

# Foundations for Monitoring and Checking Reactive Systems

Mahesh Viswanathan<sup>1</sup> and Moonzoo Kim<sup>2</sup>

<sup>1</sup>CS Dept. University of Illinois at Urbana Champaign, US

<sup>2</sup>CSE Dept. Pohang University of Science and Technology, South Korea

## Abstract.

A run-time monitoring technique has become a promising methodology for higher system assurance by validating a current execution trace with regard to a given requirement specification. To formalize and understand the computational nature of run-time monitoring is a key to utilize this valuable technique. In this paper, we formalize the notion of run-time monitoring of reactive systems in terms of  $\omega$ -languages. Then, we introduce our prototype for run-time analysis of reactive systems, called *Monitoring and Checking (MaC)* architecture. Finally, we show that the class of languages specified by MEDL, which is a requirement specification language of the MaC architecture, is exactly the class of monitorable languages.

**Keywords:** run-time monitoring and checking, formal languages, computational complexity

## 1. Introduction

As the complexity of computing systems grows, there has been active research on the methods for verification and validation (V&V) of safety critical systems such as avionics and automobiles. Reactive nature of safety critical systems adds further difficulty to V&V activity for assuring correctness of the systems. Reactive systems are systems which perform ongoing interaction with an environment rather than generate output with given input. Computation is, therefore, typically seen as being non-terminating and the correctness of systems becomes harder to achieve due to its complex behavior and difficulty of testing. This difficulty for analyzing reactive systems promotes a *run-time monitoring* technique not only as a performance measurement method, but also as a promising complementary method for higher system assurance. The monitor examines interaction of systems, rather than a result at the end of computation and determines whether the behavior is correct.

The behavior of reactive systems can be modeled as an infinite sequence of letters from some finite alphabet. This sequence can be seen as either the sequence of program states visited by the reactive system, or as the sequence of request-response pairs that is generated by the system's interaction with its environment.

---

*Correspondence and offprint requests to:* Moonzoo Kim, CSE Dept., Pohang University of Science and Technology, Pohang, South Korea, e-mail: moonzoo@postech.ac.kr

Certifying the correctness of reactive systems, therefore, involves checking to see if the set of execution sequences of the reactive system satisfies certain constraints/properties.

Past research in monitoring [Sch00] tried to identify the class of monitorable properties with the class of *safety* properties. In this paper, we show that the class of monitorable properties is a strict subset of the class of safety properties. Furthermore, we identify the class of languages that can be monitored with the class  $\Pi_1^0$  in the Arithmetic hierarchy. We also introduce a class of automata that can be used to specify these languages, and that can serve as monitors for such languages. Finally, we prove that the class of languages defined by MEDL, which is a requirement specification language of the Monitoring and Checking (MaC) architecture, is exactly the class of monitorable languages.<sup>1</sup>

Section 2 shows that the class of languages run-time monitoring can determine, say  $\mathcal{M}$ , is a strict subset of the class of safety languages. Section 3 describes the class of monitorable languages  $\mathcal{M}$  in the Arithmetic hierarchy. Section 4 introduces the model of finite state machines with storage which can specify  $\mathcal{M}$ . Section 5 proves that run-time monitoring and checking problem is NP-complete with a sufficiently expressive requirement specification language. Section 6 briefly describes the MaC architecture and shows that the class of languages that MEDL defines is exactly  $\mathcal{M}$ . Finally, we enumerate related works in Section 7 and Section 8 concludes this paper.

## 2. A Class of Monitorable Languages $\mathcal{M}$

It is clear that run-time monitoring cannot evaluate liveness properties because a monitor decides the correctness of system based on what has been observed. We generally presume that the class of properties which run-time monitoring can evaluate is safety properties. In this section, however, we study the class of properties run-time monitoring can evaluate more precisely.

### 2.1. Notations

We use standard notations of  $\omega$ -languages.  $\Sigma$  is a finite alphabet. The set of finite words over  $\Sigma$ , including the empty word  $\epsilon$ , is denoted by  $\Sigma^*$ , while the set of  $\omega$ -words is  $\Sigma^\omega$ ;  $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ . A subset of  $\Sigma^*$  is called a finitary language, and a subset of  $\Sigma^\omega$  is an infinitary language (or  $\omega$ -language).

For a (finite or infinite) word  $\alpha$ ,  $\alpha(i)$  (or  $\alpha_i$ ) denotes the  $(i+1)$ st letter of  $\alpha$ . Segments of words are denoted as follows:  $\alpha(m, n) = \alpha(m) \cdot \alpha(m+1) \cdots \alpha(n-1)$  and  $\alpha(m, \omega) = \alpha(m) \cdot \alpha(m+1) \cdots$ . The concatenation of a finite word  $u$  with another word (finite or infinite)  $\alpha$ ,  $u \cdot \alpha$ , is defined by  $u \cdot \alpha(i) = u(i)$  if  $i < |u|$  and  $u \cdot \alpha(i) = \alpha(i - |u|)$  otherwise. A finite word  $u$  is said to be a prefix of another word  $\alpha$  if there is  $\beta \in \Sigma^\infty$  such that  $u \cdot \beta = \alpha$ .  $\text{pref}(\alpha)$  is the set of all finite prefixes of  $\alpha$ , and for a (finite or infinite) language  $L$ ,  $\text{pref}(L) = \cup_{\alpha \in L} \text{pref}(\alpha)$ .

### 2.2. Safety Languages and Monitorable Languages

Informally speaking, safety languages are languages that require that nothing bad happens during an execution; if an execution is faulty, then the monitor should be able to reject it after looking at a *finite* prefix. Safety languages are formally defined by [AS85] as

**Definition 1. (Safety language)** A language  $L \subseteq \Sigma^\omega$  is a safety language if for every  $\sigma \in \Sigma^\omega$ ,  $\sigma \in L$  if and only if  $\forall i \exists \beta \in \Sigma^\omega (\sigma(0, i) \cdot \beta \in L)$

It is clear from Definition 1 that a monitorable property is a safety language. A safety language, however, is *not* necessarily a monitorable language. The definition of safety language makes no computational assumptions. It is possible to define a language that is a safety language, but which is unlikely monitorable. For example, safety closure of the halting problem is a safety language but *not* a monitorable language.

**Example 1.** Let  $\Sigma = \{0, 1, a, b\}$ . Consider a finite language  $H_* = \{x \cdot a \cdot y \mid x, y \in \{0, 1\}^*\}$ , the Turing Machine encoded by  $x$  halts on input  $y$ . We define a language  $H_\omega = H_* \cdot b^\omega \cup \{0, 1\}^* \cdot a \cdot \{0, 1\}^\omega \cup \{0, 1\}^\omega$ .

<sup>1</sup> In this paper, we use two terms “property” and “language” for the same meaning depending on its context.

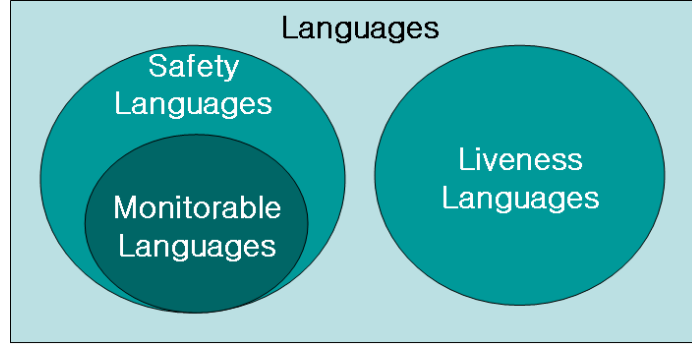


Fig. 1. Monitorable languages

The language  $H_\omega$ , defined above is a safety language. In order to see this, we only need to observe that for any execution not in  $H_\omega$ , there is a finite prefix when this violation can be detected. Executions not in  $H_\omega$  are those that are not in the “right format”, or where the finite prefix before the sequence of  $b$ 's is not in  $H_*$ ; in both cases there is a finite prefix that provides evidence of the execution not being in the language.

