MaC: A Framework for Run-time Correctness Assurance of Real-Time Systems *

Moonjoo Kim, Mahesh Viswanathan
Hanéne Ben-Abdallah†, Sampath Kannan, Insup Lee, and Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA
January 29, 1999

*This research was supported in part by NSF CCR-9619910, AFOSR F49620-95-1-0508, ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, and ONR N00014-97-1-0505 (MURI)
†Visiting from Département d’Informatique, FSEG, Université de Sfax, Tunisia
Abstract

We describe the Monitoring and Checking (MaC) framework which provides assurance on the correctness of program execution at run-time. Our approach complements the two traditional approaches for ensuring that a system is correct, namely static analysis and testing. Unlike these approaches, which try to ensure that all possible executions of the system are correct, our approach concentrates on the correctness of the current execution of the system.

The MaC architecture consists of three components: a filter, an event recognizer, and a run-time checker. The filter extracts low-level information, e.g., values of program variables and function calls, from the system code, and sends it to the event recognizer. From this low-level information, the event recognizer detects the occurrence of “abstract” requirements-level events, and informs the run-time checker about them. The run-time checker uses these events to check that the current system execution conforms to the formal requirements specification of the system.

This paper overviews our current prototype implementation, which uses JAVA as the implementation language and our Monitoring Script language as the requirements language.
Contents

1 Introduction ............................................. 5

2 Related Research ........................................ 8

3 The Monitoring and Checking (MaC) Paradigm ............... 10
   3.1 Filter ............................................. 13
   3.2 Event Recognizer ................................... 15
   3.3 Run-time Checker .................................... 15

4 The Current MaC Prototype System .......................... 17
   4.1 Event Definition Languages .......................... 17
   4.2 Logic for Events & Conditions ......................... 18
   4.3 Primitive Event Definition Language (PEDL) .......... 21
   4.4 Meta Event Definition Language ....................... 26
   4.5 Target System ....................................... 28
   4.6 Issues in MaC prototype ............................. 30
       4.6.1 Naming of Monitored Variables and Execution Points 31
       4.6.2 Object Oriented Monitoring ....................... 33
       4.6.3 Atomicity of Variable Update Detection .......... 35
   4.7 Filter and Code Instrumentation ...................... 36
   4.8 Event recognizer ..................................... 39
   4.9 Run-time Checker .................................... 40

5 Examples .................................................. 41
   5.1 A Rail-Road Crossing System ......................... 41
   5.2 A Database Client Example ........................... 41
   5.3 An avionics example .................................. 46
6 Conclusions and Future work 47

A Java implementation of Database Client 53
1 Introduction

We develop a framework for run-time monitoring of correctness of real-time systems based on a formal specification of system requirements. Computer systems are often monitored for performance evaluation and enhancement, debugging and testing, control or check of system correctness [Sch95]. Recently, the problem of designing monitors to check for the correctness of system implementation has received increased attention from the research community [CG92, SM93, ML97, Sch98]. Such monitors can be used to detect violations of timing [ML97], a logical property of a program [CG92], a constraint on a language construct [SM93], and so on.

The reason for increased interest in correctness monitors is that it is becoming more difficult to test or verify software because software size is increasing and its functionality is becoming more complicated. The most common way to validate a software system is testing. However, testing cannot be used to guarantee that the system is error-free, since it is infeasible to completely test the entire system due to the large number of possible behaviors. Also, as the functionality and structure of software becomes complex in order to satisfy a broad range of needs, testing itself needs to be sophisticated enough to check the program according to diverse criteria. For example, for testing a numerical computation, it is enough to check output with given input. However, when we test a real-time application like traffic control system, we also have to check the timing behavior.

Formal verification has been used to increase the confidence that a system will be correct by making sure that a design specification is correct. However, even if a design has been formally verified, it still does not ensure the correctness of an implementation of the design. This is because the implementation often is much more detailed, and may not strictly follow the formal design. So, there are possibilities for introduction of errors into an implementation of a design that has been verified. One way to overcome this gap between the design and the implementation is to resort to testing the implementation’s behavior on a set of
input sequences derived from the specification. This approach, however, suffers from the same drawback as testing in general and does not provide guarantees about the correctness of the implementation on all possible input sequences. Consequently, we cannot guarantee, using the two traditional methods prior to the execution of the system, that its run-time behavior will be correct. Therefore, the approach of continuously monitoring a running system has received much attention.

In this paper, we describe a framework of monitoring and checking a running system with the aim of ensuring that it is running correctly. The salient aspect of our approach is the use of formal requirements specification to decide what properties to assure. Since our goal is to check an implementation against requirements specification at run-time, we assume that we are given both requirement specifications and an implementation. To be able to monitor satisfaction of requirements, we have to correlate low-level observations with high-level notions used in the requirements specification. Therefore, the primary concern of our presentation are the following two issues:

- how to map high-level abstract events that are used in a requirement specification to low-level activities of a running system
- how to instrument code to extract necessary low-level activities.

The framework consists of the three phases: the design phase, the implementation and instrumentation phase, and the run-time phase. During the design phase, the requirements on the system are specified. Optionally, a formal system specification may also be written down and in this case we assume that verification is done to ensure that the system specification satisfies the requirements. During the implementation phase the system is implemented. Based on the requirements specification and the implementation, the user provides a *monitoring script* that contains instructions for instrumenting the code so that low-level information about program state can be passed on to the monitor. In addition, it
also contains information that can be used to produce an event recognizer that transforms this low-level information into high-level events.

Our run-time 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 running code. In order to achieve this, we instrument the code of the system to be monitored. The filter sends this information to an event recognizer. It is the job of the event recognizer to map this low-level information about the running code to high-level information that the run-time checker understands. Based on the values of the monitored variables it receives from the filter, the event recognizer detects the occurrence of events that are described in the requirement specification, and sends them to the run-time checker.

The run-time checker then checks the correctness of the system thus far, according to a requirement specification of the system, based on the information of events it receives from the event recognizer, and on the past history. It checks the correctness of a sequence of events seen by stepping through the requirements specification and the correctness of a numerical computation by executing a program checker [BK95] for that function. This is one of the main advantages that our approach offers over other approaches to monitoring: it provides an integrated framework to check general requirements both for control flow during execution and for numerical computation.

The current prototype implementation of the MaC architecture, monitors programs that are in JAVA bytecode and uses a new language called MEDL which is related to linear temporal logic to describe the formal requirements. Although MEDL is used for requirements specification, it is possible to use other formal languages like ACSR [BGLG93], temporal logic, or Petri nets, by making only minor modifications to the existing framework.

The rest of the paper is organized as follows. Section 2 summarizes related research. Section 3 gives an overview of the MaC framework and discusses pertinent issues. Section 4 describes a prototype implementation of the MaC framework. Section 5 discusses example
applications that illustrate the use of our framework. Section 6 summarizes the presentation
and outlines future work.

2 Related Research

Traditionally, *verification* and *testing* have been two approaches to trying to ensure the
correctness of a program. In formal verification, one describes both the requirements
and the system in some formal specification language, and then attempts to prove that
the formal specification of the system satisfies the requirements. This “proof of satisfac-
tion” can either be obtained using a model checker [CES86], which exhaustively checks for
satisfaction of the requirement in all possible computation scenarios, or using a theorem
prover [Gor88b, OSR93], where one shows that the requirement is a logical consequence
of the system specification. However, both model checking and theorem proving have lim-
itations. The large size of the explicit representation of the state space of most systems
severely limits the size of systems that can be model checked. This is a problem referred
to as the *state space explosion* problem. Although state reduction techniques [Kur87] and
symbolic approaches [BCD+90, McM93] have been proposed to overcome the state space
explosion, fully describing and verifying a system is still hard. Theorem provers are difficult
to use as they, unlike model checkers, require extensive user interaction during proof con-
struction. In addition, they suffer from the fact that they cannot provide counter-examples
which show that the design is incorrect. Most importantly, even if some design has been
formally proved to be correct (using a model checker or a theorem prover), it does not mean
that the implementation is correct because often the implementation has greater detail, and
so is susceptible to errors not present in the design.

Testing attempts to overcome these problems by dealing with the implementation di-
rectly. Test cases are generated first, and then the system is tested by checking sequences
of test inputs and outputs from testing. However, too many test cases are needed if one
wants to detect all faults with high probabilities. Thus it is difficult to use testing to ensure the correctness of the system on all possible input sequences.

In order to overcome the limitations of both verification and testing, some researchers have tried to systematically derive the implementation of the system from a formal specification [BCG87, Tur95] or generate test cases and oracles using formal specification [LY96, JP97, CL97]. However, these approaches have their own limitations. Implementing the system from a formal specification has problems of inefficiency and incompleteness in a sense that only skeleton code of the system can be generated and a human programmer has to be involved to complete implementation, which could introduce errors. In addition, most research on testing with a formal specification [JP97, SS97] focuses on black-box testing. However, the correctness of reactive programs depends not only on the input-output behavior, but also on on-going interaction between components of the system.

