2)$ where $e = |\Sigma|$ is

the number of events in the relevant set and $es = \sum_{q \in Q} |ES(q)|$ is the number of possible event strings in the regular expression, (worst case analysis). This then constitutes the compile time penalties for augmenting the DFA. The table in Fig. 13 compares the running time of this algorithm with the naive algorithm of dynamically simulating all possible paths. Since there can be $O(2^n)$ such paths where n is the length of the input string, the dynamic simulation approach can impose an additional running time of $O(2^n)$. In our algorithm, $O(n)$ added to the running time is for verification of the input sequence when DFA_{aug} accepts. The additional space is only $O(n^2)$ because we add alternative transitions only from lower numbered to higher numbered paths, which is $\sum_{i=1}^{n} i = O(n^2)$. Therefore, our algorithm is clearly prefered over the naive algorithm.

Algorithm	**Running Time**	**Space**
Dynamic simulation of all paths	$O(2^n)$	0
Augmenting with alternative transitions	$O(n)$	$O(n^2)$

Fig. 13. Comparison between two algorithms

5.5 Equivalence of DFA_{org} and DFA_{aug}

In this section, we provide a proof that DFA_{org} and DFA_{aug} are equivalent.

Theorem 5.1 *DFA_{aug} accepts if and only if DFA_{org} accepts.*

Proof.

Part 1: If DFA_{org} accepts, DFA_{aug} accepts.

This statement can be restated as if DFA_{org} accepts some linearization of the input sequence, then DFA_{aug} accepts all of its equivalent linearizations. We define a path and a set of paths as follows. A path in DFA_{org} is any path from the start state q_0 to the accepting state q_f. Every such path is uniquely ordered by the algorithm. A set of paths is an enumeration of all possible paths from q_0 to q_f of DFA_{org}. By the description of the algorithm, every alternative transition is a transition from one path to another path in the set, and the transitions are always taken from lower $U(t)$ to higher $U(t)$.

Let a be the event currently being considered. Let q_1 and q_2 be the two states being considered. Let es_1 and es_2 be the two event strings associated with q_1 and q_2, respectively. Assume the current state is q_1, and therefore, es_1 is one linearization of input seen so far. The algorithm has the following alternatives.

(i) The event string es_2 is a different equivalent linearization of input seen so far, i.e., ($Hashed(es_1) = Hashed(es_2)$). Then, the algorithm adds

an alternative transition from q_1 to a state q_3 on a such that a transition from q_2 to q_3 on a exists, i.e., $T(q_2, a) = q_3$. Thus, this alternative transition takes a step from one linearization to the other. ($Hashed(concat(es_1, a)) = Hashed(concat(es_2, a))$).

(ii) The removal of a from es_2 is a different equivalent linearization of the input seen so far. ($Hashed(es_1) = Hashed(Remove(a, es_2))$). Then, this algorithm adds an alternative transition from q_1 to q_2 on a. Thus again this alternative transition takes a step from one linearization to the other. ($Hashed(concat(es_1, a)) = Hashed(es_2)$).

Now we prove that such alternative transitions exist for every pair of equivalent linearizations. Let states q_1, q_2, q_3, q_4 belong to paths p_1, p_2, p_3, p_4, respectively, and let the ordering of these paths be $p_1 < p_2 < p_3 < p_4$. Then the ordering of states is $q_1 < q_2 < q_3 < q_4$. If one of the event strings associated with each of these states are different equivalent linearizations and the ordering is as given above, then it can be seen that the algorithm repeats itself for each pair of states $(q_1, q_2), (q_1, q_3), (q_1, q_4), (q_2, q_3), (q_2, q_4)$ and (q_3, q_4) in that order. Thus, the algorithm provides a path to go from the current linearization of the input to any other different equivalent linearization governed by the unique ordering of linearizations (or paths).

Since at runtime we will simulate DFA_{aug} based on the ordering of the paths; from a given path of $U(t) = k$, we always have a path using the alternative transitions to all paths with $U(t)$ greater than k and having some equivalent linearization. Thus, DFA_{aug} will eventually reach the path, which represents the linearization for which DFA_{org} accepts. Once it reaches this path, the process of picking original transitions first ensures that we will never leave this path.

Part 2: If DFA_{aug} ***accepts,*** DFA_{org} ***accepts.***

We prove this case by contradiction. Assume that DFA_{aug} accepts and DFA_{org} does not for some input sequence. By the definition of the algorithm, DFA_{aug} accepts if the following conditions exist.

(i) Simulation reaches the accepting state of DFA_{aug}.

