INFORMATION EXTRACTION FOR RUN-TIME FORMAL ANALYSIS

MOONJOO KIM

A DISSERTATION

in

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

2001

Sampath Kannan
Supervisor of Dissertation

Insup Lee
Supervisor of Dissertation

Val Tannen
Graduate Group Chairperson

Acknowledgements

O LORD, our Lord, how majestic is thy name in all the earth!
Thou whose glory above the heavens is chanted
by the mouth of babes and infants, thou hast founded a bulwark
because of thy foes, to still the enemy and the avenger.
When I look at thy heavens, the work of thy fingers,
the moon and the stars which thou hast established;
what is man that thou art mindful of him, and the son of man
that thou dost care for him? Psalm 8

Glory and thanks-giving to the One and only God, who called me to U. Penn and opened my eyes toward Him. For 22 years, He has lead His prodigal son silently to let him know His amazing love. As Joseph prospered because the Lord was with him, I have had an excellent time in Philadelphia with His presence. Considering my limited ability and narrow character, I achieved great things in both academics and human relationships. I cannot help saying that the Lord has done so wonderful things to me for His mercy and grace. He has protected me under His wings and cared me for past years.

I would like to express deep appreciation to my advisors Prof. Insup Lee and Prof. Sampath Kannan. When I was in early stage of Ph.D study, I had not followed their guidelines and had insisted my own way due to my pride and shortsight. They put up with me for a long time with patience and advice even when I resisted listening to their advice. During a painful self-breaking period, they faithfully helped me to recover my project. Without such care, I would not have grown up as a Ph.D researcher. Furthermore, the high level research direction they have suggested has been a guiding light to me when I myself was not sure about what I was doing. I never have accomplished such academic achievement as well as developed personal traits like patience and steadiness without Prof. Lee and Prof. Kannan.

Another person I would like to give deep thanks is my officemate Dr. Mahesh Viswanathan. We spent most days in the same office for 3 years. From him I have learned mathematical thinking as well as how to deliver my technical ideas to others from him. Without his insight and mathematical construction of the run-time formal analysis, MaC architecture could not have originated. Furthermore, he has been a great squash partner. Dr. Oleg Sokolsky has been a great aid in the MaC research. He has helped me to get a better understanding of the Java-MaC implementation. His knowledge on both mathematic theory and system implementation has inspired me. I also appreciate Yerang Hur for his encouragement with his experience of the similar difficulty I have in academic life. He also

ABSTRACT
INFORMATION EXTRACTION FOR
RUN-TIME FORMAL ANALYSIS
Moonjoo Kim
Supervisor: Sampath Kannan and Insup Lee

The rapid increase in the significance of software systems has made software assurance a critical requirement in the information age. Formal verification of system design and testing system implementation with a variety of inputs have been used for this purpose. However, verifying a design cannot guarantee the correctness of an implementation. Although testing is performed on an implementation, it does not give formal guarantees because it is impossible to test exhaustively. We propose a complementary solution to the weaknesses of formal verification and testing by monitoring execution of a program and checking its correctness against formally specified properties at run-time. We call this methodology *run-time formal analysis*. Run-time formal analysis aims to assure the correctness of the current execution at run-time. Run-time formal analysis is performed based on a formal specification of system requirements.

We investigate general issues for run-time formal analysis. We show that the set of properties that run-time formal analysis can detect is a subset of safety properties. Furthermore, we show that the checking of a property written in an expressive specification language such as CCS is NP-complete due to nondeterminism. Finally, we discuss the abstraction of the program execution for reducing the amount of data being monitored and analyzed.

We have designed a Monitoring and Checking (MaC) architecture for run-time formal analysis. A salient aspect of the MaC architecture is the use of a formal requirement specification to check run-time execution of the target program. For specifying formal requirements, we have designed the Primitive Event Definition Language (PEDL) and the Meta Event Definition Language (MEDL). Another important aspect of the MaC architecture is its flexibility. The architecture clearly separates monitoring implementation-dependent low-level behavior and checking high-level behavior with regard to formal requirement specifications. This modularity allows the architecture to be extended for broad target application areas. In addition, the architecture instruments the target program and analyzes the execution of the target program automatically based on given formal requirement specifications. We have implemented a MaC prototype for Java programs called Java-MaC and showed the effectiveness of the MaC architecture through several case studies.

The main thesis of this dissertation is that run-time formal analysis can assure users of the correctness of software systems in a practical manner that is flexible, automatic, and easy to use. This dissertation describes the issues and design solution of the MaC architecture to support this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation presents a methodology and an architecture for assuring the correct execution of a program at run-time. We propose a methodology of monitoring the execution of a program and checking its correctness against formally specified requirements at run-time, called *run-time formal analysis*. In this dissertation, we discuss the fundamentals of the methodology. Then, we describe an architecture for the methodology, called *Monitoring and Checking (MaC)* architecture. We show the effectiveness of the MaC architecture by implementing a MaC prototype for Java programs, called *Java-MaC*. This chapter describes the difficulties faced by the software engineering field with regard to reliability and the motivation for this work. This chapter concludes with contributions of this research to the field of software engineering in terms of increasing reliability.

1.1 Assurance on the Correctness of Software Systems

When compared to software engineering's significance and broad application areas, the field's ability to provide secure and reliable products seems greatly lacking. This deficiency has become a great danger in today's world. Specifically, safety critical software systems such as airplane controllers or nuclear reactor controllers can devastate human lives and properties when they fail to behave correctly. We have seen many disasters due to the incorrect execution of software systems [Lev95], including the tragic accident of the Ariane 5 flight 501 [Ari96]. The report "Information Technology Research: Investing in Our Future " from PITAC (President's Information Technology Advisory Committee) [Pre99] points out this deficiency of the field clearly.