Some researchers have, therefore, taken the approach of continuously monitoring the current execution of the program to ensure compliance with behavioral and temporal requirements. The “behavioral abstraction” approach to monitoring was pioneered by Bates and Wileden [BW83]. Although their approach lacked formal foundation, it provided a solid foundation for future developments. The work of [DJC94] addresses monitoring of a distributed bus-based system, based on a Petri Net specification. Since only the bus activity is monitored, there is no need for instrumentation of the system. The authors of [SS97] also consider only input/output behavior of the system. In our opinion, instrumentation of key points in the system is needed to detect violations timely and reliably, without too much performance overhead. In this spirit, the sentry system [CG92] observes the execution of program and determines whether the program is behaving “correctly” with respect to a set of specified logical properties on program variables. Sankar and Mandel have developed a methodology to continuously monitor an executing Ada program for specification consistency [SM93]. The user annotates an Ada program with constructs from ANNA, a formal specification language. But they can check only constraints on Ada constructs such as a
subtyping for Ada types. Mok and Liu [ML97] proposed an approach for monitoring the violation of timing constraints written in the specification language based on Real-time Logic. Their goal is to be able to detect timing violations as early as possible with low overhead. They describe how to derive a set of timing constraints for a given specification so that any violation can be caught as soon as it happens by checking the constraints of such a set. Our approach does not limit the scope of requirements to constraints on language construct nor focus on timing constraints. We monitor general requirements described in a formal specification. In the next section, we explain a run-time assurance monitoring architecture as a solution to the above limitations of prior research. In [LC92], an elaborate language for specification of monitored events based on relational algebra is proposed. The authors distinguish between conditions and events, as we do. The goal is to minimize effects of instrumentation on run-time performance, and to reduce the instrumentation cost through automated instrumentation.

3 The Monitoring and Checking (MaC) Paradigm

The MaC paradigm supports the run-time assurance monitoring of real-time systems. The paradigm consists of three phases for system development and deployment as shown in Figure 1: the design, implementation and run-time execution phases.

**The design phase.** In the *design* phase, we assume that the system requirements are formally specified. In this phase, a system design may also be formally specified and verified against its requirements specification. Run-time assurance monitoring in MaC can be applied both to a system with or without a formal specification. The requirements specification, however, must be given since it is the basis for run-time assurance. A formal specification of the system may be helpful in mapping high-level events of the requirements specification into the low-level system activities. However, the use of formal system speci-
The implementation phase. In the implementation phase, the code for the system is developed or derived from the design. Although there has been a progress in automating the derivation of an implementation from a design specification, currently the derivation for complex software produces at best skeleton code [SGME92, HLNP90] that must be manually augmented to complete an implementation. This gap between a design and an implementation in turn creates a gap between requirements and an implementation. The goal of the MaC paradigm is to narrow this gap.

One source of the gap is that requirements are described often in terms of “high-level events,” while code uses only “low-level state” information about execution. In the rest of the paper, we use “event” to denote “high-level event” and “state” for “low-level state.” In order to monitor and check required properties, the two types of information, i.e., events and states, need to be related and their relations must be specified explicitly in a monitoring script. The monitoring script describes how events at the requirements level are
defined on the monitored states of an implementation. For example, in a gate controller of a rail-road crossing system, the requirements may be expressed in terms of the event `train_in_crossing`; the implementation, on the other hand, only represents the train’s position with respect to the crossing in terms of a metric coordinate `train_position`. The monitoring script in this case can define the requirements event in terms of the value of the (implementation) variable `train_position`, for example as the time instant when `train_position < 800`.

The monitoring script is also used to generate a `filter` and an `event recognizer` automatically. A filter is a collection of code fragments that is used to instrument an implementation to monitor necessary state information at run-time and an `event recognizer` determines the occurrences of events using the information provided by the filter. We note that the monitoring script language depends on both the requirements and the implementation languages used.

The major part of a monitoring script language is to define events. There is a tradeoff between the expressiveness of event definitions and the run-time cost of event detection. The language can be designed to allow very expressive event definitions so that violation of any requirements property is itself an event. However, the drawback of an expressive event definition language is the cost of event detection at run-time. In general, as the language becomes more expressive, the granularity of detection needs to be finer, which incurs more overhead in both time and space. However, if the language has limited expressiveness, it cannot define some useful events. For example, if an event definition language cannot express $i^{th}$ occurrence of event, we cannot define $5^{th}$ `InCrossing` of the train. We employ a two-level approach that uses two languages, PEDL and MEDL (see Section 4.1). PEDL ensures that we have an efficient filter and event recognizer, whereas MEDL provides expressiveness. Furthermore, the implementation language specifics are restricted to PEDL.
The run-time phase. At run-time, the instrumented system is executed while being monitored and checked against the requirements specification. As Figure 1 shows, during execution the filter sends relevant state information to the event recognizer that determines the occurrence of events. These events are then relayed to the run-time checker to check adherence to the requirements. Our current system is geared towards the detection of faults. It would be desirable in future to build monitors that can steer a system to correct states.

3.1 Filter

A filter is a set of program fragments that will be inserted into the implementation to instrument the system. The essential functionality of a filter is to keep track of monitored objects, such as program variables and function calls, and to send pertinent state information to the event recognizer according to the monitoring script. This part of the MaC framework depends on the implementation language used. In addition, it must resolve the following four issues.

When to instrument. The filter can be added to the implementation either statically or dynamically. Static instrumentation means inserting the filter in the implementation before the system is executed, whereas dynamic instrumentation involves inserting the filter in the implementation at run-time. The advantage of dynamic instrumentation is flexibility, for example, where and what to insert can be determined at run-time based on the intermediate result of the execution (e.g., [MC95]). However, it may incur extra overhead at run-time to determine when it is safe to insert and remove filters. Also, it requires a complex instrumentation mechanism. In our current prototype described in Section 4, we use static instrumentation.

What to instrument. The filter can be inserted at the source-code level or the executable-code (e.g., bytecode) level. Compared to the source-code level instrumentation, the bytecode
level instrumentation is complicated since the bytecode of a system does not have source-level information useful for understanding the program. In addition, modifying a system at executable-code level requires modification not directly related to monitoring, but necessary to keep the format of executable code consistent. The major advantage, however, is that this executable-code level of instrumentation can be applied to a broad range of target systems because executable code is always available for running a system. Especially in mobile program environments it makes much sense. Mobile programs migrate in the network in executable-code form; although we may have source code, we have to instrument a mobile program in the executable code. If we have some limited source-code level information on the system such as meanings and names of monitored variables and functions, we can instrument the executable code of the system without complete access to the source code. Another advantage of the executable-code instrumentation is that low level operations on executable code can provide finer granularity of observations and reliable detection of changes in the program state. For the above reasons, our prototype instruments the bytecode (i.e., executable code) of the system.

How to instrument. Instrumentation can be done automatically or manually. With manual instrumentation, the user reads the source code of the program, then inserts filters (or probes) into the system. This can be done efficiently because it uses a heuristic and domain knowledge to pinpoint where to instrument and what state information to extract. However, its heuristic character may result in incompleteness. For assurance monitoring, instrumentation should be complete in the sense that it should capture all interesting information. Missing information could lead to false or missed detection of faults. Automatic instrumentation determines what filters/probes should be inserted where in a mechanical way based on the definition of the event and the structure of the program. The weak point of automatic instrumentation is that it may not be easy to define an event of high level behavior based on low level state information.
3.2 Event Recognizer

The event recognizer is the part of the monitor that detects an event from values of monitored variables received from the filter according to the event definition in the monitoring script (see Figure 1). Each time it recognizes an event defined in the monitoring script, the event recognizer sends it to the run-time checker. In addition to sending events, the event recognizer may also forward variables’ values to the run-time checker which uses them to check certain types of requirements, e.g., a function computation.

While it is conceivable to merge the event recognizer with the filter, separating the two modules is more advantageous: it shields the system execution from the overhead of abstracting out events from low-level information. In other words, it minimizes interference with the monitored system’s execution. On the other hand, this architecture contains communication overhead as the filter must send monitored variable changes to the event recognizer. Additionally, implementation of the event recognizer as a module separate from the filter allows us to monitor distributed systems by having a filter in each of the modules of the system communicate with the central event recognizer.

3.3 Run-time Checker

The run-time checker checks that the execution thus far belongs in a set of all acceptable behaviors, as defined by the requirements specification. The nature of this set determines the kind of assurance we may have about the correctness of the system. For example, if the set of acceptable behaviors is a set of traces, then all we can say about an execution that is “passed” by the run-time checker is that the sequence of events seen is correct. Another possibility is to have the set of acceptable behaviors to be a set of timed traces. In this case, it is possible to ensure not only the correctness of the system, but also its timing properties.