However, in order to detect that an execution  $\sigma$  is not in  $H_\omega$ , we have to check for membership in  $H_*$ . Since membership in  $H_*$  (or the Halting problem) is not decidable, it is impossible for us to design monitors that would be able to detect a violation of this language. This suggests that the class of *monitorable languages* is a strict subset of the class of safety languages (see Fig1); they should be such that sequences not in the languages should be recognizable by a Turing Machine, after examining a finite prefix. Therefore, we can define a monitorable language as follows.

**Definition 2. (Monitorable language)** A language  $L \subseteq \Sigma^\omega$  is said to be *monitorable* if and only if  $L$  is a safety language and  $\Sigma^* \setminus \text{pref}(L)$  is recursively enumerable. The class of monitorable languages is denoted by  $\mathcal{M}$ .

### 3. $\mathcal{M}$ in the Arithmetic Hierarchy

In our study of  $\omega$ -languages, we will find it useful to discuss definability relative to classical hierarchies in recursion theory and descriptive set theory. Such hierarchies have been extensively studied in the context of formal languages [Sta86, Tho90, Sta97]. In the language theoretic context, the usual set-up of these hierarchies is modified slightly. The relations that we consider are not defined over natural numbers and functions over natural numbers, but are rather over the finite and infinite words over a finite alphabet. While this change is irrelevant due to the presence of standard recursive encodings from  $\Sigma^*$  to  $\mathbb{N}$ , it provides a cleaner presentation for questions arising in automata theory.

A relation  $R$  is said to be *finitary* over  $\Sigma$ , if  $R \subseteq (\Sigma^*)^m$ . We write  $Ru_1u_2 \dots u_m$  instead of  $(u_1, u_2, \dots, u_m) \in R$ . We will define our hierarchy in terms of a class of finitary relations,  $\mathcal{C}$ . This class  $\mathcal{C}$  will be assumed to be closed under boolean operations.

First, we will consider finitary languages over  $\Sigma$ . A language  $L \subseteq \Sigma^*$  is said to be in  $\Sigma_n^0(\mathcal{C})$  if and only if for some relation  $R \in \mathcal{C}$ ,

$$L = \{u \mid \exists v_1 \forall v_2 \dots Q_n v_n R v_1 v_2 \dots v_n u\}$$

where  $Q_n$  is either  $\exists$  (if  $n$  is odd) or  $\forall$  (if  $n$  is even). The languages in  $\Pi_n^0(\mathcal{C})$  are defined analogously.  $L \subseteq \Sigma^*$  is in  $\Pi_n^0(\mathcal{C})$  if and only if for some relation  $R \in \mathcal{C}$ ,

$$L = \{u \mid \forall v_1 \exists v_2 \dots Q_n v_n R v_1 v_2 \dots v_n u\}$$

where  $Q_n$  is either  $\forall$  (if  $n$  is odd) or  $\exists$  (if  $n$  is even).

The hierarchy of infinitary languages over  $\mathcal{C}$  is defined as follows. A language  $L \subseteq \Sigma^\omega$  is in  $\Sigma_n^0(\mathcal{C})$  if and only if for some  $R \in \mathcal{C}$ ,

$$L = \{\alpha \mid \exists v_1 \forall v_2 \dots Q_{n-1} v_{n-1} Q'_n i R v_1 v_2 \dots v_{n-1} \alpha(0, i)\}$$

where, once again,  $Q_{n-1}$  and  $Q'_n$  are quantifiers, and  $i$  is a natural number. The languages in  $\Pi_n^0(\mathcal{C})$  are defined similarly in terms of logical formulae with alternating quantifiers, with the leading quantifier being  $\forall$ . Though we use the same notation for the hierarchy of infinitary languages, as in the case of finitary languages, it will often be clear from the context which hierarchy we are referring to.

By instantiating  $\mathcal{C}$  to specific families, we obtain the classical hierarchies from recursion theory and descriptive set theory. If  $\mathcal{C}$  is taken to be the class of recursive relations (*REC*), then we get the *arithmetic hierarchy*. For finitary languages,  $\Sigma_1^0(\text{REC})$  coincides with the class of recursively enumerable (R.E.) languages, while  $\Pi_1^0(\text{REC})$  is the class of co-R.E. languages. For notational convenience, we will denote the classes  $\Sigma_n^0(\text{REC})$  and  $\Pi_n^0(\text{REC})$ , simply as,  $\Sigma_n^0$  and  $\Pi_n^0$ , respectively.

Now we are ready to describe the class of monitoring languages  $\mathcal{M}$  in terms of the Arithmetic hierarchy.

**Proposition 3.1.**  $\mathcal{M} = \Pi_1^0$

*Proof.* [ $\mathcal{M} \Rightarrow \Pi_1^0$ ] Consider  $L \in \mathcal{M}$ . From the definition of  $\mathcal{M}$ , we know that  $\Sigma^* \setminus \text{pref}(L)$  is recursively enumerable. Therefore, there is a recursive relation  $R$  such that  $u \in \Sigma^* \setminus \text{pref}(L)$  if and only if  $\exists v Rvu$ . In other words,  $u \in \text{pref}(L)$  if and only if  $\forall v R'vu$ , where  $R' = \neg R$ . Furthermore, we know that  $L$  is a safety language, which implies that  $L \in \text{adh}(\text{pref}(L))$  where  $\text{adh}(L) = \{\alpha \in \Sigma^\omega \mid \text{pref}(\alpha) \subseteq \text{pref}(L)\}$  [Vis00]. Hence,  $\alpha \in L$  if and only if  $\forall i \alpha(0, i) \in \text{pref}(L)$  if and only if  $\forall i \forall v R'v\alpha(0, i)$ . By contracting the quantifiers we can see that  $L \in \Pi_1^0$ .

[ $\Pi_1^0 \Rightarrow \mathcal{M}$ ] Let  $L \in \Pi_1^0$ . Hence  $\alpha \in L$  if and only if  $\forall i R\alpha(0, i)$  for some recursive relation  $R$ . From the definition of safety language (see Sect 2.2), it is clear that  $L$  is a safety language. Also,  $u \in \Sigma^* \setminus \text{pref}(L)$  if and only if  $\neg Ru$ . Thus  $\Sigma^* \setminus \text{pref}(L)$  is recursively enumerable, and  $L \in \mathcal{M}$ .  $\square$

## 4. $\omega$ -Automata with Storage

Finite state machines on infinite words, are very similar to those which accept finite words. On an  $\omega$ -word  $\alpha$ , the machine works as if  $\alpha$  were a very large finite word. The only difference is the criteria that these machines use to accept a language (clearly, acceptance by final state cannot be used).

The general notion of an automaton on  $\omega$ -words, using some kind of storage, was first introduced and studied Engelfriet and Hoogeboom [EH93]. We use definitions and concepts described there, to develop our theory. Before defining finite state machines on  $\omega$ -words formally, we first define the notion of a storage type, and give an example.