> **Priorities for Research**
>
> ...
>
> **Software** - The demand for software has grown far faster than our ability to produce it. Furthermore, the Nation needs software that is far more usable, reliable, and powerful than what is being produced today. We have become dangerously dependent on large software systems whose behavior is not well understood and which often fail in unpredicted ways ... it has become clear that the processes of developing, testing, and maintaining software must change. We need scientifically sound approaches to software development that will enable meaningful and

1

practical testing for consistency of specifications and implementations...[1]

As the above quotation of the PITAC report mentions, we have a critical need to be confident of the correct execution of software systems. There has long been active research in the field of formal verification and testing for this purpose. For the past twenty years, formal verification has enlarged its domain of application from a humble beginning with just toy examples. Testing has been a standard practice to ensure the correctness of software. Formal verification and testing, however, both have limitations.

1.2 Necessity of Run-time Formal Analysis

Formal verification and testing are performed to ensure the correctness of a program before the program is put into the real environment. They have, however, their own limitations in terms of effectiveness. The report from PITAC [Pre99] indicates these limitations.

> **Finding: Technologies to build reliable and secure software are inadequate...** Having meaningful and standardized behavioral specifications would make it feasible to determine the properties of a software system and enable more thorough and less costly testing. Unfortunately such specifications are rarely used. Even less frequently is there a correspondence between a design specification and the software itself. Often software behavior and flaws are observable only when the program is run, and even then may be invisible except under certain unusual conditions. Programs written in such circumstances frustrate attempts to create robust systems and are inherently fragile ...[2]

Formal verification requires a design specification of software in a formal language. Design specifications, however, do not exist in most cases and there is a huge gap between a design specification and the software itself [Kah99]. Furthermore, because of the difficulty of using formalisms and the limitation on the size of the model, formal verification is still not a feasible solution in general. Testing has been the usual choice for ensuring the correctness of software. Testing, however, cannot guarantee the correctness of programs completely because exhaustive testing is infeasible [Mye76, Mye79]. Therefore, we cannot rely solely on these two methods for ensuring the correctness of software execution.

We propose a complementary solution to the weaknesses of formal verification and testing by monitoring the execution of a program and checking its correctness against formally specified requirements at run-time. We call this methodology *run-time formal analysis*. Run-time formal analysis convinces users of the run-time compliance of an execution of a system with its formal requirement. Run-time formal analysis takes a program and a formal requirement specification for the program as inputs. When a program is running, the execution of the program is checked against the formal requirement specification at run-time. The purpose of run-time formal analysis is to cover the area not covered by formal verification and testing. The aim of formal verification and testing is to check whether *a program* is correct. In other words, formal verification and testing attempt to ensure that *all possible executions* of software yield correct results. Compared to this, run-time

---

[1] Quoted from the page 3 of chapter "1. Information Technology: Transforming our Society" of [Pre99]
[2] Quoted from the page 24 of chapter "3. Technical Research Priorities" of [Pre99]

formal analysis assures users of the current execution of a program. While a program is running under the supervision of a monitor and a checker, we can provide confidence that the program has run correctly *so far*. The MaC architecture has been developed as an architecture for run-time formal analysis to give assurance to the user on the correctness of software execution. The MaC architecture especially targets safety critical applications.

People may raise the following four objections about run-time formal analysis.

- *Run-time formal analysis is not very useful in a sense that simply detecting an error does not help; if the system has crashed, just saying, "system crashed" is not helpful.*

  This is *not* true because run-time formal analysis helps users to detect and correct errors. There are cases when users cannot detect errors because of the errors' subtlety. For example, the pentium floating point unit bug was so subtle that it was not discovered for a long time. Such errors may not cause disastrous failure to the system immediately. Run-time formal analysis can find such subtle errors and help users to take a recovery action before critical failure happens to the system.

- *Run-time formal analysis requires replacement of run-time execution environments such as VM/OS/HW with ones specialized for extracting information from the execution of the target program.*

  This is *not* necessarily true. Modification of run-time environments depends on the architecture employed for run-time formal analysis. For example, the MaC architecture does not require any change in run-time execution environments.

- *Run-time formal analysis can cause undesirable side effects to a target program; run-time formal analysis can slow down the target program and alter the behavior of the program.*

  This is a *universal* problem to all run-time analysis methods unless specialized hardware is utilized. Delay in target program execution due to probes might alternate behavior of concurrent programs. This is the biggest concern in practice. Careful engineering of an architecture for the analysis may reduce side effects to an acceptable degree and minimize the possibility of altering the behavior of a target program.

- *Run-time formal analysis cannot guarantee the correctness of future execution of a target program.*

  Run-time formal analysis is not a panacea for all software reliability problems. We believe, however, that run-time formal analysis can increase the reliability of the program execution as a complementary solution to formal verifications and testing.

1.3 Scope of the Dissertation

This dissertation addresses the following three areas of the MaC architecture as depicted in Figure 1.1.

- **Formal requirement specification**

  This dissertation discusses what properties run-time formal analysis can validate.