In Section 4.4, when we discuss the requirements specification language MEDL, we will formally define the notion of a valid trace. The Trace Validity Problem, is a membership
checking problem to determine if a given trace is in the set of valid traces. For sufficiently
effective requirements specification languages (such as a process algebra like ACSR) this
problem turns out to be NP-complete. Thus care should be taken in defining this language
to make it expressive enough while still leading to a tractable trace validity problem. We
show that for MEDL the trace validity problem can be efficiently solved.

The monitor can provide several kinds of formal guarantees of the system correctness.
If all the checking that is required is the validity of a trace in MEDL then the monitor
absolutely guarantees that the system behavior so far is correct. If the program uses nu-
merical functions, the correctness of whose outputs is checked using the program checking
paradigm, then the guarantees on system behavior so far will be probabilistic. However,
the probability that the guarantee is incorrect can be made as small as we need. There
is a possibility that monitoring of a running system’s behavior over sufficiently long inter-
vals will allow us to provide statistical guarantees on the system itself and not just on its
behavior. This is a direction that we will explore in the future.

Another issue is unexpected events. Unexpected events are events which are not de-
scribed in the requirements specification and thus not detected by the event recognizer. If
an unexpected event happens at run-time, the monitor may not see this event and conse-
quently can make a wrong conclusion on the current execution of the system. For example,
let us assume that a requirements specification does not specify an event caused by arith-
metic exception such as division by zero. When arithmetic exception happens at run-time,
which may lead to the system crash, the monitor might not detect that the current execu-
tion is incorrect because it does not receive an event of arithmetic exception, since it is not
defined in the specification. Currently, we do not handle unexpected events, i.e., we assume
that unexpected events do not happen or these events do not influence the correct execution
of the system. However, these have to be considered for providing complete guarantees.
4 The Current MaC Prototype System

To evaluate the effectiveness of the MaC framework, we are developing a prototype system whose overall structure is shown in Figure 2. In this section, we first describe our event definition language in which monitored properties are written. Then we explain the target system to be monitored in the prototype and discuss three design issues related to the implementation of the prototype. Finally, we discuss the filter, the event recognizer and the run-time checker of the MaC prototype.

4.1 Event Definition Languages

In this section, we give a brief overview of the languages used in the current implementation of MaC to describe what to observe in the program and the requirements that the program
must satisfy. The scripts written in these languages are then used to generate the event recognizer and the run-time checker, automatically.

The monitor observes a trace of the current execution of a program and checks that it satisfies the requirements. We distinguish between two kinds of data that make up the trace of an execution: things that are true at some instant during the execution (which we call events), and facts that hold for a longer duration of time (which are called conditions). For example, the return from the method RaiseGate occurs only at the instant when the control returns from the method, while a boolean condition like \( \text{position} = 2 \) holds as long as the variable position doesn’t change its value from 2. Hence, in the monitoring script language (PEDL, section 4.3) and the requirements specification language (MEDL, section 4.4), we reason explicitly about both events and conditions. The distinction between events and conditions is important in terms of what the checker can infer about the execution based on the information received from the event recognizer. The checker assumes that truth values of all conditions remains unchanged between updates from the event recognizer. For events, the checker makes the dual assumption, namely, that no events (of interest) happen between updates.

Based on this distinction between events and conditions, we describe a simple two-sorted logic that defines the various operations on events and conditions. PEDL and MEDL are subsets of this logic with added definitions of primitive events and conditions.

### 4.2 Logic for Events & Conditions

**Syntax.** We assume a countable set \( C = \{c_1, c_2, \ldots\} \) of primitive conditions. For example, in the monitoring script language (Section 4.3), these primitive conditions will be Java boolean expressions built from the values of the monitored variables. In the requirements description language (Section 4.4) these will be conditions that were recognized by the event recognizer and sent to the run-time checker.
We also assume a countable set $E = \{e_1, e_2, \ldots\}$ of primitive events. When an event occurs (to be defined formally later), it can have an attribute value, which is an element of a set $D_e$. For example, $\text{StartM}(\text{RaiseGate})$ is a primitive event in the monitoring script language, which is present at the start of method $\text{RaiseGate}$ and whose attribute value is the tuple of values of all the parameters with which this method is called. The primitive events in the requirements description language are those that are reported by the event recognizer.

The logic has two sorts: conditions and events. The syntax of conditions ($C$) and events ($E$) is as follows:

$$(C) ::= c \mid [E], (E) \mid \neg (C) \mid (C) \land (C) \mid (C) \lor (C) \mid (C) \Rightarrow (C)$$

$$(E) ::= e \mid \text{start}(C) \mid \text{end}(C) \mid (E) \land (E) \mid (E) \lor (E) \mid (E) \Rightarrow (E) \text{ when } (C)$$

**Semantics.** The models for this logic are sequences of worlds, similar to those used for linear temporal logic. Each world has a description of the truth values of primitive conditions and occurrences of primitive events. More formally, a model $M$ is a tuple $(S, \tau, L_C, L_E)$, where $S = \{s_0, s_1, \ldots\}$, $\tau$ is a mapping from $S$ to the time domain (which could be integers, rationals, or reals), $L_C$ is a total function from $S \times C$ to $\{\text{true}, \text{false}\}$, and $L_E$ is a partial function from $S \times E$ to $D_e$. Intuitively, $L_C$ assigns to each state the truth values of all the primitive conditions. Similarly, in each state $s$, $L_E(s, e)$ is defined for each event $e$ that occurs at $s$ and gives the value of the primitive event $e$. The mapping $\tau$ defines the time at each state, and it satisfies the requirement that $\tau(s_i) < \tau(s_j)$ for all $i < j$, i.e., the time at a later state is greater.

We now define what we mean by a condition $c$ being true in model $M$ at time $t$ ($M, t \models c$), and an event $e$ occurring in a model $M$ at time $t$ ($M, t \models e$). The formal semantics of the logic is given in Figure 4.2. The labels on states define the truth value of primitive events and conditions. The semantics for negation ($\neg c$), conjunction ($c_1 \land c_2$), disjunction ($c_1 \lor c_2$) and implication ($c_1 \Rightarrow c_2$) of conditions is defined naturally; so $\neg c$ is true when $c$ is false,
\[
M, t \models c_k (c_k \text{ primitive}) \iff \text{there exists state } s_i \text{ such that } \tau(s_i) \leq t \text{ and } L_C(s_i, c_k) = \text{true}, \text{ and for all states } s_j, \text{ if } \tau(s_i) < \tau(s_j) \leq t \text{ then } L_C(s_j, c_k) \text{ is true, } \\
i.e., c_k \text{ is true at time } t \text{ if its last assigned value was true.}
\]

\[
M, t \models [e_1, e_2) \iff \text{there exists } t_0 \leq t \text{ such that } M, t_0 \models e_1 \text{ and for all } t_0 \leq t' \leq t, \\
M, t' \not\models e_2, \text{ i.e., it is true since event } e_1 \text{ until event } e_2.
\]

\[
M, t \models !c \iff M, t \not\models c.
\]

\[
M, t \models c_1 || c_2 \iff M, t \models c_1 \text{ or } M, t \models c_2.
\]

\[
M, t \models c_1 \&\& c_2 \iff M, t \models c_1 \text{ and } M, t \models c_2.
\]

\[
M, t \models c_1 \Rightarrow c_2 \iff M, t \not\models c_1 \text{ or } M, t \models c_2.
\]

\[
M, t \models e_k (e_k \text{ primitive}) \iff \text{there exists state } s_i \text{ such that } \tau(s_i) = t \text{ and } L_E(s_i, e_k) \text{ is defined.}
\]

\[
M, t \models \text{start}(c) \iff \exists s_i \text{ such that } \tau(s_i) = t \text{ and } M, \tau(s_i) \models c \text{ and } M, \tau(s_i - 1) \not\models c. \\
i.e., \text{start}(c) \text{ occurs when condition } c \text{ changes from false to true.}
\]

\[
M, t \models \text{end}(c) \iff \exists s_i \text{ such that } \tau(s_i) = t \text{ and } M, \tau(s_i) \not\models c \text{ and } M, \tau(s_i - 1) \models c. \\
i.e., \text{end}(c) \text{ occurs when condition } c \text{ changes from true to false.}
\]

\[
M, t \models e_1 || e_2 \iff M, t \models e_1 \text{ or } M, t \models e_2.
\]

\[
M, t \models e_1 \&\& e_2 \iff M, t \models e_1 \text{ and } M, t \models e_2.
\]

\[
M, t \models e \text{ when } c \iff M, t \models e \text{ and } M, t \models c. \\
i.e., \text{event } e \text{ occurs when condition } c \text{ is true.}
\]

**Figure 3:** Semantics of events and conditions.
is true only when both \( c_1 \) and \( c_2 \) are true, \( c_1 \land c_2 \) is true when either \( c_1 \) or \( c_2 \) is true, and \( c_1 \Rightarrow c_2 \) is true if \( c_2 \) is true whenever \( c_1 \) is true. Conjunction \( (c_1 \land c_2) \) and disjunction \( (c_1 \lor c_2) \) on events is defined similarly. Now, since conditions are true from some time until just before the instant when they become false, two events can naturally be associated with a condition, namely the instant when the condition becomes true (\( \text{start}(c) \)) and the instant when the condition becomes false (\( \text{end}(c) \)). Any pair of events define an interval of time, and forms a condition \([e_1, e_2]\) that is true from event \( e_1 \) until \( e_2 \). Finally, the event \( e \) when \( c \) is true if \( e \) occurs and condition \( c \) is true at that time instant.