**Definition 4.1.** A storage type is a 5-tuple  $X = (C, C_0, P, F, \llbracket \cdot \rrbracket)$ , where

- $C$  is a set of storage configurations,
- $C_0 \subseteq C$  is a set of initial storage configurations,
- $P$  is a set of predicate symbols,
- $F$  is a set of function symbols, and
- $\llbracket \cdot \rrbracket$  is a function that defines the semantics of the predicate and function symbols. For each  $p \in P$ ,  $\llbracket p \rrbracket : C \rightarrow \{\mathbf{true}, \mathbf{false}\}$ , and for each  $f \in F$ ,  $\llbracket f \rrbracket : C \rightarrow C$ , is a partial function.

The set of all Boolean expressions over  $P$ , built using connectives  $\wedge, \vee$ , and  $\neg$ , constants  $\{\mathbf{true}, \mathbf{false}\}$ , and the predicates in  $P$ , is denoted by  $BE(P)$ . The function  $\llbracket \cdot \rrbracket$  is extended to  $BE(P)$  in the standard way.  $\llbracket \cdot \rrbracket$  is also extended to finite words over  $F$ , by interpreting concatenation as function composition. In other words,  $\llbracket f \cdot \varphi \rrbracket = \llbracket \varphi \rrbracket \circ \llbracket f \rrbracket$ , where  $\varphi \in F^*$  and  $f \in F$ .

**Example 2.** The storage type, *accumulator*, is  $AC = (\mathbb{N}, \{0\}, \{zero\}, \{+k, -k \mid k \in \mathbb{N}\}, \llbracket \cdot \rrbracket)$ . It is the storage type of integers with a test for zero, and ability to add and subtract constants. More precisely,

$$\begin{aligned} \llbracket zero \rrbracket(c) &= \mathbf{true} \text{ if and only if } c = 0 \\ \llbracket +k \rrbracket(c) &= c + k \\ \llbracket -k \rrbracket(c) &= c - k, \text{ if } c \geq k, \text{ and undefined otherwise.} \end{aligned}$$

We will now define the notion of the product of storage types. It is a way obtaining a new storage type that combines two storage types and uses them independently.

**Definition 4.2.** Let  $X_1 = (C_1, C_{10}, P_1, F_1, \llbracket \cdot \rrbracket_1)$  and  $X_2 = (C_2, C_{20}, P_2, F_2, \llbracket \cdot \rrbracket_2)$  be two storage types with  $P_1 \cap P_2 = \emptyset$  and  $F_1 \cap F_2 = \emptyset$ . The product of these two storage types,  $X_1 \times X_2$ , is the type  $(C, C_0, P, F, \llbracket \cdot \rrbracket)$ , where  $C = C_1 \times C_2$ ,  $C_0 = C_{10} \times C_{20}$ ,  $P = P_1 \cup P_2$  and  $F = F_1 \cup F_2$ . The function  $\llbracket \cdot \rrbracket$  is then defined naturally, as follows.

$$\begin{aligned} \llbracket p \rrbracket(c_1, c_2) &= \begin{cases} \llbracket p \rrbracket_1(c_1) & \text{if } p \in P_1 \\ \llbracket p \rrbracket_2(c_2) & \text{if } p \in P_2 \end{cases} \\ \llbracket f \rrbracket(c_1, c_2) &= \begin{cases} (\llbracket f \rrbracket_1(c_1), c_2) & \text{if } f \in F_1 \\ (c_1, \llbracket f \rrbracket_2(c_2)) & \text{if } f \in F_2 \end{cases} \end{aligned}$$

We will often use the above definition to get finitely many copies of the same storage type. In such a case, we first rename the predicate and function symbols of the storage type, by adding subscripts, and then taking repeated products. The  $n$ -fold product ( $n \geq 1$ ) of a storage type  $X$  will be denoted by  $X^n$ . In order to extend the definition consistently, we take  $X^0 = (\{c\}, \{c\}, \emptyset, \emptyset, \emptyset)$ .

We are now ready to define automata with storage type  $X$ . We will consider only one acceptance condition for such machines (see Def 4.4)<sup>2</sup>

**Definition 4.3.** Let  $X = (C, C_0, P, F, \llbracket \cdot \rrbracket)$  be a storage type. An  $X$ -automata is a 5-tuple  $\mathcal{A} = (Q, \Sigma, \delta, q_0, c_0)$ , where

- $Q$  is a finite set of states,
- $\Sigma$  is a finite input alphabet,
- $\delta$  is the transition function, which is a *finite* subset of  $Q \times (\Sigma \cup \{\epsilon\}) \times BE(P) \times Q \times F^*$ ,
- $q_0 \in Q$  is the initial state, and
- $c_0 \in C_0$  is the initial storage configuration.

The instantaneous description of such a machine  $\mathcal{A}$ , is a tuple  $(q, \alpha, i, c) \in Q \times \Sigma^\omega \times \mathbb{N} \times C$ , where  $q$  is the current state of the machine,  $\alpha$  is the input to the machine,  $i$  is the position of the symbol being currently scanned, and  $c$  is the current configuration. In one step the machine either reads a symbol from the input or makes a “silent” transition, according to the transition function  $\delta$ . More precisely, we say  $(q, \alpha, i, c) \vdash (q', \alpha, i', c')$ , if there exists a transition  $(q, a, \varphi, q', h)$ , such that  $\llbracket \varphi \rrbracket(c) = \mathbf{true}$ ,  $\llbracket h \rrbracket(c)$  is defined, and  $\llbracket h \rrbracket(c) = c'$ . Furthermore, we require that, either  $a = \epsilon$  and  $i = i'$ , or  $a = \alpha(i)$  and  $i' = i + 1$ . An infinite run of the automaton  $\mathcal{A}$ , on an input  $\alpha$ , is an infinite sequence  $\langle I_i \rangle_{i \in \mathbb{N}}$  of instantaneous descriptions, such that  $I_0 = (q_0, \alpha, 0, c_0)$ ,  $I_i \vdash I_{i+1}$ , for each  $i \in \mathbb{N}$ , and for every  $j \in \mathbb{N}$ , there is a  $k$  such that  $I_k$  is scanning a position beyond  $j$ .

**Definition 4.4.** An  $\omega$ -word,  $\alpha \in \Sigma^\omega$ , is said to be accepted by an  $X$ -automaton  $\mathcal{A}$ , if there is an infinite run of the automaton on the input  $\alpha$ . The language accepted by  $\mathcal{A}$ ,  $L_{\mathcal{A}}$ , is the set of all  $\omega$ -words accepted by  $\mathcal{A}$ .

The above definition of acceptance coincides with Landweber’s [Lan69] 1’-acceptance and with the “always” acceptance of [EH93].

An automaton  $\mathcal{A}$  is *deterministic* if for any state and storage configuration there is at most one possible next state and storage configuration. More formally, for any two tuples  $(q_1, a_1, \varphi_1, q'_1, h_1)$  and  $(q_2, a_2, \varphi_2, q'_2, h_2)$  in  $\delta$ , with  $q_1 = q_2$ , either  $a_1 \neq a_2$  and  $a_1, a_2 \neq \epsilon$ , or  $\llbracket \varphi_1 \wedge \varphi_2 \rrbracket(c) = \mathbf{false}$ , for every  $c \in C$ . Automata of particular interest to us will be what are called *real-time* automata.  $\mathcal{A}$  is a real-time automata if it has no  $\epsilon$ -transition, i.e.,  $\delta \subseteq Q \times \Sigma \times BE(P) \times Q \times F^*$ . A slightly more general class of automata than real-time automata is the *finite delay* automata. An automaton  $\mathcal{A}$  is said to be finite-delay, if there is no infinite run of the automaton on a finite word.