Formal Requirement Specification
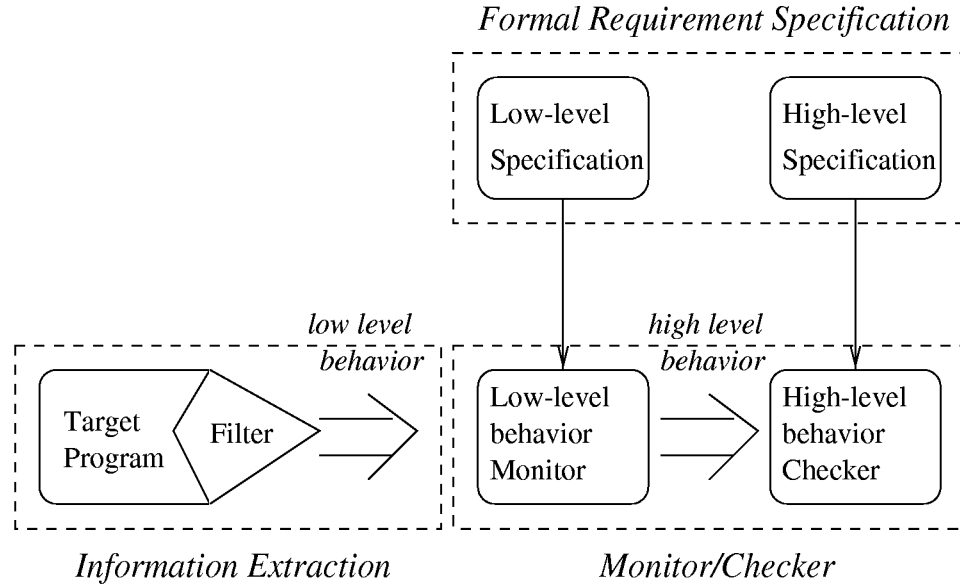
Figure 1.1: Three areas of the MaC architecture

Furthermore, we define two specification languages - a high-level specification language and a low-level specification language - to describe formal requirement specifications. We specify formal requirement specifications in these two languages based on instantaneous _events_ and continuous _conditions_. A formal requirement specification consists of a high-level specification which has requirement properties and a low-level specification which contains the definitions of primitive events and conditions used in the high-level specification in terms of the low-level behavior of the target program.

- **Monitor and checker**
  The MaC architecture monitors and checks the execution of the target program through a low-level behavior _monitor_ and a high-level behavior _checker_. The monitor monitors low-level behavior of the instrumented target program such as the updates of the program variables. The monitor maps this low-level behavior to primitive events and conditions which constitute the high-level behavior of the target program according to their definitions in a low-level specification. These primitive events and conditions are fed into the checker which checks whether the execution violates a high-level specification.

- **Information extraction**
  The target program is instrumented to extract and report its low-level behavior to the monitor. A _filter_ denotes the set of probes inserted in the target program through instrumentation. The filter extracts snapshots of the target program and sends these snapshots to the monitor. It abstracts out irrelevant snapshots to decrease the volume of snapshots to be delivered to the monitor. As a consequence, this abstraction decreases the volume of snapshots to be analyzed by the monitor.

The following issues are not addressed in this dissertation.

- *Hardware or Hybrid monitor.* This dissertation considers run-time formal analysis performed by software only but can be extended to use hybrid monitors in the future.

- *Parallel/distributed programs.* This research applies to multi-threaded programs but does not handle issues specific to parallel programs, such as process migration and clock synchronization.

- *Hard real-time system.* This dissertation does not examine the issue of the timing perturbation to hard real-time systems.

1.4 Contributions

The main thesis of this dissertation is

Run-time formal analysis can assure users of the correctness of software systems in a practical manner which is flexible, automatic and easy to use.

The contributions of this research are as follows:

- **Rigorous analysis**
  The salient feature of the MaC architecture is the use of a formal requirement specification to check the execution of the target program at run-time. For specifying requirements unambiguously, we have defined two specification languages. In addition, the tools of the MaC architecture follow the formal requirement specifications without requiring human interaction. This leads to accurate analysis because automatic instrumentation, monitoring, and checking eliminate slippery errors caused by human intervention in these procedures.

- **Flexibility**
  The MaC architecture is a *modular* architecture. The MaC architecture separates monitoring program-dependent, low-level behavior from checking high-level behavior in both its specification languages and its run-time analysis components. This modularity is illustrated clearly in Figure 1.1. First, the architecture provides two specification languages - a high-level specification language called Meta Event Definition Language (MEDL) and a low-level specification language called Primitive Event Definition Language (PEDL). This separation enables reuse of the high-level specification even when the low-level specification is changed due to the change of the target program. Second, the architecture separates analysis components such as a filter, a monitor, and a checker. This separation and the well-defined interfaces (see Appendix A and B) among the components make the MaC architecture an open architecture which can incorporate third-party tools (for example, see Section 8.3).

- **Automation**
  The analysis procedure of the MaC architecture is fully automatic. First, the target program is instrumented automatically according to a low-level specification. Second, the architecture monitors and checks the execution of a target program automatically following formal requirement specifications.

5

- **Ease of use**

  The MaC architecture is easy to set up.

  The MaC architecture does not require a specialized execution environment such as specialized VM/OS/HW. In addition, the analysis procedure including instrumentation, monitoring, and checking are performed automatically without requiring human directions. Furthermore, the executable code of a target program is instrumented in the MaC architecture. Users do not need complex source code recompilation which is mandatory if source code is instrumented.

- **Generality**

  Instrumentation, monitoring, and checking by the MaC architecture are application-independent and comprehensive. In addition, the architecture instruments executable code, not source code which is available only to the developers. Furthermore, the openness of the architecture makes the architecture extendible for various application areas by incorporating different tools from outside. Finally, with the help of overhead reduction techniques, Java-MaC imposes only modest run-time overhead to the target program. These features make the MaC architecture apply not only to specific toy examples, but to broad application areas. This generality has been shown by case studies on different application areas including network protocol and hybrid system simulation (see Chapter 8).

1.5   Outline of the Thesis

Chapter 2 describes the analysis methods for the correctness of software. First, we classify related work according to the representation of target systems: abstract model vs. implementation. In this chapter, we evaluate related work, which helps the reader to understand the design rationale of the MaC architecture.

Chapter 3 identifies fundamental issues in run-time formal analysis. Issues in this chapter are independent of specific target system architecture. These issues include monitorable properties, property specification language, and abstraction of the target program execution.