Notice that some natural equivalences that one might expect, hold in this logic. For example, for any condition \( c \), \( c \equiv [\text{start}(c), \text{end}(c)] \). This allows one to identify conditions with pairs of events, and is the reason why the languages in the MaC framework, are called “event definition languages”. Also, for conditions \( c_1 \) and \( c_2 \), and event \( e \), \( e \) when \( c_1 \) when \( c_2 \equiv e \) when \( (c_1 \land c_2) \).

4.3 Primitive Event Definition Language (PEDL)

The monitoring script in the prototype implementation is written in a language called Primitive Event Definition Language, or PEDL for short. As stated earlier, PEDL is based on the logic for events and conditions described in Section 4.2. There are two underlying principles on which the design of this language is based. First, we would like to limit all the implementation specific details of the monitoring process to this language. Hence, primitive events and conditions defined in terms of methods and objects, are present only in the monitoring script; requirement specification makes no references to methods and objects of the program. Second, we would like the process of recognizing high-level events to be simple i.e., events should be recognized based only on the current state of the program. Thus, in this language we do not have logical operators that help reason about state sequences; so the \([,\cdot)\) operation on events is not present in this language. Since it reasons only about
the current state, we call it “primitive”.

**Structure of a PEDL script.** The BNF grammar for PEDL is given in Figure 4. Every PEDL script has an identifier (<MonScrID>), which is the name of the script. This is followed by the lists of events and conditions that are “exported”, i.e., the events and conditions whose truth and falsity is reported by the event recognizer to the checker. The monitored objects of the program are declared after the MonitoredObjects keyword, and the methods under observation are declared after the MonitoredMethods keyword. Events and conditions are defined after this. As stated earlier they are built up from expressions over the monitored object and methods. The script ends with the keyword End.

**Monitored Entities.** The user should be able to reason about any program object that holds a value and thus forms a part of the program state, and any event that the program can engage in. The former include individual variables and composite data structures (objects in Java). The latter, at least for Java programs, includes only method calls and returns (which also captures other activities, such as process creation and communication).

A PEDL script contains the names of all objects, methods, and local variables of methods in the program that need to be monitored. The naming follows the standard Java convention of using “.” to move down in the class/object hierarchy. When naming methods we require that the types of the arguments be specified. This is to help distinguish between overloaded method names. Local variables are referred to by the name of the method, followed by “.”, and the name of the variable.

Note that, local variables in different invocations of a method correspond to different entities, and so must be distinguished during recursive method invocations. However, this requires the maintenance of an expensive stack at the event recognizer, to keep track of the different values of the local variable. Hence, the default is not to distinguish between values of local variables across method invocations. But if the user desires to distinguish between
Monitoring Script Declaration

<MonScr> ::= MonScr <MonScrID> <EventDcl> <CondDcl> <MonObjDcl> <MonMethDcl> <EventDef> <CondDef> End

Event Declarations

<EventDcl> ::= export event <EventDcl'>
<EventDcl'> ::= <EventID> [ , <EventDcl'> ]

Condition Declarations

<CondDcl> ::= export condition <CondDcl'>
<CondDcl'> ::= <CondID> [ , <CondDcl'> ]

Monitored Object Declaration

<MonObjDcl> ::= MonitoredObjects : <MonObjDcl'>
<MonObjDcl'> ::= <T> <Variable> ( , <Variable> )* | <T> <Object> ( , <Object> )* | <MonObjDcl'> ; <MonObjDcl'>
/* <Variable>, <Object>, <Class> are Java identifiers for variable, object and class. */

Monitored Method Declaration

<MonMethDcl> ::= MonitoredMethods : <MonMethDcl'>
<MonMethDcl'> ::= <T> <Method> ( ; <T> <Method> )* 

(* <Method> is a Java method name along with types of parameters *)

Event Definitions

<EventDef> ::= EventDef : <EventDef'>
<EventDef'> ::= Event <EventID> = <EventExp> { ; <EventDef'>' }

Condition Definitions

CondDef ::= Condition <CondID> = <CondExp> { ; <CondDef'> }
<CondExp> ::= <SimpleCE> | [[<CondExp>] <CondOp> <CondExp>

Preliminary Nonterminals

SimpleEE ::= start (<CondExp>) | end (<CondExp>) | Update (<Variable>) | StartM (<Method>) | EndM (<Method>) | IoM (<Method>)
SimpleCE ::= <BooleanExp> | InM (<Method>)
<Exp> ::= time (<EventExp>) | value (<EventExp>, <int>)

Figure 4: BNF Grammar for PEDL

23
these values then he can append a "*" after the name of the local variable and this ensures that during recursive calls, the local variable values are differentiated. This actually, in a back-handed way, allows the user to reason about the trace itself in the event recognizer. The reason we allow it is because we want all program specific information to be limited to the PEDL script.

Also, since the filter must report all changes to monitored variables, it must also do so when a monitored object is changed under an alias. For example, if a monitored object a and another object b refer to the same object at run-time, then b also becomes monitored. However, to detect such aliasing requires the filter to do expense book keeping at run-time. So once again, the default is to only detect changes to objects made under the "monitored names". The user can, however, force the filter to detect changes made through aliases by appending a "*" to the name of the object.

Defining Events. The primitive events in PEDL correspond to instants in the execution when control is transfered to and from a method or when a monitored variable is assigned a value (not necessarily a new one). The following primitive events in PEDL correspond these situations.

The event update(x) is triggered when the variable/object x is assigned a new value. The value associated with this event is the new value of x. StartM(f) is triggered when method f starts executing, and EndM(f) is triggered when control returns from method f. The value associated with StartM is the tuple containing the values of all arguments. The value of an event EndM is a tuple that has the return value of the method, along with the values of all the formal parameters at the time control returns from the method. Besides these three, we have one other primitive event which is IoM(f). This is also triggered when control returns from a method f, but has as its value a tuple that contains the return value of the method, and the values of the arguments at the time of method invocation. This event allows one to look at the input-output behavior of a method, and is needed if one wants
to program check some numerical computation. Notice once again, that this event \( \text{IoM}(f) \) violates our second design principle, which is to limit the reasoning at the event recognizer to be based only the values of the variables at the current state. \( \text{IoM} \) and local variables (see paragraph on Monitored Entities) are the only exceptions to the second design principle, and are present in PEDL only because these are program specific.

All the operations for events described in the logic, can be used to construct more complex events from these primitive events. In PEDL, we also have two predicates \( \text{time} \) and \( \text{value} \), defined on events. \( \text{time}(e) \) and \( \text{value}(e) \) give the time and value associated with the last occurrence of event \( e \), respectively. Now, as we saw in the discussion on the primitive events in PEDL, most of the events have a tuple of values associated with them, i.e., \( \text{value}(e) \) is a tuple. Individual elements of the tuple can then be obtained via a projection operation. We therefore, overload the name \( \text{value}(e,n) \), which gives the \( n \)th element of the tuple \( \text{value}(e) \).

**Defining Conditions.** Primitive conditions in PEDL, are formed by boolean-valued expressions over the monitored variables. In addition to these, we have another primitive condition \( \text{InM}(f) \). This condition is true as along as the execution is currently within method \( f \). Notice that, if there is no recursion, then \( \text{InM}(f) \) is the same as \( [ \text{StartM}(f) \), \( \text{EndM}(f) ] \). However, if \( f \) is a recursive method, then \( \text{InM}(f) \) is true from the start of the first invocation till the end of the last invocation, and hence is not equivalent to \( [ \text{StartM}(f) \), \( \text{EndM}(f) ] \). While this is a desirable equivalence, we found that in most practical situations \( [\cdot,\cdot] \), defined the way we have in this document, is a more useful operation on events. Thus we have a special primitive condition \( \text{InM} \) in PEDL.

