December 2003

Regular Expressions for Run-Time Verification

Usa Sammapun
University of Pennsylvania

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Regular Expressions for Run-Time Verification

Abstract
When specifying system requirements, we want a language that can express the requirements in the simplest and most intuitive form. An expressive and intuitive language makes specifying requirements easier and less error-prone. Although our MaC system provides an expressive language, called MEDL, to specify safety requirements, it is generally awkward to express an order of events with complex timing dependencies, timing constraints, and frequencies of events. MEDL-RE extends our MEDL language to include regular expression and three associated events to easily specify timing dependencies and its timing constraints. Our regular expression is unique in a way that a user can specify which events are relevant to a regular expression. This feature makes it easy when a user wants to specify a requirement of one specific component of a system.

Keywords
Run-time verification, regular expressions, temporal logic

Comments
Presented at the 1st International Workshop on Automated Technology for Verification and Analysis 2003 (ATVA 2003), Taipei, Taiwan, December 10-12, 2003.

Regular Expressions for Run-Time Verification

Usa Sammapun and Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104

Abstract

When specifying system requirements, we want a language that can express the requirements in the simplest and most intuitive form. An expressive and intuitive language makes specifying requirements easier and less error-prone. Although our MaC system provides an expressive language, called MEDL, to specify safety requirements, it is generally awkward to express an order of events with complex timing dependencies, timing constraints, and frequencies of events. MEDL-RE extends our MEDL language to include regular expression and three associated events to easily specify timing dependencies and its timing constraints. Our regular expression is unique in a way that a user can specify which events are relevant to a regular expression. This feature makes it easy when a user wants to specify a requirement of one specific component of a system.

1 Introduction

Real-time systems are systems that concern both computation requirements and timing requirements. These requirements sometimes are difficult to achieve due to uncertainty of external environments and unforeseen bugs of either specification or implementation. Hence, most real-time system designers typically use various methods of formal verification and testing to debug and verify correctness of the systems. However, these methods have their own limitation. Formal methods verify specification but not implementation, and they do not scale up well to handle large systems. On the other hand, testing is scalable and performed on the implementation level, but it is not exhaustive.

Because of the above reasons, several researchers have proposed runtime formal analysis based on formal requirement to bridge the gap between the two techniques: formal verification and testing. The monitoring, checking and Steering (MaC) framework [9, 10, 11, 13] is one of the runtime formal analysis researches. It 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. The safety properties include both computational requirements and timing requirements. The safety properties are defined in terms of events, conditions and auxiliary variables. Events are instantaneous incidents such as variable updates or the start or the end of a method call. Conditions are propositions about the program that may be true or false for a duration of time. Auxiliary variables are temporary storage to, for example, keep counters or a time of the last occurrence of an event, which can be used to specify timing requirements.

Using events, conditions and auxiliary variables, the MEDL language provides an elegant and intuitive way to specify computational requirements. It, however, does not provide as intuitive way to specify timing requirements, i.e., specifying order of events with complex timing dependencies between events, timing constraints or keep track the numbers of specific events in a time interval are generally awkward to represent in the current MEDL language.

In this paper, we propose an extension of the MEDL language, MEDL-RE, to provide a simpler and more intuitive language to specify timing requirements. We believe that a simple and intuitive language can offer a user with clearer and less error-prone specification. This extension allows us to easily specify ordering of events, and its timing constraints in terms of start times, end times, and frequencies.

The ordering of events are in a form of regular expressions over a specified set of events. We call this set of events a relevant set of events because events not specified in the set would be ignored during evaluation of a regular expression. Each regular expression has its own relevant set. By observing a sequence of events happening in a target system, the extension MEDL-RE matches the sequence of events with specified regular expression.

The extension MEDL-RE provides three events associated with a regular expression. They are events indicating start, success and failure of finding a regular expression. These events can be composed to specify safety and timing properties. To specify start and end times of each regular expressions, users can simply use the time function, provided by our original MEDL language. These start and end times allow us to express timing constraints in a more intuitive way.

The frequencies count the number of events or regular expression of events occurred during the time duration when a specified condition holds true. This condition is a generalization of a time interval.

We want to clarify what we mean by "an order of events" and "a sequence of events". When we say "an order of events", we mean a temporal relation among specified events. When we say "a sequence of events", we mean the actual events happening in a target system.