Chapter 4 gives an overview of the MaC architecture. This chapter includes the structure of the architecture as well as the language definitions of PEDL and MEDL. This chapter is from [KVBA+99, LKK+99, KVBA+98] which are coauthored by H. Ben-Abdallah, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan.

Chapter 5 focuses on how to monitor Java programs. This chapter discusses issues on monitoring objects. It describes a low-level specification language for Java programs, called PEDL for Java.

Chapter 6 focuses on the Java-MaC prototype which is built on the mechanism described in Chapter 5. Details of run-time components of Java-MaC and instrumentation process are described.

Chapter 7 analyzes the run-time overhead caused by Java-MaC and provides several overhead reduction techniques.

Chapter 8 demonstrates the effectiveness of Java-MaC by illustrating case studies. The case studies include an emulator of mobile physical agents, a network protocol, and hybrid

system simulation.

Finally, Chapter 9 gives the summary of what has been achieved in this dissertation and outlines the work yet to be done as future work.

Chapter 2

Background and Related Work

In this chapter, we review analysis methods for checking the correctness of programs. First, we classify the analysis methods into two groups: methods targeting a model of a program (see Section 2.1) and methods targeting an implementation of a program (see Section 2.2). We will see the strong points and the weak points for each of these two groups of methods in the following subsections. Section 2.1 describes analysis methods applied to the models of programs. The purpose of Section 2.1 is to give readers a brief background on analysis methods for models, which helps one to understand the necessity for analysis methods on implementation. Section 2.2 describes analysis methods targeting implementations. In Section 2.2, we describe the goals of these methods and the approaches these methods have taken.

2.1 Analysis of Model

The program development process begins with the model of a program and desired properties for the program. *Formal methods* are mathematical techniques that describe the models of programs and desired properties. In addition, formal methods can be used to verify the correctness of a program in terms of given properties. A method is formal if it has a sound mathematical basis, typically given by a formal specification language.

A strong point of formal methods is that formal methods reveal the ambiguity, incompleteness, and inconsistency in the model of a program. When formal methods are used in the early program development process, they can reveal design flaws that might otherwise be discovered only during costly testing and debugging phases. During the past two decades, tools dedicated to formal methods have been introduced. There have been an increasing number of successful industry case studies using formal methods [CW96, WTK00].

A weak point of this approach, however, is that the correctness of a model does not necessarily mean that the implementation is correct. This is because the implementation has more detail than the model and is susceptible to errors not present in the model. In addition, there are negative aspects of formal methods including hard to learn/use notations [Bow95, DS97], high computing power requirements, and primitive tools inadequate for practical usage [GKND97, DS97].

Two common formal paradigms are *theorem proving* and *model checking*.

2.1.1 Theorem Proving

The goal of theorem proving is to prove whether a model satisfies given properties. The approach of theorem proving is to prove $M \vdash p$ by mathematical deduction where M is a model of a program given as characteristic statements on the program and p is a property.

A strong point of theorem proving over model checking is that theorem proving provides better scalability to handle infinite state models by induction. Another strong point is that theorem proving uses a more expressive language which includes quantifiers, leading to succinct specification of parameterized systems [Pnu99].

A weak point of theorem proving is that theorem proving is undecidable. Due to this limitation, completely automated theorem provers are infeasible. Therefore, theorem proving requires user ingenuity and interaction. Another weak point is that theorem provers suffer from the inability to find counter examples. For a list of theorem provers, see [FMP95].

2.1.2 Model Checking

The goal of model checking is similar to that of theorem proving - to prove whether a model satisfies given properties. However, the approach of model checking is different from that of theorem proving. A model checker provides a *design specification language* for describing a model of a system, which is similar to a programming language but simplified. Also, a model checker provides a *property specification language* for describing desired properties. A property specification language may be the same as the design specification language. A model checker generates and explores all possible/reachable states of the model and checks whether all the states satisfy the given properties.

A strong point of model checking compared to theorem proving is that describing a model in a design specification language is more familiar to a programmer than describing a model as characteristic statements. Another advantage is that model checking provides automatic verification. In other words, given a model and properties written in formal specification language(s), the verification process does not require user interaction. Furthermore, model checking can give counter examples which are useful for debugging.

A weak point of model checking is that tractable reachability testing algorithms exist for only very simple systems [CK96, CGL94, Kur94, AH98]. In many cases, even those simple systems require large amount of computational resources. (there has been active research on state reduction techniques [Kur87], such as symbolic model checking [BCD⁺90, McM93], binary decision diagram reduction [Bry86], and partial order reduction [Pel94]). To avoid the state explosion problem [YY91], design specification languages put restrictions on data types and the expressive power of the languages [ACHH93, AH96, AH98]. Consequently, approximation of a model is required and modeling becomes difficult [AEK⁺99, AEK⁺00]. For a list of model checkers, see [FMP95].

2.2 Analysis of Implementation

Methods targeting the implementation of a program check whether executions of the program satisfy given properties. These methods are applied in later stages of the program development process because they require the implementation of the program.

A strong point of these methods is that they can check actual executions of a program. Although a model is proven correct, however, it does not necessarily mean that the implementation is correct; the implementation has more detail than the model and is susceptible to errors not present in the model. For example, a model may implicitly assume a natural number can be arbitrarily large. However, an implementation may restrict a natural number to be less than 2^{32} because the implementation chooses to represent a natural number in 32 bits. Then, overflow error which may happen in the implementation cannot be detected in the model. Furthermore, there is no guarantee that implementation is strictly following a model because there can be errors caused by human programmers who implement the program after understanding the model.

These methods, however, do not provide mathematical guarantees on the correctness of the target program as the analysis methods targeting the model do. Another negative when compared to the analysis methods targeting the model of a program is that these methods are applied in the late stages of program development process, which leads to costly debugging and re-implementing the program.