Complex conditions are built from the primitive conditions using boolean connectives. Pairing events (i.e. \( [\cdot,\cdot] \) operation) to define conditions is not allowed in PEDL.
4.4 Meta Event Definition Language

The safety requirements that need to be monitored are written in a language called Meta Event Definition Language, or MEDL. Like PEDL, MEDL is also based on the logic for events and conditions, described in section 4.2. Events and conditions in MEDL are built up from the events and conditions defined in PEDL; hence the language has the adjective "meta". MEDL also has auxiliary variables that may be used to record certain aspects of the current trace, that cannot be described using the connectives of the logic.

Structure of MEDL script. The structure of a MEDL script is shown in Figure 5. Once again, there is a name associated with the MEDL script, <ReqSpecID>. The list of events and conditions, the checker expects the event recognizer to inform it about, is then "imported" in <EventDcl> and <CondDcl>. The auxiliary variables are declared after the keyword AuxilliaryVariables. These auxiliary variables may be used to store certain aspects of the current trace, and their type must be one of the primitive Java types. The events are conditions defined after EventDef and CondDef are events and conditions that are used to define the safety properties. The properties that the program needs to satisfy is specified after this. The alarms are defined after the keyword Alarms. Finally, the rules for manipulation of auxiliary variables is given. The termination of the script is indicated by End.

Auxilliary Variables. The logic described is section 4.2 has a limited expressive power. For example, using the logical connectives described 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 the user to define auxiliary variables, whose values may then be used is defining events and conditions. These auxiliary variables, however, must be of one of the basic types in Java. The update rules for these auxiliary variables take the form of guarded commands. The antecedent of these guarded commands are events, while the consequent
Figure 5: BNF grammar for MEDL

```plaintext
/* Requirement Script Declaration */
<ReqSpec> ::= ReqSpec <ReqSpecID> [EventDcl] [CondDcl]
             [AuxVarDcl] [EventDef] [CondDef]
             [SafePropDef] [AlarmDef] [AuxVarDef] End

/* Event Declaration */
<EventDcl> ::= import event <EventDcl'> ;
<EventDcl'> ::= <EventID> [ , <EventDcl'> ]

/* Condition Declaration */
<CondDcl> ::= import condition <CondDcl'> ;
<CondDcl'> ::= <CondID> [ , <CondDcl'> ]

/* Auxiliary Variable Declaration */
<AuxVarDcl> ::= AuxiliaryVariables : <AuxVarDcl>' ;
<AuxVarDcl'> ::= <T> <Id> ( , <Id> )* [ ; <AuxVarDcl'> ]

/* Event Definitions */
<EventDef> ::= EventDef : <EventDcl'>
<EventDef'> ::= Event <EventID> = <EventExp> { ; <EventDef'> }

/* Condition Definitions */
<CondDef> ::= CondDef : <CondDcl'>
<CondDcl'> ::= Condition <CondID> = <CondExp> { ; <CondDef'> }

/* Safety Property Definitions */
<SafePropDef> ::= SafetyProperties : ( property <SafePropID> = <CondExp> ; )+

/* Alarm Definitions */
<AlarmDef> ::= Alarms : ( alarm <AlarmID> = <EventExp> ; )+

/* Auxiliary Variable Definitions */
<AuxVarDef> ::= AuxVarDef : ( <EventExp> -> <Update> ; )+
<Update> ::= <AuxVar(T)> := <Exp(T)> [ ; <Update> ]

/* Preliminary Nonterminals */
SimpleEE ::= start (<CondExp>) | end (<CondExp>)
CondExp ::= <SimpleCE> | [CondExp] CondOp CondExp
SimpleCE ::= <BooleanExp> | [ <EventExp> , <EventExp> ]
BooleanExp ::= Java Boolean Expression | <Exp> CompOp <Exp>
<Exp> ::= time (<EventExp>) | value (<EventExp>, <int>)
        | any Java expression built from monitored objects & methods
EventOp ::= && | ||
CondOp ::= ! | && | ||
BoolOp ::= ! | && | ||
CompOp ::= < | <= | > | >= | == | !=

27
```
is a rule for updation. For example, \( \text{RaisingGate} \rightarrow t = \text{time (RaisingGate)} \) says that when the event \( \text{RaisingGate} \) happens, we set the auxiliary variable \( t \) to the time of this event. Updates to auxiliary variables take place after the truth value of all events and conditions has been determined. This means that if an event (or condition) is defined using an auxiliary variable, then the old value (or the value computed in the previous state) is used to determine occurrence of the event.

**Defining events and conditions.** The primitive events and conditions in MEDL are those that are defined in PEDL. Besides these, primitive conditions can also be defined using boolean expressions using the auxiliary variables. Complex events and conditions are built up using the connectives of the logic described in section 4.2. These events and conditions are then used to define the 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 *always* be true during the execution. Alarms, on the other hand, are events that must never be raised. Note, that all safety properties [MP92] can be described in this way. Also observe that alarms and safety properties are complementary ways of expressing requirement. Indeed, the event corresponding to the safety property condition changing state from false to true is an alarm. The reason we have both of them is for the convenience of the user, because some properties are easier to think of in terms of conditions, while others are easier to think of in terms of alarms.

### 4.5 Target System

As discussed at section 3.1, our monitoring approach is different from similar approaches in the way instrumentation is done. The difference is twofold. First, we instrument a low level code directly rather than instrument high level code and re-compile it. Second, we
instrument a program so that we can monitor variables continuously without missing any updates of the variables. To support the above features, we chose Java bytecode as the target implementation for monitored system monitoring.

The choice of Java bytecode for implementation does not commit us to Java as high-level implementation language. "Java bytecode" for "Java" is like assembler code for C/C++. As assembler code can be generated from any high-level languages including C/C++, Java bytecode can be generated from many high-level languages. We use the term "class file" indicating a file which contains a program written in Java bytecode.

The rationale of selecting target system written in Java bytecode is as follows. First, a class file contains rich symbolic information of a program in constant pool which contains all class names, method and field type and name, exception names and ranges and type information [LY97]. Given a monitoring script, we can automatically instrument a class file with help of the information. Second, Java bytecode prohibits using pointers and is strongly typed. A program written in Java bytecode makes it easier to monitor program variables continuously than a program compiled from other conventional languages like C/C++. Let us see this through example. Suppose we want to monitor integer variable $x$ in $\text{main()}$.

```java
void main() {
    int x = 3;
    int *px = &x;
    int px2 = px;
    long px3 = px;

    x ++; // x == 4
    (*px)++; // x == 5
    (*(int *)px2)++; // x == 6
    (*(int *)px3)++; // x == 7

    g(px); // x == ?
}
```

As you see, $x$ can be updated directly by variable name $x$ or indirectly by pointer $px$. 

29
Even \( x \) can be updated through normal variable \( px2 \) of type \( int \) and \( px3 \) of type \( long \). In order to monitor one variable, we have to keep watching all other variables which can be converted to pointer type in general. It causes heavy overhead at run-time. Furthermore, pointer of \( x \) can be passed to another function \( g() \). Although \( x \) is a variable declared in \( main() \), we have to keep track of execution of function \( g() \) too.

Java bytecode does not use pointers to primitive types. It makes monitoring of accessing to variables of primitive type easy. Local variable \( x \) of primitive type can be updated only through the name \( x \) and only inside of the method where it's declared. For global variable(field variable) \( x \) of primitive type can be updated only through the name \( x \) and by methods which can access \( x \) following access modifier of \( x \). Monitoring variables of a non-primitive type (class type) is more complex than monitoring variables of primitive type, because Java bytecode uses reference type for handling objects. Still, strong typing system of Java bytecode makes monitoring non-primitive type easier than C/C++ (we will discuss it at section 4.6.2)

As Java gains popularity, more systems will be implemented in Java. In addition, there are many high-level languages, such as Ada and Lisp, that can be compiled into Java bytecode [Tol], making use of platform-neutral and well-designed structure of Java bytecode. Thus, choosing target system written in Java bytecode allows us to apply our methodology to a wide variety of systems.

4.6 Issues in MaC prototype

MaC prototype monitors a property which is a composition of assertions of the form “thread \( T \) updates object \( O \) to value \( V \) using method \( M \)” (we can omit references to the thread, the method, or the value of the object from the assertion when they are irrelevant). The composition can be done by temporal, boolean, conditional operators. MaC prototype can monitor well-known properties, including safety, temporal and fault-tolerance properties
(see Section 5). For example, in a railroad crossing specification, we can describe a safety property as when a train reaches a crossing (i.e. a thread Train updates train.x to 500 using method running()), the gate must be down (i.e. thread Gate has finished method gd()).