The class of  $\omega$ -languages accepted by  $X$ -automata will be denoted by  $X\mathcal{L}$ ;  $X^*\mathcal{L} = \cup_n X^n\mathcal{L}$ , where  $X^n$  is the  $n$ -fold product of the storage type  $X$ . The prefixes *d*-, *r*-, and *f*- will be used to denote the class of languages accepted by deterministic, real-time, and finite delay automata respectively. Similarly the prefix *dr*- (and *df*-) will be used for languages accepted by automata that are both deterministic and real-time (deterministic and finite delay).

Before presenting the automata theoretic characterization of  $\mathcal{M}$ , we define a storage type that will play an important role. This is the type of storage where one has finitely many integer locations that one can

<sup>2</sup> For a discussion of the relative power of the various other acceptance conditions, readers are directed to [Sta86, Tho90].

manipulate using addition, subtraction, multiplication and division. We will then prove a result relating the powers of real-time and finite delay automata with such a storage.

**Definition 4.5.** The storage type of  $m$  integer variables  $\mathbb{N}_m$  is given by  $\mathbb{N}_m = (C, C_0, P, F, \llbracket \cdot \rrbracket)$ .  $C = \mathbb{N}^m$  is the set of  $m$ -tuples of natural numbers, and  $C_0 = \langle 0, \dots, 0 \rangle$ .  $P$  consists of predicates  $zero_i$ , which test if the  $i$ th element of the current configuration is 0, i.e.,  $\llbracket zero_i \rrbracket(\langle c_0, \dots, c_{m-1} \rangle) = \mathbf{true}$  if and only if  $c_i = 0$ . There are various operations that one can perform on these configurations; one can add, subtract, and multiply integers to some element of the tuple, find the quotient or remainder when dividing an entry by an integer, and also add and subtract one entry in the tuple to another. The operation  $ADR_{i,j}$  (add register) adds the  $i$ th entry to the  $j$ th entry;  $SBR_{i,j}$  (subtract register) subtracts the  $i$ th entry from the  $j$ th entry.  $ADC_{i,k}$  adds constant  $k$  to  $i$ th entry; similarly,  $SBC_{i,k}$  and  $MLC_{i,k}$  subtract and multiply constants  $k$ , while  $QC_{i,k}$  and  $RM C_{i,k}$  find the quotient and remainder when divided by  $k$ .

As usual,  $\mathbb{N}_m\mathcal{L}$  is the class of languages accepted by automata with storage type  $\mathbb{N}_m$ . By  $\mathbb{N}_*\mathcal{L}$  we denote the class of languages  $\cup_m \mathbb{N}_m\mathcal{L}$ .

**Proposition 4.1.**  $f\text{-}\mathbb{N}_*\mathcal{L} \subseteq r\text{-}\mathbb{N}_*\mathcal{L}$  and  $df\text{-}\mathbb{N}_*\mathcal{L} \subseteq dr\text{-}\mathbb{N}_*\mathcal{L}$ .

*Proof.* The difference between a real-time automaton and a finite delay automaton is that the real-time automaton may not have the “time” to do the computation performed by the finite delay machine, since the real-time machine must process an input symbol at each time instant. The idea behind the proof is if the real-time machine can simulate a buffer, then it has enough time to do everything done by the finite delay machine. The real-time automaton, then, reads an input symbol every time and puts it into the buffer, while the actual computation is then performed on the buffered input.

We will basically show that  $f\text{-}\mathbb{N}_m\mathcal{L} \subseteq r\text{-}\mathbb{N}_{m+2}\mathcal{L}$ , where the two extra integer locations will be used by the real-time machine to simulate a buffer. It is straight-forward to see that the operations for manipulating a queue can be performed in one step using two locations, one storing the contents of the queue and the other storing some measure of the number of elements in the queue. In order to simplify the description, we assume that  $\Sigma = \{0, 1\}$ , and we use a program-like notation to describe the various operations. Let the two locations be  $q$  and  $l$ . Initially  $q = 0$  and  $l = 1$ . Enqueuing the element  $i$  is done by the following steps:  $q = q + l$  (if  $i = 1$ ) or  $q = q$  (if  $i = 0$ ), and  $l = 2l$ . Dequeuing is  $q = q/2$  and  $l = l/2$ , where  $/2$  gives the quotient when divided by 2. Reading the element at the head of the queue is just looking at the remainder when  $q$  is divided by 2. All of these steps can be done by the storage type  $\mathbb{N}_m$ . Notice that this construction preserves determinism if the original finite delay machine was deterministic. Therefore,  $f\text{-}\mathbb{N}_*\mathcal{L} \subseteq r\text{-}\mathbb{N}_*\mathcal{L}$  and  $df\text{-}\mathbb{N}_*\mathcal{L} \subseteq dr\text{-}\mathbb{N}_*\mathcal{L}$ .  $\square$

We are now ready to provide an automata theoretic characterization of  $\mathcal{M}$ .

**Theorem 4.1.** The following classes of  $\omega$ -languages are equivalent.

1.  $\mathcal{M} = \Pi_1^0$
2.  $df\text{-}\mathbb{N}_*\mathcal{L}$
3.  $dr\text{-}\mathbb{N}_*\mathcal{L}$

*Proof.* [(1)  $\Rightarrow$  (2)] For a language  $L \in \Pi_1^0$ , we know that  $\alpha \in L$  if and only if  $\forall i R\alpha(0, i)$ , where  $R$  is a recursive language. Since  $R$  is a recursive language, there exists a deterministic finite delay  $\mathbb{N}_m$ -automaton,  $\mathcal{A}$ , that has runs on exactly the same finite words as  $R$ . It is easy to see that the language accepted by  $\mathcal{A}$  is  $L$ .

[(2)  $\Rightarrow$  (3)] This follows from Proposition 4.1

[(3)  $\Rightarrow$  (1)] Let  $\mathcal{A}$  be the deterministic real-time automaton that accepts  $L$ . Observe that since  $\mathcal{A}$  is real-time, it cannot distinguish between infinite runs that read the whole input and runs that do not read the whole input (because there are no such runs). Thus  $\alpha \in L$  if and only if  $\forall i \alpha(0, i)$  has a run. Hence,  $L \in \Pi_1^0$ .  $\square$

## 5. Computational Complexity of Checking Behavior of Systems

In order to check the correctness of system execution, we need to describe constraints/properties for the system in a requirement specification language. The characteristics of the requirement specification language

can affect the computational complexity of validating executions. [HR02] develops a linear time monitoring algorithm for safety formulae written in past time LTL. [SS97] shows the monitoring overhead due to non-determinism in requirements in their monitoring system, called Supervisor, and provides heuristics to decrease the overhead. In this section, we will discuss the computational complexity of monitoring and checking for requirements written in process algebra.

Run-time monitoring and checking of a reactive system can be thought of as a *trace validity problem* where a trace is generated from execution of the program. The trace validity problem is a membership checking problem of deciding whether a given trace is in the set of valid traces. For sufficiently expressive requirement specification languages such as CCS [Mil89], this problem turns out to be NP-complete. We formulate the trace validity problem using the notation of [Mil89].

## 5.1. Notations

We will denote the *i*th character in a string  $x$  by  $x^{(i)}$ .  $\mathcal{A}$  is an set of names  $a, b, c, \dots$ . Then,  $\overline{\mathcal{A}}$  is the set of *co-names*  $\bar{a}, \bar{b}, \bar{c}, \dots$ ;  $\mathcal{A}$  and  $\overline{\mathcal{A}}$  are disjoint and are in bijection via  $(\bar{\cdot})$ ; we declare  $\bar{\bar{a}} = a$ .  $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$  denotes the set of labels. We also introduce a distinguished silent action  $\tau \notin \mathcal{L}$ . We set  $Act = \mathcal{L} \cup \{\tau\}$ .

