Foundations for the Run-Time Monitoring of Reactive Systems
- Fundamentals of the MaC Language

Mahesh Viswanathan[1] and Moonzoo Kim[2]

[1] CS Dept. University of Illinois at Urbana Champaign
vmahesh@cs.uiuc.edu
[2] CSE Dept. Pohang University of Science and Technology
moonzoo@postech.ac.kr

Abstract. As the complexity of systems grows, the correctness of systems becomes harder to achieve. This difficulty promotes a run-time monitoring technique as a promising complementary methodology for higher system assurance. 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 ω-languages and show that the language of *Monitoring and Checking (MaC) architecture*, called MEDL, is expressive enough for the run-time monitoring.

First, we provide a descriptive theory for the class of monitorable languages and show that this class of languages coincides with the class Π_1^0 of the Arithmetic hierarchy. Second, we introduce a class of automata with storage that can be used to describe the class of monitorable languages using connections to the Arithmetic hierarchy. Finally, we show that MEDL can express the class of monitorable languages via the correspondence between MEDL and the automata with storage.

1 Introduction

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. Such systems are notorious for its complex behavior and difficulty of testing. As the complexity of systems grows, the correctness of systems becomes harder to achieve. This difficulty 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.

It is customary to model the behavior of such systems 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

Z. Liu and K. Araki (Eds.): ICTAC 2004, LNCS 3407, pp. 543–556, 2005.
© Springer-Verlag Berlin Heidelberg 2005

of request-response pairs that is generated by the system's interaction with its environment. 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 [1] has tried to identify the class of monitorable properties[1] with the class of *safety* properties. We instead identify the class of properties that can be monitored with the class Π_1^0 in the Arithmetic hierarchy. Π_1^0 consists of properties whose violation can be detected by a Turing machine by examining a finite prefix of the errant behavior. We will also introduce a class of automata that can be used to specify these properties, and that can serve as monitors for such properties.

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 briefly describes the Monitoring and Checking (MaC) architecture and the specification language Meta Event Definition Language (MEDL) of the MaC architecture. Then, we show that MEDL is expressive enough for $\mathcal{M}$. Finally, we enumerate related works in Sec 6 and Sec 7 concludes this paper.

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

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

For a (finite or infinite) word α, $\alpha(i)$ (or α_i) denotes the $(i+1)$st letter of α. 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) α, $u\cdot\alpha$, is defined by $u\cdot\alpha(i) = u(i)$ if $i \leq |u|$ and $u\cdot\alpha(i) = \alpha(i-|u|)$ otherwise. A finite word u is said to be a prefix of another word α if there is $\beta \in \Sigma^\infty$ such that $u\cdot\beta = \alpha$. $\mathrm{pref}(\alpha)$ is the set of all finite prefixes of α, and for a (finite or infinite) language L, $\mathrm{pref}(L) = \cup_{\alpha\in L}\mathrm{pref}(\alpha)$.

[1] In this paper, we use two terms "property" and "language" for the same meaning depending on its context.

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 [2] 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 $H_\omega = H_* \cdot b^\omega \cup \{0,1\}^* \cdot a \cdot \{0,1\}^\omega \cup \{0,1\}^\omega$.*

The language H_ω, defined above is a safety language. In order to see this, we only need to observe that for any execution not in H_ω, there is a finite prefix when this violation can be detected. Executions not in H_ω 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 σ is not in H_ω, 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 language* is a strict subset of a class of safety languages; 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 $\Sigma^* \setminus \mathrm{pref}(L)$ is recursively enumerable. The class of monitorable languages is denoted by $\mathcal{M}$.*

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

In our study of ω-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[3, 4, 5]. 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 Σ^* to $\mathbb{N}$, it provides a cleaner presentation for questions arising in automata theory.