These methods support different levels of monitoring according to application purpose. A method can monitor *process-level* behavior (for example, communication between processes), *statement-level* behavior (for example, method invocations or assignments), or *instruction-level* behavior (for example, step-by-step instruction trace) [TFC90]. The finest-level target activity of statement-level monitoring is the execution of a statement in a source code. That of instruction-level monitoring is the execution of an instruction. A statement consists of several instructions. For example, a Java statement `a.b.x = 10` consists of the following three Java bytecode instructions `getfield A/a;getfield B/b;putfield I/x 10`. In this example, instruction-level monitoring targets single instruction such as `putfield I/x 10`. Compared to instruction-level monitoring, statement-level monitoring needs to recognize all three consecutive instructions forming the statement. We classify analysis methods on implementation into the following three groups according to the level of monitoring.

- process-level monitoring

- statement-level monitoring

- instruction-level monitoring

The level of monitoring affects the amount of overhead and the instrumentation stages in general. Higher-level monitoring causes less overhead because higher-level behavior generates less frequent events than lower-level behavior. In addition, location of instrumentation is decided by the level of monitoring in general. For example, analysis methods for process-level behavior instrument the communication interface between processes. Analysis methods for instruction-level activity instrument the run-time execution environment on which each instruction of the target program is executed. We will see the details of these three levels of monitoring in the following subsections.

2.2.1 Features to be Examined

This section describes features to be examined in the following sections on analysis methods for implementation.

- *Low-level specification language*: a language used to define primitive events used in a requirements specification in terms of low-level program entities such as variables and methods.

- *High-level specification language*: a specification language dedicated to describing requirement specifications in terms of high-level events.

- *Code size increase*: the amount the size of the target program increases due to inserted probes which reports snapshots of the target program to the monitor.

- *Overhead to the target system*: amount by which the execution speed of the target program is slowed down due to monitoring activity.

- *Modification of execution environment*: the necessity of specialized run-time execution environment such as OS/VM/communication channel.

- *Frequency of events*: the frequency of events reported to a monitor.

- *Complexity of instrumentation*: the complexity regarding where to insert what probes into the target program.

2.2.2 Process-level Monitoring

The goal of this group of methods is to monitor externally observable behavior of processes such as input/output or communication between processes.

A strong point of these methods is that they decrease the difficulty of instrumenting target program by instrumenting only an interface of a process which is well defined and observable outside the process. Another benefit is its low overhead cost. The external behavior of a process generates less frequent events compared to the internal behavior. This leads to less overhead in the instrumented target program.

A drawback to these methods is that they check only external behavior of processes. They cannot check desirable properties concerning internal behavior of a process. For example, suppose a process has a stack implementing push() and pop(). A non-emptiness property stating that the number of pop()'s should be less than the number of push()'s cannot be checked in process-level monitoring. In other words, process-level monitoring cannot detect a violation in internal behavior which does not effect external behavior or at least not until the violation corrupts external behavior eventually. Even when an error in external behavior is detected, the source of the error inside of a process is hard to find. Some systems performing process-level monitoring are described below.

- **Model-based testing** aims to detect violations by using a formal model of the program. Testing oracle takes input/output execution of a program and check whether the execution is correct with regard to the formal model of the program. The difference between these works is mainly the formalism they use to describe a model. Supervisor [SS98] compares the output of a target program and expected output generated by interpreting the model of the program written in SDL [SDL89]. Dillon [DY94] develops an algorithm to generate a testing oracle from a Graphical Interval Logic specification [DKM+94]. [DR96] has a tableau algorithm to generate

a testing oracle from a temporal logic specification [Pnu77]. Parissis [PO96] presents a similar tableau algorithmic technique using LUSTRE [HCRP91]. Clarke [CL97] conducts a case study on automatic testing of Philips Audio Control Protocol using a specification written in ACSR [BGLG93].

- The goal of **MOTEL** (MOnitoring and TEsting tooL) [Log00] is to detect violations of communication behavior between distributed processes. A target program is continuously monitored and checked against properties written in LTL [MP92]. MOTEL assumes that a communication medium among processes is a CORBA event channel. MOTEL modifies CORBA middleware in order to intercept messages among components.

  Strengths of MOTEL include its independence of implementation and scalability. These properties stem from the fact that MOTEL observes only the interfaces of objects only, not the details of objects. A weakness is that MOTEL cannot handle events concerning the internal behavior of objects.

- **JEM** (Java Event Monitor) [LM99] is an event-mediator type of system like the CORBA event channel. In other words, JEM provides its own communication channel similar to the CORBA event channel. JEM receives predefined primitive events from event suppliers and detects composite events written in a Java Event Specification Language [LMK98] based on primitive events. A benefit of using its own event channel rather than a standard event channel is freedom in defining primitive events. Using its own event channel, however, does not scale well compared to using a standard event channel such as the CORBA event channel. A large distributed system may consist of many components from different vendors. Without using a standard event channel, inter-operation between the components of different vendors is difficult to achieve.

2.2.3 Statement-level Monitoring

The goal of this group of methods is to monitor internal behavior of processes at a statement-level. For example, a non-emptiness property that the number of `pop()`'s should be less than the number of `push()`'s in a stack can be monitored at this level of monitoring.

A strong point of these methods is that these methods can check various desirable properties not limited to properties on external behavior. A weak point of these methods is that the instrumentation of a target program for monitoring various activities is a complicated task. Deciding where to insert probes into a program in general is a nontrivial task [Sno88] because it requires knowledge of program structure. Furthermore, instrumentation may modify the semantics of the target program if thorough care is not taken. For example, suppose a `return` statement is inserted into a method accidently. The statements of the method after the `return` would not be executed. This is a serious problem especially for manual instrumentation. Second, source code is usually only available to the developers of a program. In many cases, though not all cases, these methods require the source code of a program in order to recognize and instrument target statements. The MaC architecture belongs to this category. We will make brief comparison between these works and the MaC architecture.