**Definition 3 (Syntax).** The set of processes is defined by

$$P ::= Nil \mid \alpha.P \mid P + Q \mid P \parallel Q \mid P \setminus L$$

where  $L \subseteq \mathcal{L}$  and  $\alpha \in Act$ .

**Definition 4 (Operational semantics).** The labeled transition relation  $\xrightarrow{\alpha}$  between two processes is defined by the following rules. In the following rules,  $\alpha \in Act$ ,  $l \in \mathcal{L}$ , and  $L \subseteq \mathcal{L}$ .

$$\begin{aligned} [Prefix] & \frac{}{\alpha.P \xrightarrow{\alpha} P} \\ [Choice] & \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} P' \quad P + Q \xrightarrow{\alpha} Q'} \\ [Parallel] & \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\alpha} Q' \quad P \xrightarrow{l} P', Q \xrightarrow{\bar{l}} Q'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q \quad P \parallel Q \xrightarrow{\alpha} P \parallel Q' \quad P \parallel Q \xrightarrow{\tau} P' \parallel Q'} \\ [Restriction] & \frac{P \xrightarrow{\alpha} P'}{P \setminus L \xrightarrow{\alpha} P'} \text{ where } \alpha \notin L \cup \overline{L} \end{aligned}$$

Note that  $\xrightarrow{\tau}$  is *not* restricted by the [Restriction] rule.

**Definition 5.** Given processes  $P$  and  $P'$ , and  $\alpha \in \mathcal{L}$ , we say that  $P \xrightarrow{\alpha} P'$  if  $P(\xrightarrow{\tau})^* \xrightarrow{\alpha} (\xrightarrow{\tau})^* P'$ , where  $(\xrightarrow{\tau})^*$  is the transitive reflexive closure of  $\xrightarrow{\tau}$ .

## 5.2. Trace Validity Problem

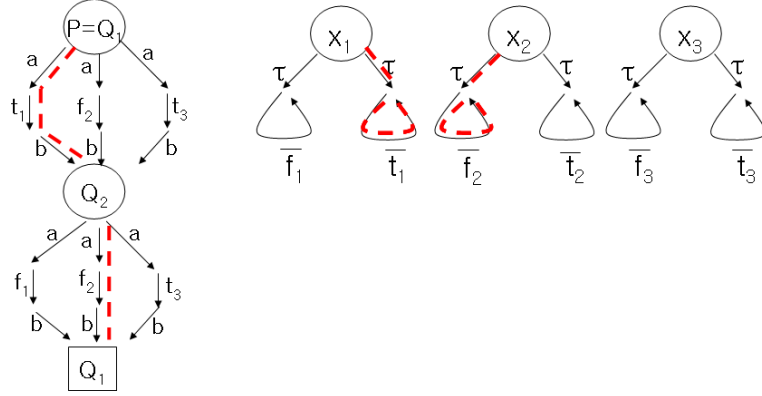
In this section, we formulate the trace validity problem using the notations in Sect 5.1.

**Definition 6 (Valid trace).** A string  $s \in \mathcal{L}^*$ , of length  $n$ , is said to be a valid trace of a process  $P$ , if there exist processes  $P_0, P_1, \dots, P_n$ , such that  $P \equiv P_0$ , and  $P_{(i-1)} \xrightarrow{s^{(i)}} P_i$ , for all  $i \in \{1, \dots, n\}$

Then, the Trace Validity Problem is formally defined as follows:

*Input* A process  $P$  and a string  $s \in \mathcal{L}^*$ .  
*Output* Is  $s$  a valid trace of  $P$ ?

**Theorem 5.1.** The trace validity problem is NP-complete.



**Fig. 2.** Processes  $P, X_1, X_2,$  and  $X_3$  for  $P(\varphi)$  where  $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$

*Proof.* To prove hardness, we reduce 3SAT to the trace validity problem. We are given a formula  $\varphi$  in conjunctive normal form with variables  $x_1, \dots, x_n$  and clauses  $C_1, \dots, C_m$ , each with three literals. We construct a process  $P(\varphi)$  and a string  $s(\varphi)$  such that  $s(\varphi)$  is a valid trace of  $P(\varphi)$  if and only if the formula  $\varphi$  is satisfiable.

For each  $i$ , define processes,  $X_i$ , as follows,

$$\begin{aligned} X_i &\stackrel{\text{def}}{=} \tau.F_i + \tau.T_i \\ F_i &\stackrel{\text{def}}{=} \bar{f}_i.F_i \\ T_i &\stackrel{\text{def}}{=} \bar{t}_i.T_i \end{aligned}$$

In our reduction, these processes express a truth value assignment to the variables. If the  $X_i \xrightarrow{\tau} F_i$  then it expresses the fact that under this assignment the variable  $x_i$  gets the value **false**, and if  $X_i \xrightarrow{\tau} T_i$  then it means that the variable  $x_i$  gets the value **true**.

In addition to these processes, we define another process  $P$ . The idea is that  $P$  will deadlock, when it runs concurrently with the processes  $X_i$ , if and only if the truth assignment defined (as above) by the processes  $X_i$  is not a satisfying truth assignment for the formula  $\varphi$ . In order to define the process  $P$ , we assume that  $C_i \equiv l_{i,1} \vee l_{i,2} \vee l_{i,3}$  for  $i \in \{1, \dots, m\}$  and  $j \in \{1, 2, 3\}$  in following formulae.

$$\begin{aligned} P &\stackrel{\text{def}}{=} Q_1 \\ Q_1 &\stackrel{\text{def}}{=} a.L_{1,1} + a.L_{1,2} + a.L_{1,3} \\ &\vdots \\ Q_i &\stackrel{\text{def}}{=} a.L_{i,1} + a.L_{i,2} + a.L_{i,3} \\ L_{i,j} &\stackrel{\text{def}}{=} \begin{cases} f_k.L'_{i,j} & \text{if } l_{i,j} \equiv \neg x_k \\ t_k.L'_{i,j} & \text{if } l_{i,j} \equiv x_k \end{cases} \\ L'_{i,j} &\stackrel{\text{def}}{=} b.Q_{i+1} \\ &\vdots \\ L'_{m,j} &\stackrel{\text{def}}{=} b.Q_1 \end{aligned}$$

The process  $P(\varphi)$  is thus  $(P || X_1 || \dots || X_n) \setminus \{t_1, f_1, \dots, t_n, f_n\}$ . The requirement this process has is that, for any  $i, j$ , the transition  $L_{i,j} \rightarrow L'_{i,j}$  can be taken if and only if the literal  $l_{i,j}$  gets the truth value **true** under the truth assignment defined by the processes  $X_1, \dots, X_n$ . Hence,  $Q_i \rightarrow^* Q_{i+1}$  can take place if and only if one of the literals in the clause  $C_i$  gets the truth value **true** under the assignment described by  $X_1, \dots, X_n$ . Thus it can be seen that  $abab \dots ab$  is a valid trace of  $P(\varphi)$  if and only if  $\varphi$  has a satisfying assignment.

For example, suppose we have  $\varphi$  as  $(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3)$ . Then, we can construct a process  $P(\varphi)$  as depicted in Fig 2. One assignment strategy for satisfying  $\varphi$  is  $x_1 = \mathbf{true}$ ,  $x_2 = \mathbf{false}$ . This assignment corresponds to the process behavior indicated as dotted lines in Fig 2.