A relation R is said to be *finitary* over Σ, if $R \subseteq (\Sigma^*)^m$. We write $Ru_1u_2 \ldots u_m$ instead of $(u_1, u_2, \ldots u_m) \in R$. 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 Σ. A languages $L \subseteq \Sigma^*$ is said to be in $\Sigma^0_n(\mathcal{C})$ if and only if for some relation $R \in \mathcal{C}$,

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

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

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

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 $\Sigma^0_n(\mathcal{C})$ if and only if for some $R \in \mathcal{C}$,

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

where, once again, Q_{n-1} and Q'_n are quantifiers, and i is an natural number. The languages in $\Pi^0_n(\mathcal{C})$ 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, $\Sigma^0_1(REC)$ coincides with the class of recursively enumerable (R.E.) languages, while $\Pi^0_1(REC)$ is the class of co-R.E. languages. For notational convenience, we will denote the classes $\Sigma^0_n(REC)$ and $\Pi^0_n(REC)$, simply as, Σ^0_n and Π^0_n, respectively.

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

Proposition 1. $\mathcal{M} = \Pi^0_1$

Proof. $[\mathcal{M} \Rightarrow \Pi^0_1]$ Consider $L \in \mathcal{M}$. From the definition of $\mathcal{M}$, we know that $\Sigma^* \setminus \mathrm{pref}(L)$ is recursively enumerable. Therefore, there is a recursive relation R such that $u \in \Sigma^* \setminus \mathrm{pref}(L)$ if and only if $\exists v R v u$. In other words, $u \in \mathrm{pref}(L)$ if and only if $\forall v R' v u$, where $R' = \neg R$. Furthermore, we know that L is a safety language, which implies that $L \in adh(\mathrm{pref}(L))$ where $adh(L) = \{\alpha \in \Sigma^\omega | \mathrm{pref}(\alpha) \subseteq$

$\text{pref}(L)\}$ [6]. 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 $L \in \Pi^0_1$.

$[\Pi^0_1 \Rightarrow \mathcal{M}]$ Let $L \in \Pi^0_1$. 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 $L \in \mathcal{M}$. □

4 ω-Automata with Storage

Finite state machines on infinite words, are very similar to those which accept finite words. On an ω-word, α, the machine works as if α 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 ω-words, using some kind of storage, was first introduced and studied Engelfriet and Hoogeboom [7]. We use definitions and concepts described there, to develop our theory. Before defining finite state machines on ω-words formally, we first define the notion of a storage type, and give an example.

Definition 1. *A storage type is a 5-tuple $X = (C, C_0, P, F, [\![\cdot]\!])$, 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*
- *$[\![\cdot]\!]$ is a function that defines the semantics of the predicate and function symbols. For each $p \in P$, $[\![p]\!] : C \rightarrow \{\mathbf{true}, \mathbf{false}\}$, and for each $f \in F$, $[\![f]\!] : C \rightarrow C$, 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 $[\![\cdot]\!]$ is extended to $BE(P)$ in the standard way. $[\![\cdot]\!]$ is also extended to finite words over F, by interpreting concatenation as function composition. In other words, $[\![f \cdot \varphi]\!] = [\![\varphi]\!] \circ [\![f]\!]$, where $\varphi \in F^*$ and $f \in F$.

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

$$[\![zero]\!](c) = \mathbf{true} \textit{ if and only if } c = 0$$
$$[\![+_k]\!](c) = c + k$$
$$[\![-_k]\!](c) = c - k, \textit{ if } c \geq k, \textit{ and undefined otherwise.}$$

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 2. *Let $X_1 = (C_1, C_{10}, P_1, F_1, [\![\cdot]\!]_1)$ and $X_2 = (C_2, C_{20}, P_2, F_2, [\![\cdot]\!]_2)$ 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 $(C, C_0, P, F, [\![\cdot]\!])$, 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 $[\![\cdot]\!]$ is then defined naturally, as follows.*

$$[\![p]\!](c_1, c_2) = \begin{cases} [\![p]\!]_1(c_1) \text{ if } p \in P_1 \\ [\![p]\!]_2(c_2) \text{ if } p \in P_2 \end{cases}$$
$$[\![f]\!](c_1, c_2) = \begin{cases} ([\![f]\!]_1(c_1), c_2) \text{ if } f \in F_1 \\ (c_1, [\![f]\!]_2(c_2)) \text{ otherwise} \end{cases}$$