- The goal of **ALAMO (A Lightweight Architecture for MOnitoring)** [Jef93, JZTB98, TJ98] is to reduce the difficulties in writing monitoring tools by constructing a platform on which monitor construction is relatively easy. Another goal is to access and modify target program states including local variables and the sequence of instructions of the target program. The configuration language controls which activities are to be recognized as events. The events are defined by target language statements. For example, procedure calls, memory references, and assignments statements can be defined as events. The configuration specifies events as pairs consisting of an event code (type of statement) and an event value which depends on the corresponding event code. ALAMO instruments the C source code following the configuration automatically.

The ALAMO monitor runs in the same address space with the target program. When an event is reported to the monitor, execution control is given to the monitor and the target program is suspended. During the execution of the monitor, the monitor can directly access and modify the target program if necessary. This activity is performed by scanning a run-time image of target program as an array of bytes because the target program and the monitor exist in the same address space. After the monitor finishes its execution, the control is given back to the suspended target program and the monitor is suspended. This execution model prevents the monitor from reading inconsistent states of the target program. The monitor is written in C by a user.

A strong point of ALAMO is that its automatic instrumentation of C programs prevents the slippery errors possible in manual instrumentation. Another strength is its accessibility and control of all target program states.

The first drawback is the 2–3 orders of magnitude overhead in execution speed it adds. Accessibility to the entire set of states in the target program causes this high overhead because the ALAMO monitor and the target program use same address space which requires memory protection and explicit context switching. The MaC architecture does not aim to achieve such thorough inspection power. The MaC architecture monitors events generated from three types of statements: *variable assignments*, *method invocation*, and *method returns*. Thus, the MaC architecture does not cause such high overhead.

The second weakness is that the ALAMO architecture is platform and target programming language specific because it uses its own platform specific program loader to load a target program and a monitor in the same address space.

The third negative is that the monitor should be written by a user in C; erroneous implementation of the monitor can cause the crash of both the target program and the monitor. A monitor and a checker of the MaC architecture are generated automatically from the requirement specifications.

The final weak point is that ALAMO does not monitor handle aliasing because it monitors only textually recognizable entities. For example, suppose a configuration specifies the assignment statement of int x in the record r1 to be monitored. After r2 = r1;, an assignment r2.x = 1; changes the x. However, ALAMO cannot recognize that this statement updates the x being monitored. Java-MaC identifies

13

an object with its address on the heap memory, not by its syntactic name (see Section 5.1).

- **The Java Run-time Timing constraint Monitor (JRTM)** [ML97a, ML97b] aims to detect violation of timing properties. JRTM uses Real-Time Logic (RTL) [JG90] as a property specification language. Timing constraints can specify relationship between ith occurrence time of an event $e1$ and the j th occurrence time of an event $e2$. Given timing constraints, JRTM generates implicit constraints which are logically implied constraints from the explicit constraints. From these explicit/implicit constraints, JRTM generates *constraint graphs*. JRTM develops efficient algorithms to catch violations of timing constraints at the earliest possible time based on these constraint graphs. A Java program is manually instrumented with a probe put in the place where a primitive event happens. The probe sends the event and the monitor receives it and checks constraint graphs to detect the violation of constraints.

  A strong point of JRTM is its guarantee of timely detection for violations of timing properties. A weak point is that JRTM does not provide a low-level specification language. Based on informal definitions of primitive events, the target program is instrumented manually. Unless instrumentation is performed correctly, the analysis cannot produce the correct result. The MaC architecture provides a low-level specification language to define primitive events In addition, the MaC architecture instruments the target program automatically according to the definitions of primitive events.

- **The Sentry System** [CG92, CG95, CG96] aims at a low-cost, low-precision monitor. Sentry is a monitor watching over the behavior of a target program continuously. A target program is instrumented to extract snapshots of variable values into shared storage between the program and the Sentry. Key features of the Sentry system are

  - linear amount of shared storage in the number of monitored program variables
  - wait-freedom (the program being monitored never waits for the sentry)
  - mutual exclusion (a snapshot being read by the sentry is not overwritten by the program)

  These features are accomplished by using non-blocking buffers between the target program and Sentry. When the target program writes snapshots into a buffer $b1$, Sentry reads from a buffer $b2$. When $b1$ becomes full, the target program starts writing into $b2$ even when Sentry does not read all the content of $b2$ yet. Then, Sentry reads from $b1$. This alternation of buffers in writing and reading guarantees consistent snapshot reading. Sentry, however, provides only *weak* completeness of violation detection due to non-blocking buffer. In other words, Sentry can miss violations unless the violations persists in the execution. This is because Sentry loses snapshots when the writing speed of a program is faster than reading speed of Sentry.

  The target C program is annotated by a user to describe assertions over program variables to be checked at run-time. Then the annotated target program is compiled by the Sentry compiler to generate an instrumented target program and Sentry code. The instrumented target program writes its snapshots concerning the assertions into

a shared memory. The Sentry reads snapshots from the shared memory and evaluates properties.

A strong point of Sentry is its low-overhead cost with linear amount of shared storage and wait-freedom. A weak point is its weak completeness. Missing a single violation may result in losing a chance to prevent system crash.

- The goal of **Time Rover** [Rov97] is to detect the violations of assertions over target program variables. Assertions are written in temporal logic at run-time. Time Rover uses a normal temporal logic and metric temporal logic which extends the temporal logic by supporting the specification of real-time constraints to temporal operators. Also, the temporal logic used by Time Rover supports counting operators so that a user can specify the number of event occurrences in the assertion. A user has to write and insert assertions into a target source code. Time Rover supports various languages such as Java, C++, and Verilog.