To prove completeness, we prove that trace validity problem belongs to NP. We can view a process  $P$  as

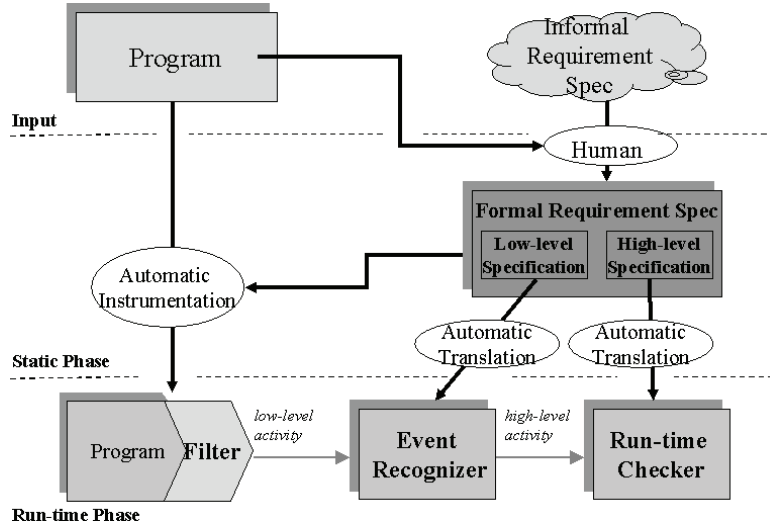


Fig. 3. Overview of the MaC architecture

a labeled transition graph  $G_P$  over a set of label  $\mathcal{L}$  rooted at the node  $n_P$ . For a given process  $P$  and a string  $s \in \mathcal{L}^*$ , we choose a path  $p$  corresponding to  $s$  from  $n_P$  be the certificate. Checking can be accomplished in polynomial time by traversing  $G_P$  from  $n_P$  following  $p$ .  $\square$

The main cause for NP-completeness in the trace validity problem is non-determinism. We should be careful to define or use a requirement specification language so that trace validation against properties is tractable (see Sect 6).

## 6. The Monitoring and Checking (MaC) Architecture

The *Monitoring and Checking (MaC)* architecture [KVK<sup>+</sup>04, Kim01] is a framework for monitoring and checking a running system with the aim of ensuring that the target program is running correctly with respect to a formal requirement specification. Section 6.1 describes overview of the MaC architecture. Section 6.2 describes specification languages of the MaC architecture.

### 6.1. Overview

The overview of the MaC architecture is depicted in Fig 3. The MaC architecture consists of three components: *filter*, *event recognizer*, and *run-time checker*. The filter extracts low-level information (such as values of program variables and time when variables change their values) from the instrumented target program. The filter sends this information to the event recognizer, which detects primitive events and conditions where primitive events are changes of values, beginnings of functions, and endings of functions and primitive conditions are boolean variables or boolean statements composed by primitive typed variables (see Sect 6.2.1 for more details). These events and conditions are then sent to a run-time checker. The run-time checker determines whether the current execution history satisfies the requirement specification.

Monitoring and checking as well as target program instrumentation are automatically performed from a given requirement specification, which makes the run-time analysis rigorous. In addition, monitoring program-dependent low-level behavior and checking high-level behavioral requirements are separated. This separation allows the specification of high-level requirements independent of the implementation since implementation specific details are confined to the low-level specification. Furthermore, this modularity of the MaC architecture and well-defined interfaces among the components makes it easy to extend the architecture to incorporate third-party tools. We have demonstrated the effectiveness of the MaC architecture using Java-MaC, a prototype implementation of the MaC architecture for Java programs, through several case studies [BGK<sup>+</sup>02, KLS<sup>+</sup>02].

---

$\langle C \rangle$	$::=$	$c \mid \mathbf{defined}(\langle C \rangle) \mid [\langle E \rangle, \langle E \rangle] \mid !\langle C \rangle \mid \langle C \rangle \&\& \langle C \rangle \mid \langle C \rangle \parallel \langle C \rangle \mid \langle C \rangle \Rightarrow \langle C \rangle$
$\langle E \rangle$	$::=$	$e \mid \mathbf{start}(\langle C \rangle) \mid \mathbf{end}(\langle C \rangle) \mid \langle E \rangle \&\& \langle E \rangle \mid \langle E \rangle \parallel \langle E \rangle \mid \langle E \rangle \mathbf{when} \langle C \rangle$

---

**Table 1.** The syntax of conditions and events

## 6.2. Specification Languages

In this section, we give a brief overview of the formal specification languages used to describe specifications. The language for low-level specification is called Primitive Event Definition Language (PEDL). PEDL is used to define what information is sent from the filter to the event recognizer, and how it is transformed into events used in high-level specification by the event recognizer. High-level specifications are written in Meta Event Definition Language (MEDL). This separation ensures that the architecture is portable to different implementation languages and specification formalisms. Before presenting the two languages, we first define the notions of *event* and *condition*, which are fundamental to the MaC languages.

### 6.2.1. Events and Conditions

The MaC architecture assumes that it is possible to observe the behavior of the target system and evaluate the observed behavior to check whether required properties are satisfied or not. The observation is based on the occurrence of “interesting” state change in the target system. We use the notions of event and condition to capture interesting state changes.

*Events* occur instantaneously during the system execution, whereas *conditions* represent information that holds for a duration of time. For example, an event denoting return from method `RaiseGate()` occurs at the instant the control returns from the method, while a condition (`position == 2`) holds as long as the variable `position` does not change its value from 2. The distinction between events and conditions is very important in terms of what the monitor can infer about the execution based on the information it gets from the filter. The monitor can conclude that an event does not occur at any moment except when it receives an update from the filter. By contrast, once the monitor receives a message from the filter that variable `position` has been assigned the value 2, we can conclude that `position` retains this value until the next update. Notice that MaC reasons about temporal behavior and data behavior of the target program execution using events and conditions. Events are abstract representation of time. An event has an attribute `time` indicating the time of occurrence which enables reasoning about temporal properties of monitored systems. In contrast, conditions are abstract representation of data. This distinction between events and conditions is made in many formalisms (e.g. event-condition-action rules [NO99], SCR\* [HBGL95], and event/condition systems [SK91]).

We assume a countable set  $\mathcal{C} = \{c_1, c_2, \dots\}$  of primitive conditions. For example, these primitive conditions can be Java boolean expressions built from the monitored variables. In MEDL (see Sect 6.2.3), these will be conditions that were recognized by the event recognizer and sent to the run-time checker. We also assume a countable set  $\mathcal{E} = \{e_1, e_2, \dots\}$  of primitive events. Primitive events correspond to updates of monitored variables and calls/returns of monitored methods. The primitive events in MEDL are those that are reported by the event recognizer. Table 1 shows the syntax of conditions (C) and events (E).

There are some natural events associated with conditions, namely, the instant when the condition becomes *true* (`start(c)`), and the instant when the condition becomes *false* (`end(c)`). Also, any pair of events define an interval of time, so forms a condition  $[e_1, e_2)$  that is *true* from event  $e_1$  until event  $e_2$ . Finally, the event ( $e \mathbf{when} c$ ) is present if  $e$  occurs at a time when condition  $c$  is *true*. Note that our specification languages do not have non-deterministic operators due to the computational complexity mentioned in Sect 5.