(ii) The input sequence on which DFA_{aug} has accepted is equivalent to the concatenation of an event e_{last} and some event string associated with the penultimate state which has a transition on an event e_{last} to the accepting state, where e_{last} is the last event that occurs in the input sequence.

From the definition of event strings at a state, we can claim that the event string stored at the penultimate state is equal to a string parsed by DFA_{org} to

reach this state. This claim is valid because while determining event strings, we use only the original transitions present in DFA_{org}. The transition taken from the penultimate state to reach the accepting state is also an original transition. Therefore, the event string concatenated by the last transition is accepted by DFA_{org}, contradicting the assumption above. □

6 Related Work

There are a few existing works that incorporate regular expressions into logic. The ForSpec Temporal Logic (FTL) [3], Intel's new formal specification language, extends a linear temporal logic [13] with the ability to specify all ω-regular properties. FTL allows a user to define temporal connectives over time windows, regular sequences of Boolean events, and then relate such events via special connectives. Sugar [4] adds an extensive set of operators including regular expressions as a *syntactic sugar* to CTL [7]. Property Specification Language [1] is a specification language for hardware modeling and verification, which supports LTL [13] and extended regular expressions. Monitoring oriented Programming (MoP) [6] provides a monitoring architecture based on LTL [13] and extended regular expressions. Temporal Rover [8] is an architecture that helps a system do monitoring. Its specification language uses LTL [13] and MTL [12] with regular expressions and Time-Series. Time-Series observes temporal properties over time and is used for properties like stability, monotonicity, temporal average, sum, and max/min value. Comparing to the MEDL-RE based on LTL and regular expressions, FTL [3], MoP [6], and Temporal Rover [8] provide similar languages based on LTL and/or MTL and regular expressions while Sugar [4]'s language is based on CTL and similar regular expressions. However, none of them seem to have the issue of simultaneous events and therefore no algorithm for checking dependency and augmenting a DFA has been proposed.

7 Conclusion

We have presented an algorithm to simulate an augmented DFA for the extension MEDL-RE. The MEDL-RE incorporates regular expressions, which provide a more intuitive and less error-prone language to express complex dependencies between sequence of events, timing constraints, and a frequency of events during a time interval. The events associated with a regular expression offers the ability to detect the instance when the regular expression starts and the instance when we succeed or fail to find the regular expression. The DFA is augmented by adding alternative transitions after we statically detect the

possibility of simultaneous events. We also prove that the augmented DFA is equivalent to the original DFA.

References

[1] Accellera Organization, Inc. *Property Specification Language Reference Manual, Version 1.01*, 2003. http://www.accellera.org/.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. Tiemeyer, M. Y. Vardi, and Y. Zbar. The Forspec Temporal Logic: A New Temporal Property-Specification Language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 296–211, 2002.

[4] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, and Y. Rodeh. The Temporal Logic Sugar. *Lecture Notes in Computer Science*, 2102:363–367, 2001.

[5] J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, 1964.

[6] F. Chen and G. Rosu. Towards Monitoring-Oriented Programming: A Paradigm Combining Specification and Implementation. In *Proceedings of the 3nd International Workshop on Run-time Verification*, July 2003.

[7] E. M. Clarke and E. A. Emerson. Synthesis of Synchronization Skeletons for Branching Time Temporal Logic. In *Logic of Programs: Workshop Lecture Notes in Computer Science Vol 131, Dexter and Kozen (editors)*, Yorktown Heights, New York, 1981. Springer-Verlag.

[8] D. Drusinsky. Monitoring Temporal Rules Combined with Time Series. In *Proceedings of the 2003 Computer Aided Verification Conference (CAV)*, July 2003.

[9] M. Kim. *Information Extraction for Run-time Formal Analysis*. PhD thesis, University of Pennsylvania, 2001.

[10] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, Checking, and Steering of Real-Time Systems. In *Proceedings of the 2nd International Workshop on Run-time Verification*, July 2002.

[11] M. Kim, M. Viswanathan, S. K. Hanêne Ben-Abdallah, I. Lee, and O. Sokolsky. Formally Specified Monitoring of Temporal Properties. In *Proceedings of the European Conference on Real-Time Systems - ECRTS'99*, pages 114–121, June 1999.

[12] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *RealTime Systems*, 2(4):255–299, 1990.

[13] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[14] U. Sammapun and O. Sokolsky. Regular Expressions for Run-Time Verification. In *Proceedings of the 1st International Workshop on Automated Technology for Verification and Analysis*, Dec. 2003.

[15] B. W. Watson. *Taxonomies and Toolkits of Regular Languages Algorithms*. PhD thesis, Eindhoven University of Technology, 1995.