The paper is organized as follows. Section 2 briefly explains an overview of the MaC framework. Section 3 discusses motivation and design issues. Section 4 introduces an extension MEDL-RE. Section 5 discusses implementation issues. Section 6 presents related work. Lastly, section 7 discusses future work and concludes the paper.

2 MaC Overview

2.1 MaC Architecture

The MaC system has been designed and developed to ensure that a target program is running correctly with respect to a formal requirement. Figure 1 shows the overall structure of the MaC architecture. The system works as follows. A user specifies a requirement of a target program in a formal language. The MaC formal language composes of three scripts explained in the next section: a monitoring script, a requirement specification, and a steering script. Given a target program and the requirement, the MaC system inserts a collection of probes or a *filter* into a target program.

During run-time, the probed target program are running and being monitored and checked by the MaC system. An *event recognizer* detects an event from the state information received from the *filter*. Events are recognized according to a monitoring script. Recognized events are sent to the run-time checker. A *run-time checker*, then, determines whether or not the current execution history satisfies a requirement specification. The execution history is captured from a sequence of events sent by the event recognizer. If the run-time checker

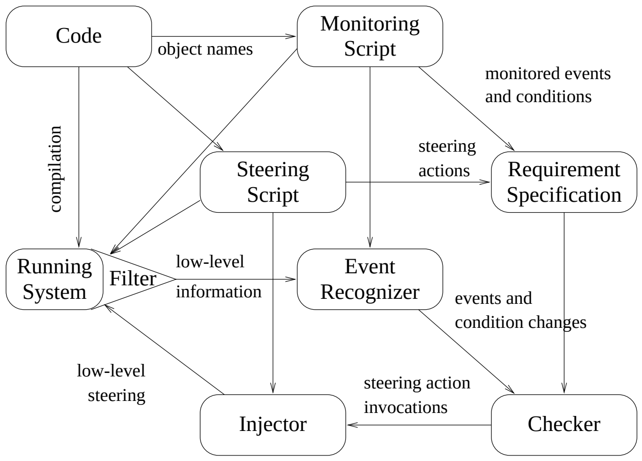

Figure 1: Overview of the MaC architecture

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 Language

The MaC system provides three languages. The requirement specification is called the Meta-Event Definition Language (MEDL). MEDL is used to express requirements in terms of events and conditions. MEDL, based on an extension of a linear temporal logic (LTL) [14], allows to express a large subset of safety properties of systems, including real-time properties. The monitoring script is expressed in the Primitive Event Definition Language (PEDL). PEDL describes primitive high-level events and conditions in terms of system objects. PEDL is used to define what information is sent from the probed target program or a filter to the MaC system, and how it is transformed into events used in high-level specification or MEDL. Both of these languages are based on the notions of events and conditions, explained in Section 2.2.1. The steering script of the Steering Action Definition Language (SADL) is used to specify actions that need to be invoked when violations occur. Since PEDL and MEDL are quite similar to each other and the main focus for this paper is the language MEDL, we are not explaining PEDL and SADL in detail here. Please see [9, 10, 11, 13] for more details.

2.2.1 Events and Conditions

Events occur instantaneously during the system execution, whereas *conditions* represent information that holds 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 `(difference < 0.1)` holds as long as the value of the variable `difference` does not exceed `0.1`. For formal semantics of events and conditions, see [9, 11, 13].

2.2.2 Meta Event Definition Language (MEDL)

The safety requirements (invariants) are written in MEDL. A MEDL specification includes the following sections:

Imported events and conditions. A list of events and conditions imported by PEDL is declared.

Definitions of events and conditions. Events and conditions are defined using imported events, imported conditions, and auxiliary variables, whose role is explained later in this section. These events and conditions are then used to define safety properties and alarms.

Safety properties and alarms. The correctness of the system is described in terms of safety properties and alarms. Safety properties are conditions that must be *always* true during the execution. Alarms, on the other hand, are events that must *never* be raised (all safety properties [14] can be described in this way).

Auxiliary variables. Auxiliary variables increase expressive power of MEDL. They can be used to define events and conditions. Their values are updated in response to events. Auxiliary variables allow us, for example, to count the number of occurrences of an event.