During execution, variables routinely become undefined when they are out of scope. We choose to use a three-valued logic, where the third value is taken to represent undefined ( $\Lambda$ ). We interpret conditions over three values, *true*, *false*, and  $\Lambda$ . The predicate `defined(c)` is true whenever the condition  $c$  has a well-defined

---

```

ReqSpec <spec_name>
  // Imported event and condition declaration
  import event <e>;
  import condition <c>;

  // Auxiliary variable declaration
  var int <aux_v>;

  // Event and condition definition
  event <e> = ...;
  condition <c>= ...;

  // Property and violation definition
  property <c> = ...;
  alarm <e> = ...;

  // Auxiliary variable update
  <e> -> { <aux_v'> := ... ; }
End

```

---

**Fig. 4.** Structure of MEDL

value, namely, *true* or *false*. Negation ( $!c$ ), disjunction ( $c_1 || c_2$ ), and conjunction ( $c_1 \&\& c_2$ ) are interpreted classically whenever  $c$ ,  $c_1$  and  $c_2$  take values *true* or *false*.<sup>3</sup>

### 6.2.2. Primitive Event Definition Language (PEDL)

PEDL is the language for writing low-level specifications. The design of PEDL is based on the following two principles. First, we encapsulate all implementation-specific details of the monitoring process in PEDL specifications. Second, we want the process of event recognition to be as simple as possible. Therefore, we limit the constructs of PEDL to allow one to reason only about the current state in the execution trace. The name, PEDL, reflects the fact that the main purpose of PEDL specifications is to define primitive events of requirement specifications. All the operations on events can be used to construct more complex events from these primitive events. PEDL is dependent on its target programming language.

### 6.2.3. Meta Event Definition Language (MEDL)

The safety requirements are written in MEDL. Primitive events and conditions in MEDL specifications are imported from PEDL specifications. The overall structure of a MEDL specification is given in Fig 4.

**Importing events and conditions.** A list of events and conditions to be imported from an event recognizer is declared.

**Defining 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 [MP92] can be described in this way). Also observe that alarms and safety properties are complementary ways of expressing the same thing. The reason that we have both of them is because some properties are easier to think of in terms of conditions, while others are in terms of alarms.

---

<sup>3</sup> For the non-standard cases where these take the value  $\Lambda$  and formal semantics of events and conditions, see [KVK<sup>+</sup>04].

---

```

ReqSpec RRC
// Imported event and condition declaration
import event OpenGate, CloseGate;
import condition Gate_Down;

// Auxiliary variable declaration
var long lastClose;

// Requirement property definition
property TimelyGateClosing =
  [ CloseGate when !Gate_Down, OpenGate || start(Gate_Down) )
  => ( lastClose + 30 > currentTime);

// Auxiliary variable update
CloseGate -> {lastClose' := time(CloseGate);}
End

```

---

**Fig. 5.** A MEDL specification for the railroad crossing system

**Auxiliary variables.** The language described in Sect 6.2.1 has a limited expressive power. For example, one cannot count the number of occurrences of an event, or talk about the  $i$ th occurrence of an event. For this purpose, MEDL allows users to define auxiliary variables, whose values may then be used to define events and conditions. Updates of auxiliary variables are triggered by events. For example,

$$e1 \rightarrow \{\text{count\_}e1' := \text{count\_}e1 + 1;\}$$

counts occurrences of event  $e1$ .

We illustrate the use of MEDL using a simple but representative example. The example is inspired by the railroad crossing problem, which is routinely used as an illustration of real-time formalisms [HD96]. The system is composed of a gate that can open and close, taking some time to do it, trains that pass through the crossing, and a controller that is responsible for closing the gate when a train approaches the crossing and opening it after it passes. The common specification approach is to assume an upper bound on the time necessary for the gate to open or close. In reality, however, mechanical malfunctions may result in unexpectedly slow operation of the gate. A timely detection of such a violation lets the train engineer stop the train before it reaches the crossing.

In this example, we monitor the controller to check if the gate is down (indicated by `start(Gate_Down)`) within 30 seconds after signal `CloseGate` is sent, unless signal `OpenGate` is sent before the time elapses. Fig 5 specifies this requirement in MEDL. The time of the last occurrence of event `CloseGate` is recorded by the auxiliary variable `lastClose`. The requirement imports two events and one condition and states that if there is a `CloseGate` event at the time when the gate is not down, which is not followed by either event `OpenGate` or condition `Gate_Down` becoming true, then the time allotted for gate closing should not elapse yet.<sup>4</sup>

### 6.3. Expressive Power of MEDL

In this section, we show that MEDL is expressive enough for the monitoring purpose. More specifically, we show that for every  $dr\text{-}\mathbb{N}_*$ -automaton  $\mathcal{A}_M$ , there exists a MEDL script  $M_{\mathcal{A}}$  which accepts exactly the same strings. Furthermore, we also show that the class of languages that can be specified with MEDL is exactly the class of monitorable languages  $\mathcal{M}$ .

**Theorem 6.1.** MEDL is expressive enough for  $\mathcal{M}$ .

*Proof.* Consider a  $dr\text{-}\mathbb{N}_*$ -automaton  $\mathcal{A}$ . The elements of  $\Sigma$  (the input alphabet of  $\mathcal{A}$ ) will be all the imported events, and there will be an auxiliary variable corresponding to each of the  $m$  storing locations of the automaton  $\mathcal{A}$ . In addition, there will be an auxiliary variable `state` that will store the state of the automaton. Let  $Pr$  be the set of all boolean expressions that label the edges of the automaton  $\mathcal{A}$ . Corresponding to each

---

<sup>4</sup> `currentTime` is a predefined system variable indicating current time of the target system, which is updated periodically.

such boolean expression  $b \in Pr$ , we will define a condition  $C_b = b$  and an event  $E_b = start(C_b)$ ; note, that the expression  $b$  contains no primed variable. A transition  $(q_1, a, b, q_2, f)$  is transformed into a guard

$$(a \& \& E_b) \text{ when } (\text{state} == q_1) \rightarrow \{\text{state}' := q_2; f';\}$$

where,  $f'$  is the sequence of updates that produces the same result as function  $f$ . Finally, the automaton accepts only those strings that do not cause it to be stuck at any point; this is captured by defining the safety property of the MEDL script to be something that says if  $\text{state} == q$ , then the boolean expression labeling one of the out-going transitions must be true. It is clear that this MEDL script will behave exactly like the automaton.  $\square$

**Theorem 6.2.** For every MEDL script  $M$ , there exists a  $dr\text{-}\mathbb{N}_*$ -automaton  $\mathcal{A}_M$ .

*Proof.* Let  $M$  be a MEDL script. The idea behind constructing the automata is that the input labels on the transitions will be subsets of events; since conditions can be identified with events corresponding to the changes in the values of the condition, the presence of conditions (imported and locally defined) in the MEDL script will not present a problem. So for every change in an imported condition  $c$  will be associated with events  $c_T$  ( $start(c)$ ),  $c_F$  ( $end(c)$ ), and  $c_\Lambda$  ( $end(defined(c))$ ) corresponding to  $c$  becoming **true**, **false** and  $\Lambda$ , respectively.

The input alphabet of the automaton will consist of all subsets of the imported events, and the events associated with the imported conditions. The automaton will have two states  $a$  and  $r$ , corresponding to “accept” and “reject”. There will be no transitions from state  $r$ , and there will be one transition from  $a$  to  $r$  if either an alarm becomes true or a safety property becomes false. For every subset,  $S_i$  of events and conditions imported by  $M$ , and every subset  $S_d$  of events and conditions defined in  $M$ , there will be a transition, from  $a$  to  $a$ , whose input label will be the one corresponding to  $S_i$ , whose boolean condition will be  $S_d$ , and the transforming function will be the result of applying the guards, given that  $S_i$  and  $S_d$  were true.  $\square$

From Theorem 6.1 and Theorem 6.2, we conclude that the class of languages expressible in MEDL is exactly  $\mathcal{M}$ .