For monitoring a property, a target system should be instrumented in such a way that information about threads, values of objects and method invocations can be correctly extracted and passed to the monitor (see Section 4.7 for more detail). Thus main design issues are

- how to describe entities (i.e. threads, objects, values and methods) of an assertion in the monitoring script;
- how to recognize the entities during the execution of the target system;
- how to implement a correct mechanism for the recognition.

All the three issues are closely related to the language of target system. Below we discuss each issue in detail.

4.6.1 Naming of Monitored Variables and Execution Points

When we specify variables to be monitored, we have to identify the variable by its name and its static and dynamic environment explicitly. Because we want to monitor not only class fields, but also local variables in a method, the object naming scheme includes method names as well as object names. First, consider an example.

// Double linked list
class A {
    int x;
    A next, before;
    A(int x) { this.x = x;}

1We do not consider inner classes here.
public static void main(String[] args) {
    A a1 = new A(1), a2 = new A(2), a3 = new A(3);
    a1.setLink(null, a2);
    a2.setLink(a1, a3);
    a3.setLink(a2, null);
    a2.traverse();
}

void setLink(A b, A a) {
    this.before = b;
    this.after = a;
}

void traverse() {
    A a = this;
    for (int length = 0; a != null; length++) {
        System.out.println(a.x);
        a = a.next
    }
    System.out.println("Length is "+ length);
}

All object specifications begin with a name in the scope of main() or init(). Suppose we want to monitor length in method traverse of object a2; we specify this variable by a2.traverse().length. Suppose we want to monitor the value of variable x in an object that is being traversed from a2. We specify it as a2.traverse().a.x. For static variables and methods, we use class name rather than object name. In addition, we can include a name of a thread into the object specification. This is necessary to distinguish between local variables of a method invoked concurrently in two threads. The following BNF describes the syntax for monitored object naming specification\(^2\):

\[
<\text{VarName}> = ( <\text{ThreadID}>, <\text{VarName}'>)
\]
\[
<\text{VarName}'> = <\text{Var_of_main}>(\)
\]

\(^2\)This syntax is not complete in a sense that there can be some object which we can not distinguish from others using this syntax. However, we prefer current syntax because of its simplicity than a complete and complex syntax.

32
Here, `<Var_of_main()>` is a variable in the scope of `main()` and `<MethodNames>` is a sequence of nested method invocations.

Similarly, we name an execution point as a monitored entity.

```
<ExePoint> = ( <ThreadID>, <ExePoint'> )
<ExePoint'> = start of <VarName'>.<MethodName>
|   end of <VarName'>.<MethodName>
```

### 4.6.2 Object Oriented Monitoring

When we monitor the system as an evolving collection of objects, we face two difficulties: an aliasing problem and a reference changing problem. The following example explains the problems and our approach to its solution.
class A {
  int x;
  A(int x) { this.x = x; }
  public static void main(String[] args) {
    A a1 = new A(1);
    A a3 = a1;
    A a2 = new A(2);
    a3.x = 3;
    a2.swap(a1);
    a1 = a2;
  }
  void swap(A a) {
    int tmp_x = a.x;
    a.x = this.x;
    this.x = a.x;
  }
}

On line 6, an object with field x having value 1 is created with name a1. On line 7, it is
given another name a3, and on line 10 the object acquires still another name a4 in method
swap(). We refer to this as name aliasing. The reference changing problem is illustrated
on line 12. Name a1 does not refer the object created at line 6 anymore, but to a different
object created at line 8. Thus, to monitor an object correctly, we have to check all reference
variables which refer to the object. We can not know statically which references will refer
to an object. We have to check it at run-time by monitoring references which can refer to
the object. This task is made simpler by the fact that Java bytecode is strongly typed, that
is, a reference of type T can refer only to an object of type T. In order to handle aliases
to an object, the filter maintains a table that contains names of monitored variables and
addresses of corresponding objects. By checking value of every reference variable of type
T against the address of the object of type T, we can ensure that we monitor the object
without missing any updates, and that we monitor the correct object. In addition, we may
have to keep a signature of a frame stack for monitoring local variables.

4.6.3 Atomicity of Variable Update Detection

We insert a probe of the filter immediately after storing instruction such as `istore` and `putfield`. When a monitored variable is updated by storing instruction, a filter immediately sends information about the update, including the timestamp, to monitor. Based on the timestamp, the event recognizer determines the order of updates (we assume that clock synchronization is outside the scope of the problem). However, in a multi-threaded application, the order of updates may be determined incorrectly due to preemption by thread scheduler. Consider the following example.

As you see in Fig 6, update of x should be reported with an earlier timestamp than that of update of y. But thread 1 is preempted just before reporting of update of x, and update of y has earlier timestamp. Furthermore if thread 2 updates x too, the filter is left with an erroneous view of the value of x.

There are two conceivable solutions for the problem. The first one is to use two time stamps. One will be assigned immediately before an update of a variable, the other immediately after the update. If the time interval between the two timestamps of update of
x overlaps with the interval between the timestamps of updating y, the event recognizer detects that order of updates can be either way. However, this solution can not prevent the latter problem when thread 2 updated value of x. The second solution is to use a global lock for "send buffer" of a filter. A filter has a buffer for storing and sending update information. Immediately before and immediately after the storing instruction, we insert a filter probe to acquire the lock and place update information into the buffer, and then release the lock. This makes the update and its detection mechanism atomic. It causes overhead of acquiring and releasing lock at run-time. However, it guarantees the correct detection of order of updates and can be implemented simply using monitor. Therefore we choose the second for update detection mechanism.

4.7 Filter and Code Instrumentation

A filter consists of a set of probes inserted into class files of the target system and a separate filter object. The filter object is an instance of a class, which has methods for storing information about updates and for communication between filter and event recognizer, a table containing name and address of the monitored objects (see section 4.6.2), and a thread to process updates. To minimize a system overhead, inserted probes just store updation information in the buffer in the filter class. Checking a reference variable against the address of the object and sending updation information is performed by the filter thread in the background.

As discussed in section 3.1, we instrument a class file statically. We instrument the system to detect changes of values in monitored variables and send these values. Let us briefly describe how instrumentation works. The filter keeps watch over three program entities; execution point, local variables and field variables. We do not discuss thread identification here.
**Execution point.** The current prototype detects when the execution point reaches a method invocation, returns from a method. The start and end of the program and exceptions within a method are also detected. Invocation and return from the method can be detected by inserting the following bytecode before the first instruction and after the last instruction of the code for the method.

```
ldc "Execution Point"
ldc <ex_point>
invokestatic MAC/Filter/send(Ljava/lang/String;Ljava/lang/String;)V
```

Instruction ldc loads a constant into operand stack. `<ex_point>` is a string representation of start or end of a method, such as "Start of add(II)I", where I means integer types for two parameters and return value. The last line invokes static method MAC.Filter.send() of the filter class with two parameters "Execution Point" and `<ex_point>`. Other information about the update, such as thread name, timestamp, the parent object, the signature of a frame stack are implicitly passed to the filter. The method checks whether this update is related to the specified monitored entities or not, by looking at the thread name, static environment given by a parent object and dynamic environment
given by a signature of frame stack. Similarly, the start of the program is detected by inserting code at `main()` or `init()`. Ending of program can be detected by looking at invocation `exit()` method and checking whether every thread except the daemon thread terminates. An exception of a method can be detected by looking at the exception table. Every method has its own exception table that contains names of exception and jump addresses for the case when the exception occurs. We insert the above code at the jump address.

**Local variable.** Every local variable is indexed and accessed through this index which is fixed in bytecode instruction. The only instructions that update local variable are `<T>store <id>` where `<T>` is type of local variable and `<id>` is the index of a local variable and `iinc <id>`. Therefore, we can statically identify the places in the bytecode that update the local variable of interest. Immediately after the updating instruction we insert the following code:

```
ldc <local_var_name>
<T>load id
invokestatic MAC/Filter/send(Ljava/lang/String;<T>)V
```

**Field variable.** Detecting a change in a field variable is similar to detecting a change in a local variable. The `putfield` operator has a reference to an object as a run-time parameter and an index to the field in that object as a fixed parameter in bytecode. We can detect changes of the field variable by keeping watch on `putfield` operators and their reference parameters.

Inserting code may cause side effects in the monitored program. For example, the instrumented program may use more resources such as heap, stack and constant pool than the original program. It may affect the scheduling of threads because the newly added code consumes cpu time additionally. However, it seems very rare that a program exhausts all the resource due to inserted code and abnormally halts. Furthermore, a sound multi-threaded program depends on synchronization to guarantee correct execution.
The filter is generated by a filter generator, which is implemented in Java using JTrek library [Dig] for inserting code into the program. The filter generator takes a program to be instrumented and list of monitored variables and monitored methods as input. It generates an instrumented program as output by inserting probes at proper places of the code. The filter sends timestamped messages containing the updated values through object serialization [AG98], whenever it detects updates of any of the above three entities. To minimize the overhead to the system, sending values to event recognizer through the network is performed by a separate thread.