3 Motivation and Design Issues

3.1 Motivating Example

The motivation for our need of an extension to the MEDL language arises when we need to specify complex timing requirements. We encounter several difficulties shown below when trying to use our existing MEDL language.

- Order of events with complex timing dependencies between elements are generally awkward to represent in the current MEDL.
- To specify timing constraints, we need an auxiliary variable to keep track of the last occurrence of an event. We also need to update this variable every single time the event occurs.
- Keeping track of how many events have occured in a desired period of time requires a few auxiliary variables and constantly updating these variables.

An example of such requirements are "an event a must not happen 3 times in a row" and "the ordered events w,x,y,z can happen out of order for less than 10 times at night," assuming a system consists of six events and one condition. Events are a, b, w, x, y, z, where events a and b are not related to events w, x, y, and z, and vice versa. A condition is *night* which is true during at night.

To specify those requirements in our existing MEDL, we need a few auxiliary variables to keep track of when and how many times events a, b, w, x, y, and z happen. The fragment of MEDL below shows how we specify the above requirements in our original MEDL.

```
var happenA, count;

alarm 3a = a when happenA == 3;
event wxyz = ( y||z when [w,x) ) ||
             ( z when [w, end([x,y))) ) when night;
```

```
property 10wxyz = count < 10;

a -> { happenA' = happenA+1; }
b -> { happenA' = 0; }
wxyz -> { count' = count+1}
```

The `happenA` variable keeps track of how many times an event A happens. Whenever a happens, we increment it and whenever b happens, we reset it. The alarm `3a` alerts users when `happenA` becomes 3. The event `wxyz` is triggered only at night when y or z happens between w and x and when z happens between x and y. The `count` variable stores a number of times the event `wxyz` has occured. The property `10wxyz` alerts users when `count` is more than 10.

Consider when we need to specify the ordering of more than four or even ten events, the event such as `wxyz` can get very complicated and error-prone because of too many cases to be considered. This shows that writing requirements using our existing MEDL is awkward, error-prone and difficult to understand, especially when the ordering is more complex than the above requirements. We believe that by adding regular expressions to our language, it could ease these difficulties because regular expressions can express an order of events in a more intuitive way than our existing MEDL. Besides regular expressions itself, adding a few events associated to regular expressions such as event indicating start, success and failure of finding such regular expression could facilitate expressing system requirements.

The below fragment of the specification shows how to specify the example requirements in our extended MEDL-RE. The regular expression `3aRE` and `wxyzRE` denote the ordering of events a happening two times in a row and the ordering of the event w followed by the events x, y, z. The alarm `3a` and and the property `wxyz` alert users when we find the first ordering, and when the second ordering fails more than 10 times at night. The requirements now are much simpler and easier to understand than our original MEDL.

```
RE 3aRE = <a.a.a>;
RE wxyzRE = <w.x.y.z>;
alarm 3a = success(3aRE);
property 10wxyz = occur(fail(wxyzRE),night) < 10;
```

However, this is not quite correct because there are six events occurring but, for example, the regular expression `wxyzRE` is composed of only four events. If a occurs right after x occurs, `wxyzRE` will fail. But because event a and b are not related to event w, x, y, and z, `wxyzRE` should not fail. We could rewrite the two regular expressions as:

```
RE 3aRE = <a.(w+x+y+z)*.a.(w+x+y+z)*.a>;
RE wxyzRE = <w.(a+b)*.x.(a+b)*.y.(a+b)*.z>;
```

As shown above, specifying events of two non-related components can complicate a requirement. We propose that we associate each regular expression with a relevant set and concern only events specified in the regular expression and events specified in the relevant set. The two above regular expressions become:

```
RE 3aRE {b} = <a.a.a>;
RE wxyzRE {} = <w.x.y.z>;
```

which is much simpler. The set {b} in `3aRE` indicates that if an event b occurs between two events a, then this sequence would fail to match `3aRE`. However, the MEDL-RE would ignore an event x if it occurs between two events a and would not fail the sequence.

3.2 Design Issues

Before getting into details about our syntax and semantics, we need to decide on several designing issues regarding regular expressions because one regular expressions can match more than one sequence of events. Our concerns regarding regular expressions are as follows.