2.2.4 Instruction-level Monitoring

The first goal of this group of analysis methods is to monitor and check the behavior of the target program in terms of instruction-level activities. The second goal is to steer the target program. These methods provide a virtual machine as run-time execution environment to monitor the execution of the target program instructions one by one.

An advantage of these methods is that they do not require the instrumentation of a program because a virtual machine can monitor execution of a target program as it is without instrumentation. This removes the difficulty of instrumenting a target program. In addition, a code size does not increase due to inserted probes. Furthermore, these methods can monitor and check the behavior of the target program at a finer-grained level, compared to the statement-level monitoring methods. Finally, these methods provide the accessibility and controllability of entire set of target program states.

A weak point is that this fine-grain monitoring capability costs a run-time overhead magnitude of 2-3 because of the slow interpretation speed of a virtual machine. In addition, these methods are machine dependent. Furthermore, these methods need a specialized run-time execution environment rather than the existing environment which is reliable and familiar to users. This requirement may prohibit these methods from being widely applied. Finally, reasoning about high-level properties based on instruction-level behavior is not an easy task.

- **Dynascope** [Sos92, Sos95b, Sos95a] serves as an instruction-level behavior monitor such as an array bound checker. Dynascope is similar to the Java architecture [LY99]. As the Java architecture compiles a Java program into bytecode and the Java virtual machine executes the bytecode, the Dynascope architecture compiles ANSI C program into DLX [HP90]-like languages and the Dynascope virtual machine executes the DLX-like codes.The Dynascope architecture provides

  - a virtual machine which executes a DLX-like machine language
  - an ANSI C compiler to generate Dynascope machine code
  - a library of monitoring and steering routines for Dynascope virtual machine

A user writes a monitor in C which communicates with the Dynascope virtual machine using provided monitoring and steering functions.

A strong point of Dynascope is its *hybrid* execution model. The parts of the program which do not need to be monitored are compiled into host machine code, not Dynascope code. These codes are executed directly on the host processor. This alleviates the overhead of interpretation of the target program.

A weak point is that Dynascope does not provide a requirement specification language. A monitor should be written manually in C, which may lead to incorrect analysis results and furthermore crash the target system at run-time. [1]

- The goal of **Dalek** [OCH90, OCH91] is to detect composite events from the instruction-level execution of target program. Dalek is an event-based debugger for C programs, which is based on `gdb`. Dalek provides a debugging language to define primitive events based on instruction-level behavior during the execution of a program. The debugging language provides assignment statements, `if` and `while` statements, local/global variables, and functions. This debugging language can describe composite events based on primitive events. For example, suppose we monitor whether a program frees unallocated address or not. A composite event `mismatch` is raised when an address value of a primitive event `free(address)` does not match an address of any previous primitive event `malloc(address)`.

  A strong point of Dalek is its debugger language. This language is useful compared to that of other conventional debuggers such as `gdb`. Primitive events and composite events can be recognized based on the specification of primitive events written in this debugger language. However, this language is cumbersome to describe complex requirements because the language provides only primitive constructs such as `if/while`.

- **The Java Platform Debugger Architecture (JPDA)** [JVM99] provides a debugging interface so that a programmer can access information available on JVM and manipulate the execution of the target program. The JVM specification itself does not contain a specification for the debugging interface. JPDA consists of three components - Java Virtual Machine Debugger Interface (JVMDI), Java Debug Wire Protocol (JDWP), and Java Debugging Interface (JDI). JVMDI specifies what should be provided by the JVM for supporting the debugger. JDWP defines the format of information and requests to be transfered between the debugging process and the debugger. JDI defines a high-level language interface which tool developers can easily use to write remote debugger applications. All functionalities in JPDA are accessed through the Java Native Interface.

  A strong point of JPDA is its modular approach to enable separation of a debugging

---

[1] One notable thing is that Dynascope had the concept of the platform independent virtual machine and the hybrid execution model similar to a just-in-time compiler much earlier than SUN's JVM. Furthermore, Dynascope has the concept of debugger interface similar to Sun's Java Virtual Machine Debugger Interface. However, Dynascope did not provide a well-designed language for the virtual machine. Dynascope also failed to put emphasis on the platform neutral architecture which might not have been interesting to the community around early 1990. As a result, Dynascope did not succeed as a tool, but JVM did.

process and a target program. Thus, JPDA can build debugging architectures flexibly. For example, a debugging interface of different vendor's VM can be adopted by changing just portion of a debugger written in JVMDI.

A weakness of JPDA is that it does not provide a monitoring language. JPDA provides a platform specific API in C. Therefore, to monitor a platform neutral Java program, a user has to write a platform specific C program, which is neither safe nor convenient.

2.2.5 Summary

This section summaries the features (defined in Section 2.2.1) of analysis methods on implementation which we have discussed. Table 2.1 shows the summary. We use **N** for No or None, **Y** for Yes, **S** for Small, **M** for Medium, **L** for Large in the table.