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) [2]

Definition 3. *Let $X = (C, C_0, P, F, [\![\cdot]\!])$ 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,*
- *Σ is a finite input alphabet,*
- *δ 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, α 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 δ. More precisely, we say $(q, \alpha, i, c) \vdash (q', \alpha, i', c')$, if there exists a transition (q, a, φ, q', h), such that $[\![\varphi]\!](c) = \mathbf{true}$, $[\![h]\!](c)$ is defined, and $[\![h]\!](c) = c'$. 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 α, 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. *An ω-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 α. The language accepted by $\mathcal{A}$, $L_\mathcal{A}$, is the set of all ω-words accepted by $\mathcal{A}$.*

---

[2] For a discussion of the relative power of the various other acceptance conditions, readers are directed to [3, 4].

The above definition of acceptance coincides with Landweber's [8] $1'$-acceptance and with the "always" acceptance of [7].

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 δ, with $q_1 = q_2$, either $a_1 \neq a_2$ and $a_1, a_2 \neq \epsilon$, or $[\![\varphi_1 \wedge \varphi_2]\!](c) = \mathbf{false}$, 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 ϵ-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 ω-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 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 5. *The storage type of m integer variables $\mathbb{N}_m$ is given by $\mathbb{N}_m = (C, C_0, P, F, [\![\cdot]\!])$. $C = \mathbb{N}^m$ is the set of m-tuples of natural numbers, and $C_0 = \langle 0, \ldots, 0\rangle$. P consists of predicates $zero_i$, which test if the ith element of the current configuration is 0, i.e., $[\![zero_i]\!](\langle c_0, \ldots, c_{m-1}\rangle) = \mathbf{true}$ 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 ith entry to the jth entry; $SBR_{i,j}$ (subtract register) subtracts the ith entry from the jth entry. $ADC_{i,k}$ adds constant k to ith entry; similarly, $SBC_{i,k}$ and $MLC_{i,k}$ subtract and multiply constants k, while $QC_{i,k}$ and $RMC_{i,k}$ 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 $\cup_m \mathbb{N}_m\mathcal{L}$.

Theorem 1. *The following classes of ω-languages are equivalent.*

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

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)] A real-time automaton may not have the "time" to do the computation performed by a finite delay machine. However, if the real-time machine can simulate a buffer, 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 basically show that df-$\mathbb{N}_m\mathcal{L} \subseteq dr$-$\mathbb{N}_{m+2}\mathcal{L}$. The two extra integer locations will be used by the real-time machine to simulate a buffer. 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. Detailed proof is omitted.

[(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, $L \in \Pi^0_1$. □

5 The Language of the Monitoring and Checking Architecture

5.1 Overview of the MaC Architecture

The *Monitoring and Checking (MaC)* architecture [9, 10] 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. Fig 1 shows the overview of the MaC architecture.

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 code. 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. 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

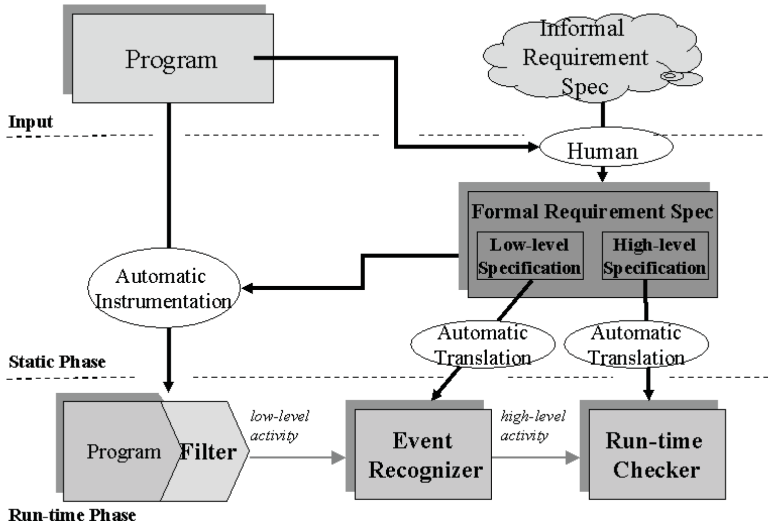

Fig. 1. Overview of the MaC architecture

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 [11, 12].