4.8 Event recognizer

The event recognizer needs to determine the truth or falsity of the events and conditions defined in the PEDL script, at all time instants. Assuming that the filter informs the event recognizer whenever some object of interest changes or some monitored method is invoked, the event recognizer needs to check the truth values at the times when it receives a message from the filter. The event recognizer maintains, at all times, a table that stores the current value for each monitored variable. Each message from the event recognizer causes this table to be updated. For local variables whose value needs to be distinguished across method invocation, we maintain a stack; each time the execution invokes the method in which this variable is defined, the new value of the variable is pushed onto the stack, and when the execution returns from such a method invocation, the top element is popped from the stack.

Whenever the event recognizer gets an update from the filter, it evaluates the truth of all events and conditions. Since conditions are defined in terms boolean expressions over monitored variables, they can be directly evaluated from the table storing the current values of all monitored variables. However, in order to determine if an event like start(c) is present it must not only know the current truth value of condition c, but also its truth value at the time of the previous update. The same is true for the event end(c). Hence
the checker also keeps track of the values of all the conditions at the time of the previous update, in addition to the values of the monitored objects. Finally, once the checker has determined the truth of all the conditions and events defined in the PEDL script, it sends to the checker the truth values of those events and conditions that have been “exported”.

4.9 Run-time Checker

The run-time checker gets information about the truth of certain events and conditions from the event recognizer. The checker then tries to see if the safety properties defined in the MEDL script are always true and that the alarms in the MEDL script are never raised. The checker, as well as the event recognizer, determines the truth of the events and conditions only at the times it gets a message from the event recognizer. To do this, the checker maintains a table that stores the values of the auxilliary variables defined in the MEDL script. The checker also keeps track of the truth of events and conditions at the time of the previous update. As we saw in the previous section, this is sufficient to determine the truth value of all events and conditions that do not involve the pairing operation. The truth of the condition $[e, f]$ can also be evaluated. The condition $[e, f]$ is true now, if either $e$ is present now and $f$ is not present now or, if $[e, f]$ was true at the time of the previous update and $f$ is not present now.

Once all the truth of all the events and conditions has been determined, the auxilliary variables are updated. If the event corresponding to the update rule is present is in the current time instant, then the auxilliary variable is updated as per the assignment. Whenever an alarm becomes true or a safety property becomes false, the checker declares the program to be incorrect.
5 Examples

5.1 A Rail-Road Crossing System

The first example illustrates how MaC framework is used for monitoring a safety-critical system, based on the railroad crossing. This example is commonly used as a benchmark in formal methods research [HD96]. There are three components in the system: a train, a gate, and a controller. The goal of the system is to operate the crossing gate subject to safety and utility properties. The safety property states that when a train is in the crossing, the gate is down.

We define four events to check the safety requirement in PEDL. \( \text{startIC} \) means the train reaches in the crossing, \( \text{endIC} \) means the train passes the crossing, \( \text{startGD} \) means the gate enters the closed state, \( \text{endGD} \) means the gate starts raising. We define the safety property written in MEDL using condition IC which means a train is in cross and GD which means a gate is down. Figure 8 presents a monitoring script containing the definitions of these events and the safety property.

Let us briefly explain what the program variables mean. \( \text{train}_x \) is the position of the tail of a train. \( \text{train}_\text{length} \) is the length of a train. Thus \( \text{train}_x + \text{train}_\text{length} \) yields the position of the head of a train. \( \text{cross}_x \) is the position of the left end of the crossing. Thus \( \text{cross}_x + \text{cross}_\text{length} \) is the position of the right end of the crossing. Method \( \text{Gate.gd}() \) is used to lower the gate. \( \text{Gate.gu}() \) is the method for raising the gate.

5.2 A Database Client Example

Another example describes monitoring of a database client. The client in our example periodically probes some database server (choosing it randomly from a list of servers) for some information. The informal pseudo-code for the client appears in Figure 9. Some of the requirements that one might be interested in monitoring for such a system are:
MonScr RailRoadCrossing

export event startIC, endIC, startGD, endGD;

MonVarDcl:
  float RRC.train_x;
  int RRC.train_length;
  int RRC.cross_x;
  int RRC.cross_length;

MonMethodDcl:
  Gate.gd();
  Gate.gu();

CondDef:
  Cond IC = RRC.train_x + RRC.train_length > RRC.cross_x &&
          RRC.train_x <= RRC.cross_x + RRC.cross_length;

EventDef:
  Event startIC = start(IC);
  Event endIC = end(IC);
  Event startGD = end_m(Gate.gd());
  Event endGD = start_m(Gate.gu());

End

ReqSpec RailRoadCrossing

import event startIC, endIC, startGD, endGD;

CondDef:
  Cond IC = [startIC, endIC];
  Cond GD = [startGD, endGD];

SafePropDef:
  SafeProp safeRRC = IC -> GD;

End

Figure 8: Monitoring script for the railroad crossing
Client:

loop periodically (every p time units do)
    randomly select a URL location url from a given list
    open an HTTP connection to url with a timeout of T
    if connection is established do
        repeat for R times or until successful
            send a CGI query for data
            successful <- response from server within d units of time
        if unsuccessful use old data
        else use the data just received
    else use old data
    process data
endloop

Figure 9: Informal description of the Database Client example

RT The client is indeed periodic, i.e., every p units of time it tries to query a new server.

FT If the processed data is old, then either a connection to the chosen server could not be opened or the client could not get a response to the query, even after R retries.

A sample Java implementation of the client, according to the pseudo-code above, is given in Appendix. The corresponding monitoring script is given in Figure 10. The events to detect in the monitoring script are the start of a period (periodStart), the start of an attempt to establish a connection to a server (conStart), the failure of an attempt to establish a connection (conFail), the resending of a query (queryResend), and the event of old data being used during processing (oldData). Using these events we can check the properties RT and FT described above. The properties RT and FT are formally specified in the corresponding MEDL script in Figure 11. The checker counts the number of times the query was resent since the begining of the period, using the auxiliary variable numRetries. The duration of the last period is stored in the variable periodTime, and the start time of the previous period is stored in the variable lastPeriodStart. The variable chkConFail is used to check if the connection did indeed fail during this period.
MonScr DBMon

export event periodStart, conStart, conFail, queryResend,
    oldData;

MonitoredObjects:
    int dbc.getData(BufferedReader).r;
    long dbc.run().startTime;
    Object dbc.oldData;

MonitoredMethods:
    void ct.ConnectTry (string, int);
    Socket ct.result ();
    Object dbc.getData();

EventDef:
    event periodStart = update(dbc.run().startTime);
    event conStart = startM(ct.ConnectTry);
    event queryResend =
        update(dbc.getData(BufferedReader).r);
    event oldData = start(value(endM(dbc.getData())) != dbc.oldData);
    event conFail = start(value(endM(ct.ConnectTry.result)) == null);

End

Figure 10: PEDL script for the database client
ReqSpec DBReq
    import event periodStart, conStart, conFail, queryResend, oldData;
    AuxilliaryVariables:
        long periodTime;
        long lastPeriodStart;
        int numRetries;
        boolean chkConFail;
    SafetyProperties:
        property periodic = (periodTime == p);
    Alarms:
        alarm wrongFT = oldData && ((numRetries < R) || !(chkConFail));
    AuxVarDef:
        initial -> periodTime = p;
        lastPeriodStart = 0;
        numRetries = 0;
        chkConFail = false;
        periodStart -> periodTime = time(periodStart) - lastPeriodStart;
        lastPeriodStart = time(periodStart);
        numRetries = 0;
        conStart -> chkConFail = false;
        conFail -> chkConFail = true;
        queryResend -> numRetries = numRetries + 1;
    End

Figure 11: Requirements for the database client
5.3 An avionics example

One more example of the utility of MaC approach is taken from a very important domain of modern warfare. Micro air vehicles (MAV), small unmanned planes that can be dispatched in large quantities very quickly (e.g., dropped from another aircraft), can be employed to perform many different tasks. One such task involves arranging MAVs into a hexagonal pattern, illustrated in Figure 12. Each MAV has to be near a grid of the pattern; several MAV’s can occupy the same grid point (this is, clearly, a two-dimensional view of a three-dimensional situation). Control of individual MAVs from a centralized controller is not feasible, therefore the MAVs must form the pattern through communications with their neighbours, using local information only. Gordon and Speers at NRL [Gor88a] devised a distributed algorithm to solve this problem. The algorithm is based on relative positions of the neighbors of an MAV.