## 7. Related Work

Run-time monitoring technique to check correctness of system execution (called “run-time verification”) has received great attention recently, and many systems have been developed for the purpose. There are monitoring systems that analyze programs written in C [JZTB98, TJ98] and Java [HR01, SRA03, KVK<sup>+</sup>04, OSK05], by instrumenting the program to extract information. Different specification languages with varying expressive powers have been used to specify monitoring requirements ranging from simple boolean expressions [JZTB98] to some versions of propositional temporal logic [HR01], extensions of propositional temporal logic [KVK<sup>+</sup>04], second-order monadic logic [SS98], and logics for partial-order traces [SG03]. There has been, however, very little work in understanding the fundamental limitations of what properties can and cannot be monitored. In the seminal paper [Sch00], monitorable properties are identified with safety properties. This was refined in [Vis00].

In addition to the passive monitoring where primary focus is to detect violation of given properties, there has been research on “active” monitoring whose aim is to take remedial steps for dangerous behaviors. Hamlen et. al. [HMSar] have identified the class of properties that can be enforced; namely properties that can be detected and for which corrective action can be taken before a serious violation happens. The class of properties they identify as enforceable is strict subset of the class identified in this paper. The difference between these classifications stems from the fact that in this paper, we are only concerned with the problem of monitoring to detect errors (possibly after the violation has occurred) and not in enforceable properties. [BLW03] designs a general-purpose language for designing run-time security policies based on computational lambda calculus. [LBW05] introduces *edit automata* as a program monitor, which characterize practical security monitors more accurately.

## 8. Conclusion and Future Work

This paper presents descriptive theory on run-time monitoring. Run-time monitoring can serve as a complementary method, in addition to formal verification and testing, for assurance of the systems' correctness. Formalization of the computational nature of run-time monitoring is, therefore, a key requirement for utilizing this valuable technique. We have provided a descriptive theory for the class of monitorable languages  $\mathcal{M}$ . We showed that  $\mathcal{M}$  is a strict subset of the class of safety languages and  $\mathcal{M}$  corresponds to  $\Pi_1^0$  in the Arithmetic hierarchy. Also, we introduced a class of automata with storage which can specify  $\mathcal{M}$ . We showed that there exists a MEDL specification which can express such automaton and vice versa, by which we proved that the class of languages that MEDL defines is exactly  $\mathcal{M}$ . Therefore, the MaC architecture, whose specification language is MEDL, can be a general framework for run-time monitoring and checking. Furthermore, we studied the computational complexity of monitoring and checking behavior of systems and showed that this problem is NP-complete with a sufficiently expressive requirement specification language.

One interesting future direction is extension of MEDL for specifying properties more conveniently. Although MEDL is expressive enough for monitoring purpose, it is sometimes awkward to express certain features like temporal ordering of complex events in MEDL [SEL<sup>+</sup>04]. For example, [OSK05] extends MEDL by adding quantifiers for specifying dynamically created objects in the system. Another research direction is to provide formal framework for extracting *minimal traces* from the target system, which are still able to check requirement properties correctly. We might be able to abstract out snapshots of execution based on the result from static analysis of the target program.

## References

- [AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.
- [BGK<sup>+</sup>02] K. Bhargavan, C.A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transaction on Software Engineering*, 2002.
- [BLW03] Lujio Bauer, Jarred Ligatti, and David Walker. Types and effects for non-interfering program monitors. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security—Theories and Systems. Mezt-NSF-JSPS International Symposium, ISSS 2002, Tokyo, Japan, November 8-10, 2002, Revised Papers*, volume 2609 of *Lecture Notes in Computer Science*. Springer, 2003.
- [EH93] J. Engelfriet and H. J. Hoogeboom.  $X$ -automata on  $\omega$ -words. *Theoretical Computer Science*, 110:1–51, 1993. Earlier version in Proceedings of the International Colloquium on Automata, Languages and Programming, pages 289–303, 1989.
- [HBGL95] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. Scr\*: A toolset for specifying and analyzing requirements. In *Proc. of COMPASS*, 1995.
- [HD96] C. Heitmeyer and D. Mandrioli, Eds. *Formal Methods for Real-Time Systems*. Number 5 in Trends in Software. John Wiley & Sons, 1996.
- [HMSar] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, to appear. Available from <http://www.cs.cornell.edu/fbs/publications/EnfClasses.pdf>.
- [HR01] K. Havelund and G. Rosu. Monitoring Java Programs with JavaPathExplorer. In *Proceedings of the Workshop on Runtime Verification*, volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier Publishing, 2001.
- [HR02] K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2002.
- [JZTB98] C. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. In *ACM Workshop on Program Analysis for Software Tools and Engineering*, 1998.
- [Kim01] Moonjoo Kim. *Information Extraction for Run-time Formal Analysis*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2001.
- [KLS<sup>+</sup>02] M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, checking, and steering of real-time systems. In *Runtime Verification, Copenhagen Denmark*, July 2002.
- [KVK<sup>+</sup>04] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-mac: A run-time assurance approach for java programs. *Formal Methods in System Design*, 2004.
- [Lan69] L. H. Landweber. Decision problems for  $\omega$ -automata. *Mathematical Systems Theory*, 3:376–384, 1969.
- [LBW05] Jay Ligatti, Lujio Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
- [MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, New York, 1992.
- [NO99] N.W.Paton and O.Díaz. Active database systems. *ACM Computing Surveys*, 31(1):63–103, 1999.
- [OSK05] I. Lee O. Sokolsky, U. Sammapun and J. Kim. Run-time checking of dynamic properties. In *Workshop on Runtime Verification, University of Edinburgh, Scotland, UK*, Electronic Notes in Theoretical Computer Science. Elsevier, 2005.

- [Sch00] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, Feb 2000.
- [SEL<sup>+</sup>04] U. Sannappan, A. Easwaran, I. Lee, , and O. Sokolsky. Simulation of simultaneous events in regular expressions for run-time verification. In *Runtime Verification, Barcelona, Spain*, April 2004.
- [SG03] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Systems. In *Proceedings of the Workshop on Runtime Verification*, Electronic Notes in Theoretical Computer Science. Elsevier, 2003.
- [SK91] R. Sreenivas and B. Krogh. On condition/event systems with discrete state realizations. *Discrete Event Dynamic Systems: Theory and Applications*, 1(2):209–236, 1991.
- [SRA03] K. Sen, G. Rosu, and G. Agha. Runtime Safety Analysis of Multithreaded Programs. In *Proceedings of the International Conference on Automated Software Engineering*, 2003.
- [SS97] T. Savor and R. E. Seivora. An approach to automatic detection of software failures in real-time systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 136 –146, June 1997.
- [SS98] Anders Sandholm and Michael I. Schwartzbach. Distributed safety controllers for Web services. *Lecture Notes in Computer Science*, 1382:270+, 1998.
- [Sta86] L. Staiger. Hierarchies of recursive  $\omega$ -languages. *Journal of Information Processing and Cybernetics EIK*, 22:219–241, 1986.
- [Sta97] L. Staiger.  $\omega$ -languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 339–387. Springer-Verlag, Berlin, 1997.
- [Tho90] W. Thomas. Automata on infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–191. Elsevier, Amsterdam, 1990.
- [TJ98] K. S. Templer and C. Jeffery. A configurable automatic instrumentation tool for ansi c. In *Proceedings of the International Conference on Automated Software Engineering*, 1998.
- [Vis00] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 2000.