1. **Do we automatically include events in a regular expression into a relevant set?** We are concerned about this issue when we monitor a distributed system where two machines trigger the same events. Since those same events from different machines are not related to each other, we want to be able to exclude the same events of different machines from a relevant set. Therefore, we have decided that we would automatically include events in a regular expression into a relevant set for events from the same machine and users can manual include events specified in a regular expression into a relevant set from a relevant set for events from different machine.

2. **If there are more than one place to start evaluating a regular expression, when should we start? Earliest or latest?** We take the earliest because we would not know if any more valid event is coming.

3. **If there are more than one place to end evaluating a regular expression, when should we end? Earliest (shortest sequence), latest (longest sequence), or report all?** We are choosing the shortest and not the longest because the longest can be infinite and reporting all can be redundant.

4. **How do we deal with overlapping sequence?** For each regular expression, only one instance of the regular expression can be checked at one time. For example, if a regular expression R is "a.b.a*.c" and a sequence of occuring events is "a, b, a, c." We consider the second 'a' as 'a*' in the regular expression and would not start another instance where the second 'a' is a start of the regular expression R.

4 Syntax and Semantics of MEDL-RE

Now we extend our MEDL language to include ordering of events with complex timing dependencies by expressing them as a regular expression and to include a frequency of events in a time interval. The regular expression has its associated events that help specifying timing constraints such as start, success and failure of regular expression.

4.1 Syntax

Atomic events are events in the original MaCS framework described in Section 2.2.1. These events can be primitive or composed by other events, conditions and auxiliary variables. Each regular expression ranges over its own set Σ of atomic events. It consists of atomic events (e), concatenation (R . R), union (R+R), and Kleene star (R*). We define the regular event expression syntax as follows.

$R ::= e \mid R\ .\ R \mid R{+}R \mid R^*$

The regular event expression R itself is neither a condition nor an event. It can be considered as a declaration of a regular expression, which has three associated events: `startRE`(R), `success`(R), and `fail`(R). The `startRE`(R) event is used to indicate that we start observing this regular expression. The `success`(R) event indicates that we have found a sequence of events that specifies the expression R, and the `fail`(R) event indicates that we have started observing but fail to finish finding such sequence.

For an event e and a condition c, the function `occur`(e, c) returns a frequency or a number of occuring of an event e during the time interval when a condition c holds true.

4.2 Semantics

In order to correctly define our regular event expression, we need to specify a set Σ of events for a regular expression R. Let Σ_R denote a set of events specified in a regular expression R and Σ_V denote a set of events specified in a relevant set of R described in Section 3. Then, $\Sigma = \Sigma_R \cup \Sigma_V$.

We also modify the model M defined in [9]. A model M is a tuple (S, τ, L_C, L_E, o), where $S = \{s_0, s_1, \ldots\}$ a set of states, τ is a mapping from S to the discrete time domain, L_C is a total function from $S \times \mathcal{C}$ to $\{true, false, \Lambda\}$ where $\mathcal{C}$ denotes a set of conditions and Λ denotes undefined, and L_E is a partial function from $S \times \mathcal{E}$ to a value domain. 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 occuring e_k is distinct. Although those events have the same time stamp, they are considered to occur at different time and have different orders. In other words, each event occurs a distinct time. We define such order of an event e_k at state s_i as $o(s_i, e_k)$ where 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.

Derivative of a Regular Expression We start defining our semantics by reminding a reader with the definition of a derivative of a regular expression [5]. The derivative of a regular expression R with respect to an alphabet a produces a set of strings by chopping the prefix a off from each string in R. In other words, if a DFA M simulates a regular expression R, then the derivative is a regular expression resulted from taking a step in M with an input a. Formally, for any regular expression R and any alphabet a, a *derivative of R with respect to a*, denoted by $D_a(R)$, is the regular expression

$$D_a(R) = \{x \in \Sigma^* | ax \in R\}$$

The semantics of a derivative of a regular expression is defined as follows.