5.2 Specification Languages of the MaC Architecture

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.

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.

We assume a countable set $\mathcal{C} = \{c_1, c_2, \ldots\}$ of primitive conditions. For example, these primitive conditions can be Java boolean expressions built from the monitored variables. In MEDL (see Sec 5.2), these will be conditions that were recognized by the event recognizer and sent to the run-time checker. We also assume a countable set $\mathcal{E} = \{e_1, e_2, \ldots\}$ 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).

Table 1. The syntax of conditions and events

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

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 (Λ). We interpret conditions over three values, $true$, $false$, and Λ. The predicate `defined`(c) is true whenever the condition c has a well-defined 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 $false$; the only non-standard cases are when these take the value Λ. In these cases, we interpret them as follows. Negation of an undefined condition is Λ. Conjunction of an undefined condition with $false$ is $false$, and with $true$ is Λ. Disjunction is defined dually; disjunction of undefined condition and $true$ is $true$, while disjunction of undefined condition and $false$ is Λ. Implication ($c_1 \Rightarrow c_2$) is taken to $!c_1||c_2$. For events, conjunction ($e_1\&\&e_2$) and disjunction ($e_1||e_2$) are defined classically; so $e_1\&\&e_2$ is present only when both e_1 and e_2 are present, whereas $e_1||e_2$ is present when either e_1 or e_2 is present.

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)). Notice that the event corresponding to the instant when the condition becomes Λ can be described as `end(defined`(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 `when` c) is present if e occurs at a time when condition c is *true*.

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 and conditions are abstract representation of data. For formal semantics of events and conditions, see [9].

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.

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 2.

```
ReqSpec <spec_name>

  /* Import section */
  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 section*/
  <e> -> {  <aux_v'> := ... ; }
End
```

Fig. 2. Structure of MEDL

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 [13] 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.