The goal of this example is to demonstrate how monitoring and checking can be used to observe whether the desired pattern is forming as expected. The approach to monitoring is based on the observation that in the hexagonal pattern, each neighbor of an MAV is either at a fixed distance that is the parameter of the pattern (adjacent grid point), or very close to the MAV in question (same grid point). If the pattern is not fully formed, there are MAVs that have neighbors in other locations, and this can be detected as a violations of
the pattern. Intuitively, we should expect that as the pattern forms, the number of such violations should go down.

An implementation-independent MEDL specification of this property is shown in Figure 13. The primitive event MAVAlert, supplied by the event recognizer, denotes a misplacement of some neighbor of an MAV. Auxiliary variable currCount is used in the checker to count the number of violations of the pattern in the current interval. When the interval elapses, the accumulated number is compared with the number of violations in the previous interval. If a significant increase in the number of violations is detected, an alarm NoPattern is sent to the user as a notification of potential problems with the pattern formation.

This monitoring approach is applied to a distributed emulator of MAV deployment, implemented in Java. Each MAV is represented as a separate instance of class MAV, based on standard Java class Thread. When the thread in an MAV is run, it continuously executes the positioning algorithm and queries its neighbours for their positions. A local variable distance in the run() method of the class is used to hold the distance from the currently queried neighbor. The monitoring script for this implementation is shown in Figure 14. It defines event MAVAlert that is delivered to the checker. The event is defined in terms of the value of the variable distance. By declaring the variable as a monitored entity, the script instructs the filter to send all updates of this variable to the event recognizer which, in turn, compares them with the acceptable range of values as described in the script.

6 Conclusions and Future work

We propose a monitoring and checking architecture as a framework for run-time assurance. It is bridging a gap between the traditional two approaches, static verification and testing. We use ACSR and Java as our design and implementation language and mechanically instrument the target system on the bytecode level. To make run-time assurance monitoring convincing, we should provide formal guarantee of the requirement property at run-time,
ReqSpec HexPattern

import event MAValert;

Auxiliary Variables :

long currInterval;
int currCount;
int prevCount;

Alarm Definitions :

Alarm NoPattern = (currCount > prevCount + 5) && (prevCount != -1)

Auxiliary Variable Definitions:

start(true) ->
    currInterval := System.currentTimeMillis()
    currCount := 0;
    prevCount := -1

System.currentTimeMillis() > currInterval + 3000 ->
    currInterval := currInterval + 3000;
    prevCount := currCount;
    currCount := 0

MAValert ->
    currCount++

End

Figure 13: Requirement specification for pattern monitoring
MonScr MAVpattern

export event MAValert;

Monitored Entities:

double MAV.run().distance

Event Definitions:

Event tooClose = (distance > 0.25*R) && (distance < 0.75*R);
Event tooFar = (distance > 1.25*R) && (distance < 1.5*R);
Event MAValert = tooClose || tooFar

End

Figure 14: PEDL script for pattern monitoring

which we did not fully investigate yet. We intend to incorporate process algebraic techniques for reasoning about concurrency and communication, and program checking for numerical calculations into our monitoring approach.

We are investigating a number of issues that deal with extension of MAC architecture to distributed monitoring. For example, in a distributed environment, we have to capture the global state based on information from local monitors only. We think instrumentation of the Java virtual machine can help distributed monitoring to obtain proper timing information about threads and communications within the system. We performed experiments using a multiple reader-writer example and have some theoretical results on distributability of monitor.

Another interesting question is about gap between the abstraction provided by a computation model of a specification language and the real program. Most of formal specification languages have an ideal model which might not fit into the real program. The question here is how to compare abstract executions of the specification and concrete runs of the monitored system and adjust the conclusions drawn from the monitoring process to account
for this kind of difference.

References


A Java implementation of Database Client

```java
import java.io.*;
import java.net./*;

class DBClient extends Thread{
    URL[] urlList;
    int numUrls;
    int period;
    int conTimeout;
    int queryTimeout;
    int retry;
    int retryDelay;
```
Object data;
Object oldData;

Socket sock;

DBClient(String[] urlList, int numUrls, int period, int conTimeout, int queryTimeout, int retry, int retryDelay) {
    int port = 0;
    URL u = null;

    this.numUrls = numUrls;
    this.period = period;
    this.conTimeout = conTimeout;
    this.queryTimeout = queryTimeout;
    this.retry = retry;
    this.retryDelay = retryDelay;
    this.urlList = new URL[256];

    for (int i = 0; i < numUrls; i++) {
        try {
            this.urlList[i] = new URL(urlList[i]);
        } catch (MalformedURLException e) {
            System.err.println(e);
        }
    }

    start();
}

/**
 * Main function gets file of URLs and parameters and passes all to
 * instance of DBClient
 */
public static void main(String[] args) {
    BufferedReader br = null;
    String line = null;
    String[] urlList = new String[256];
    DBClient dbc = null;
    int numUrls = 0;
// Get url list file
if (args.length != 6) {
    System.err.println("Usage: java DBClient url_list " +
    "period connection_timeout read_timeout retry retry_delay");
    System.exit(1);
}

// Get url list file
try {
    br = new BufferedReader(new FileReader(args[0]));
} catch (FileNotFoundException e) {
    System.err.println(e);
    System.exit(1);
}

// Read url list file
do {
    try {
        line = br.readLine();
        if (line != null) {
            urlList[numUrls++] = line;
        }
    } catch (IOException e) {
        System.err.println(e);
    }
} while (line != null);

// Run DBClient
dbc = new DBClient(urlList, numUrls, Integer.parseInt(args[1]),
    Integer.parseInt(args[2]), Integer.parseInt(args[3]),
    Integer.parseInt(args[4]), Integer.parseInt(args[5]));

/**
 * Main procedure of DBClient
 */
public void run()
{
    BufferedReader in = null;
    PrintWriter out = null;
long startTime = 0;
int index = 0;

/* Test
for(int i=0; i < numUrls; i++)
    System.out.println( urlList[i]); */

// Send query and receive data periodically
while(true) {
    startTime = System.currentTimeMillis();

    try {
        System.err.println("-----------------------------");
        index = connect(); // Fault tolerant connection
        sock.setSoTimeout(queryTimeout); // Set timeout for receiving data

        in = new BufferedReader (new InputStreamReader(
            sock.getInputStream()));
        out = new PrintWriter(new OutputStreamWriter(
            sock.getOutputStream()));

        // Send query to URL
        out.println("GET /" + urlList[index].getFile());
        out.flush();
        System.err.println("Query sent:" + "GET" + urlList[index]);

        // Receive data from the URL
        data = getData(in);
        System.err.println("Data received:" + data);
        oldData = data;
        process(data);

        // Close the connection
        in.close();
        out.close();
        sock.close();
        sleep(period - (System.currentTimeMillis() - startTime));
    } catch( InterruptedException e) {
        System.err.println(e);
    }
}
} catch( IllegalArgumentException e1) {
    System.err.println(e1);
} catch( UnknownHostException e) {
    System.err.println(e);
} catch( IOException e) {
    System.err.println(e);
}

int connect() {
    String url = null;
    String host = null;
    int port = 0;
    int index = 0;
    ConnectTry ct = null;

    // Select random url and read content from the url.
    while(true) {
        index = (int)(Math.random() * numUrls); // 0 <= random < 1
        host = urlList[index].getHost();
        port = urlList[index].getPort();
        port = (port == -1) ? 80 : port;

        ct = new ConnectTry(host, port);
        try {
            ct.join(conTimeout);
            if( ct.result() == null) {
                System.err.println("Connection timeout(" + conTimeout + ") to " + urlList[index]);
                ct.interrupt();
                continue;
            }
            sock = ct.result();
        } catch( InterruptedException e) {
            
        }
        break;
    } catch( InterruptedException e) {
        
    }
}
return index;
 }

Object getData(BufferedReader in) {
 int r = 0;

 while (r < retry) {
   try{
     data = in.readLine();
     oldData = data;
     return data;
   } catch(InterruptedException e) {
     System.err.println("Query Timeout(" + queryTimeout+ "") happen!");
     try{ sleep(retryDelay);} 
     catch(Exception e2) {} 
     r ++;
   } catch(IOException e3) {} 
 }

 System.err.println("Old data is used due to max query failure"); 
 return oldData;
 }

Object process(Object o) {
 return null;
 }
}

class ConnectTry extends Thread {
   Socket s;
   String host;
   int port;

   ConnectTry(String host, int port) {
    this.host = host;
    this.port = port;
    start();
}
public Socket result() {
    return s;
}

public void run() {
    try{
        s = new Socket(host,port);
        if( this.currentThread().isInterrupted() ) {
            s.close();
        }
    } catch(InterruptedIOException e) { }
    catch( SocketException e) {System.err.println(e);}  
catch(UnknownHostException e) {System.err.println(e);}  
catch(IOException e) { System.err.println(e);}

    }
}