$$\begin{aligned}
D_a(a) &= \epsilon \\
D_a(b) &= \Lambda \\
D_a(\Lambda) &= \Lambda \\
D_a(\epsilon) &= \Lambda \\
D_a(R+S) &= D_a(R) + D_a(S) \\
D_a(R*) &= (D_a(R))\ .\ R^* \\
D_a(R\ .\ S) &= \begin{cases} (D_a(R))\ .\ S & \text{if } E(R) = \texttt{false} \\ (D_a(R))\ .\ S + D_a(S) & \text{if } E(R) = \texttt{true} \end{cases}
\end{aligned}$$

We also need to define a function $E(R)$ from Σ^* to *boolean* to test whether $\epsilon \in R$. This test is needed in $D_a(R\ .\ S)$.

$$\begin{aligned} E(a) &= \texttt{false} \\ E(\Lambda) &= \texttt{false} \\ E(\epsilon) &= \texttt{true} \\ E(R+S) &= E(R) \vee E(S) \\ E(R\ .\ S) &= E(R) \wedge E(S) \\ E(R^*) &= \texttt{true} \end{aligned}$$

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 regular expression R.

$$\texttt{FIRST}(R) = \{a \in \Sigma \cup \{\epsilon\} | ax \in R \ \mathit{where}\ x \in \Sigma^*\}$$

$\Phi^o_M(R)$ represents the remainder of the regular expression R at an order o after a sequence of derivatives. We define $\Phi^o_M(R)$ as follows.

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

We can now define the semantics of a regular expression and its associated events. We define a language $\mathcal{L}\ (R)$ of R as follows:

$$\begin{array}{ll} \mathcal{L}(\emptyset) = \emptyset & \mathcal{L}(\epsilon) = \{\epsilon\} \\ \mathcal{L}(\mathrm{a}) = \{\mathrm{a}\} & \mathcal{L}(R+S) = \mathcal{L}(R)\ \cup\ \mathcal{L}(S) \\ \mathcal{L}(R\ .\ S) = \{x_1\ .\ x_2 \mid x_1 \in \mathcal{L}(R) \text{ and } x_2 \in \mathcal{L}(S)\} & \mathcal{L}(R^*) = (\mathcal{L}(R))^* \end{array}$$

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.

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

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 regular expression R. The event $\texttt{success}(R)$ is triggered when an occuring event e causes the derivative of the remainder to contain empty string, and the event $\texttt{fail}(R)$ is triggered when an occuring event e causes the derivative of the remainder to become undefined.

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

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

At time $\tau(s_i)$, $\texttt{occur}(e, c) = f(s_i, e, c)$.

```
import event sendToCtrl, recvVolt, sendToIP;
import action killCtrl{};

RE dmTask = <sendToCtrl.recvVolt.sendToIP>
event OK = successRE(dmTask);
event notOK = failRE(dmTask);

RE deadCtrl {OK} = <notOK.notOK.notOK>;
event deadCtrlOccur = successRE(deadCtrl);

deadCtrlOccur -> { invoke killCtrl(); }
```

Figure 2: Excerpts from the `IP.medl`

4.3 Example Revisited

We are using an inverted pendulum (IP) example introduced in [10]. The example demonstrates how to use steering to steer the inverted pendulum system back to a safe state. The architecture of the control system for the inverted pendulum is based on the Simplex architecture for control systems [15].

The IP system consists of a motor driven cart which is equipped with two quadrature encoders. One sensor measures the position (`track`) of the cart. The other sensor measures the angle (`angle`) of the pendulum attached to the cart. The pendulum freely moves in the vertical plane that is parallel to the track. The purpose of the IP control system is to maintain the pendulum upright by activating an appropriate controller which transmits a correct voltage output (`volts`) to the motor. The software part of a system consists of three components: a decision module, a safe controller and an array of external controllers. To run this system, the decision module picks a controller to run, sends `track` and `angle` to the controller, receives `track` from a controller and sends the voltage to a hardware cart. A safe controller always calculates a safe voltage just in case something is wrong with external controllers. The external controllers are controllers written by outsiders and not always reliable.

The safety requirements written in MaCS language as specified in [10] follows closely the requirement specified in [15]. The safety requirements indicate that if a voltage sent by a current controller is not safe or a cart is too close to the end of the track, the decision module should pick a voltage calculated by a safe controller. However, it omits one requirement indicating that if a controller fails to send a voltage value in time for three times in a row, it is considered a dead controller and needs to be killed. We have tried to write this specification using our existing MaCS and found that it is not quite simple.