Auxiliary variables. The language described in Sec 5.2 has a limited expressive power. For example, one cannot count the number of occurrences of an event, or talk about the ith 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 -> {count_e1' := count_e1 + 1;}
```

counts occurrences of event `e1`.

5.3 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-$\mathbb{N}_*$-automaton $\mathcal{A}_M$, there exists a MEDL script $M_\mathcal{A}$ which accepts exactly the same strings.

Theorem 2. *MEDL is expressive enough for $\mathcal{M}$.*

Proof. Consider a dr-$\mathbb{N}_*$-automaton $\mathcal{A}$. The elements of Σ (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 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

$$(a\&\&E_b)\ \texttt{when}\ (\texttt{state} == q_1)\ \texttt{->}\ \{\texttt{state}' := q_2; f';\}$$

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 `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. □

6 Related Work

Monitoring systems at runtime to ensure correctness has received a lot of attention recently, and many systems have been developed. There are monitoring systems that analyze programs written in C [14, 15] and Java [16, 17, 18, 9], 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 [14] to some versions of propositional temporal logic [17] to extensions of propositional temporal logic [9] and logics for partial-order traces [19]. However, there has been very little work in understanding the fundamental limitations of what properties can and cannot be monitored. In the seminal paper [1], monitorable properties are identified with safety properties. This was refined in [6]. More recently, Hamlen et. al. [20] 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.

7 Conclusion and Future Work

Run-time monitoring can serve as a complementary method, in addition to formal verification and testing, for assurance of the systems' correctness. In this paper, we have formalized the computational nature of run-time monitoring, which is necessary for utilizing this valuable technique. We have provided a descriptive theory for the class of monitorable languages $\mathcal{M}$ and showed that MEDL, the specification language of the MaC architecture, is expressive enough for $\mathcal{M}$. We showed that $\mathcal{M}$ is a strict subset of the class of safety languages and $\mathcal{M}$ corresponds to Π_1^0 in the Arithmetic hierarchy. Also, we introduced a class of automata with storage which can specify $\mathcal{M}$, then showed that there exists a MEDL specification which can express such automaton. Therefore, the MaC architecture, whose specification language is MEDL, can be a general framework for run-time monitoring.

Although the MaC architecture provides an expressive language MEDL, it is sometimes awkward to express certain features like temporal ordering of complex events in MEDL. Extending MEDL for specifying requirements more easily could be one further research direction. For example, [21] extends MEDL for describing regular expressions more conveniently.

References

1. Schneider, F.B.: Enforceable security policies. Technical Report TR98-1664, Cornell University (1998)
2. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters **21** (1985) 181–185

3. Staiger, L.: Hierarchies of recursive ω-languages. Journal of Information Processing and Cybernetics EIK **22** (1986) 219–241
4. Thomas, W.: Automata on infinite objects. In Leeuwen, J.V., ed.: Handbook of Theoretical Computer Science. Volume B. Elsevier, Amsterdam (1990) 133–191
5. Staiger, L.: ω-languages. In Rozenberg, G., Salomaa, A., eds.: Handbook of Formal Languages. Volume 3. Springer-Verlag, Berlin (1997) 339–387
6. Viswanathan, M.: Foundations for the Run-time Analysis of Software Systems. PhD thesis, Department of Computer and Information Science, University of Pennsylvania (2000)
7. Engelfriet, J., Hoogeboom, H.J.: X-automata on ω-words. Theoretical Computer Science **110** (1993) 1–51 Earlier version in Proceedings of the International Colloquium on Automata, Languages and Programming, pages 289–303, 1989.
8. Landweber, L.H.: Decision problems for ω-automata. Mathematical Systems Theory **3** (1969) 376–384
9. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Formal Methods in System Design (2004)
10. Kim, M.: Information Extraction for Run-time Formal Analysis. PhD thesis, Department of Computer and Information Science, University of Pennsylvania (2001)
11. Bhargavan, K., Gunter, C., Kim, M., Lee, I., Obradovic, D., Sokolsky, O., Viswanathan, M.: Verisim: Formal analysis of network simulations. IEEE Transaction on Software Engineering (2002)
12. Kim, M., Lee, I., Sammapun, U., Shin, J., Sokolsky, O.: Monitoring, checking, and steering of real-time systems. In: Runtime Verification, Copenhagen Denmark. (2002)
13. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer-Verlag, New York (1992)
14. Jeffery, C., Zhou, W., Templer, K., Brazell, M.: A lightweight architecture for program execution monitoring. In: ACM Workshop on Program Analysis for Software Tools and Engineering. (1998)
15. Templer, K.S., Jeffery, C.: A configurable automatic instrumentation tool for ansi c. In: Proceedings of the International Conference on Automated Software Engineering. (1998)
16. Mok, A.K., Liu, G.: Early detection of timing constraint violation at run-time. In: Proceedings of the IEEE Real-Time Systems Symposium. (1997)
17. Havelund, K., Rosu, G.: 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)
18. Sen, K., Rosu, G., Agha, G.: Runtime Safety Analysis of Multithreaded Programs. In: Proceedings of the International Conference on Automated Software Engineering. (2003)
19. Sen, A., Garg, V.K.: Partial Order Trace Analyzer (POTA) for Distributed Systems. In: Proceedings of the Workshop on Runtime Verification. Electronic Notes in Theoretical Computer Science, Elsevier (2003)
20. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Transactions on Programming Languages and Systems (2004, to appear) Available from `http://www.cs.cornell.edu/fbs/publications/EnfClasses.pdf`.
21. Sammapun, U., Easwaran, A., Lee, I., , Sokolsky, O.: Simulation of simultaneous events in regular expressions for run-time verification. In: Runtime Verification, Barcelona, Spain. (2004)