With the MEDL-RE, we can easily specify this requirement as shown in Figure 2. Imported events `sendToCtrl`, `recvVolt` and `sendToIP` respectively indicate an instant when the decision module sends `track` and `angle` to a controller, an instant when the decision module receives a voltage value from a controller, and instant when the voltage value is being sent to the hardware. We assume that `sendToCtrl` and `sendToIP` always occur at the right instant, but `recvVolt` does not because external controllers are not always reliable.

A regular expression `dmTask` describes an order which three imported events should follow. An event `OK` indicates a success of `dmTask`, and an event `notOK` indicates that the decision module does not receive a voltage value in time. An event `deadCtrlOccur` indicates that the controller fails to send a voltage value three times in a row and needs to be killed.

5 Implementation Issues

To monitor and detect a regular expression, we need to translate a regular expression into a finite automaton. When a current event takes an automaton to a start stage, it would trigger a `startRE` event. If the automaton moves to a final stage when the event occurs, the event triggers a `success` event. Similarly, the event causing the automaton to get stuck triggers a `fail` event. For each `occur`(e, c), we store information of a current frequency and update the information according to the value of c and e.

The main difficulty of the implementation is to translate a regular expression into a finite automaton. Translating the regular expression R into an NFA could cause the monitor to be very inefficient in detecting a sequence of events of R while translating into a DFA could exponentially blow up the number of states. A number of previous researches have tried to solve this problem. We have chosen the algorithm by Aho, Sethi and Ullman [1] because the empirical result by Watson [17] suggests that this DFA contruction is fast and efficient. Their algorithm calculates a follow set for each character in the regular expression where the follow set is a set of characters that can follow after each character in the regular expression. The DFA is then constructed using the follow set. The details of the algorithm could be found in [1].

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, is a linear temporal logic [14] that 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 operator including regular expressions as a *syntactic sugar* to CTL [7]. Monitoring oriented Programming (MoP) [6] provides a monitoring architecture based on LTL [14] and extended regular expressions. They will also support RTL [2] and MTL [12] in a near future. Temporal Rover [8] is an architecture that helps a system do monitoring. Its specification language uses LTL [14] 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. They also have a counting operators which specify properties like event A must occur between 10 and 20 times until event B occur.

Comparing to the MEDL-RE based on LTL and regular expressions, FTL [3], MoP [6], and Temporal Rover [8] provides 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, our `startRE`, `success` and `fail` events are unique and none of them has a similar notion of a relevant set which facilitates users when they need to specify a regular expression for different components with different sets of events.

We also consider a Unix *RegExp* [16] command related to our work. *RegExp* is a software that returns one substring of a given string s that matches a given regular expression *exp*. It is related because their decision in dealing with multiple substrings matching a regular expression is similar to ours. If *RegExp* can match a regular expression with more than one substrings, it chooses using the rule "earliest then longest." It can choose the longest substring because its input strings are finite while our execution trace can be infinite. Our issue of a relevant set in Section 3.2 is also irrelevant to *RegExp* because all of its characters are relevant whereas not all of our events are relevant.

7 Conclusion and Future Work

We have presented our extension MEDL-RE, which incorporates regular expressions and its associated events into our MEDL language. The extension provides a more intuitive and simpler language to express complex dependencies between sequence of events, timing constraints, and a frequency of events during a time interval. Our MEDL-RE gives users a power to include or exclude events that are related or not related to a regular expression. Our events associated with a regular expression offers an ability to detect the instant the regular expression starts being observed and the instant we success or fail to find the regular expression. We believe that this extension facilitates expressing system requirements with a clearer and less error-prone language.

Since the implementation of extension MEDL-RE is still in progress, our future plan is to complete and test the implementation. This work may be furthered by extending our MEDL language to support a specification written in MTL and/or other logics. The MaCS system itself maybe extended to monitor distributed and parallel systems.

References

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

[2] R. Alur and T. A. Henzinger. Real-time logics: Complexity and expressiveness. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 390–401, Washington, D.C., 1990. IEEE Computer Society Press.

[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] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

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

[15] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, July/August 2001.

[16] H. Spencer. *RegExp.* http://arglist.com/regex/.

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