Related work	Low-level spec lang.	High-level spec lang.	Code size incr-ease	Over-head	Modifi-cation of env.	Freq. of events	Comp-lexity of instr.	Needs source code	Note
Model-based testing	N	Y	S	S	N	Low	Low	Y	Various formal req. spec.
MOTEL	N	Y	N	S	Y	Low	N	N	Modified CORBA
JEM	N	Y	N	S	Y	Low	N	Y	Own comm. channel
ALAMO	Y	N	L	L	N	Med	Auto	Y	Insp. power.
JRTM	N	Y	M	M	N	Med	High	Y	Early. detect. of violation
Sentry	Y	Y	M	S	N	Med	Auto	Y	Low overhead low precision
Time Rover	N	Y	M	M	N	Med	High	Y	Extended Temp. Logic
MaC	Y	Y	M	M	N	Med	Auto	N	Usability
Dynascope	N	N	N	L	Y	High	N	N	Hybrid exec.
Dalek	Y	N	N	L	Y	High	N	N	Spec. lang.
JPDA	N	N	N	L	Y	High	N	N	Modular design

Table 2.1: Summary of analysis methods on implementation

The top three rows in the Table 2.1 are the methods of process-level monitoring. The middle four rows are the methods of statement-level monitoring. The bottom three rows are the methods of instruction-level monitoring. Table 2.1 shows that the overhead and the frequency of events depend on the level of monitoring in general; process-level monitorings have low overhead and less frequent events and instruction-level monitorings have large overhead and frequent events. Note that the advantage of low overhead in Sentry comes with the cost of incomplete analysis because the Sentry may miss snapshots of target program. Also, the high overhead in ALAMO is a tradeoff for the thorough inspecting power.

Chapter 3

Fundamentals of Run-time Formal Analysis

This chapter discusses fundamental issues in run-time formal analysis. The issues in this chapter are not specific to one run-time formal analysis architecture such as the MaC architecture, but universal. In this chapter, we will describe what class of properties run-time formal analysis can check. Then, we will discuss the computational complexity of property evaluation when a property specifies non-deterministic behavior. Then, we will discuss the abstraction of the program execution.

3.1   Monitorable Properties

The first issue is discovering what class of properties can be checked by a run-time formal analysis. We define the term *property* formally.

Definition 1 (Execution) *An execution of a program is an infinite sequence of program states* $\sigma = s_0 s_1...$ *where* $s_i \in S$ *is a set of program states,* $s_0 \in S_{init}$ *is a set of initial states, and* $\sigma[i..j]$ *is the subsequence of* σ *from a state* s_i *to a state* s_j. [1]

Definition 2 (Property) *A* property *is a set of executions. We write* $\sigma \models P$ *to denote that* σ *is in property* P.

The class of properties that a run-time formal analysis can check is the class of *safety properties*. Informally speaking, a safety property means that bad things do not happen during execution of a program. Consider a safety property P_{safe} that means some bad thing x does not happen. If $\sigma \not\models P_{safe}$, σ includes some bad thing which cannot be remedied afterward. In other words, there is some prefix of σ which includes some bad thing for which no extension to an infinite sequence will satisfy P_{safe}. Throughout this chapter, S^ω denotes the set of infinite sequences of states.

---

[1] The definition of an execution can apply to finite sequences by obtaining an infinite sequence from a finite one by repeating the final state of the finite sequence. This corresponds to the view that a terminating execution is the same as non-terminating execution in which after some finite time (once the program has terminated) the state remains fixed.

Definition 3 (Safety Property)[2] *A property* $P \subseteq S^\omega$ *is a safety property if for every* $\sigma \in S^\omega$, $\sigma \in P$ *if and only if* $\forall i \exists \beta \in S^\omega(\sigma[0..i]\beta \in P)$ *where* S *is the set of program states.*

It is clear from Definition 3 that the properties run-time formal analysis can check are safety properties. A monitor can watch only a finite number of execution steps; a monitor can only check a property based on a finite number of states.

However, a safety property is not necessarily a monitorable property. The definition of safety properties makes no computational assumptions; it is possible to define a property that is a safety property, but which is unlikely to be monitorable. [Vis00] shows that the safety closure of the halting problem is a safety property but not a monitorable property [Vis00]. This suggests that the class of *monitorable properties* is a strict subset of a class of safety properties; they should be such that sequences not in the properties should be recognizable by a Turing Machine, after examining a finite prefix. Therefore, we can define a monitorable property as follows. We use pref(σ) for $\sigma \in S^\omega$ as the set of all finite prefixes of σ.

Definition 4 (Monitorable Property)[3] *A property* $P \subseteq S^\omega$ *is said to be* monitorable *if and only if* P *is a safety property and* $S^* \setminus \text{pref}(P)$ *is recursively enumerable, where* $\text{pref}(P) = \cup_{\sigma \in P}\text{pref}(\sigma)$

Figure 3.1 illustrates the relationship of monitorable properties with other properties.
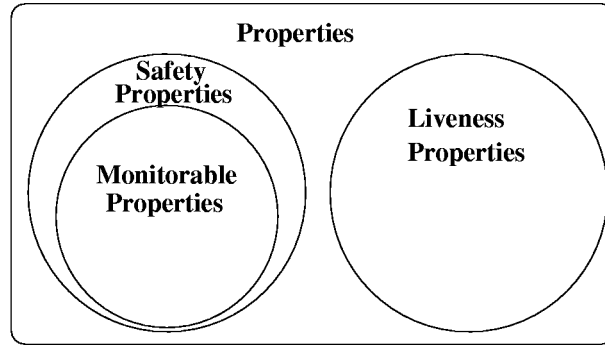
Figure 3.1: Monitorable properties and other properties

3.2 Property Specification Language

Monitorable properties need to be described in a property specification language. The characteristics of the property specification language can affect the computational complexity of evaluating properties. In this subsection, we will discuss the effect of non-determinism in the property specification language.

Run-time formal analysis can be thought of as a *trace validity problem* where a trace is generated from the execution of the program. The trace validity problem is a membership checking problem to determine whether a given trace is in the set of valid traces. For

---

[2]A formal definition of safety property from [AS85].

[3]A formal definition of monitorable property is from [Vis00].

sufficiently expressive requirement specification languages, such as a process algebra, e.g. CCS [Mil89] or ACSR [BGLG93], this problem turns out to be NP-complete. We formulate the trace validity problem using the notation of [Mil89] for the formulation. We will denote the *ith* character in a string x by $x^{(i)}$. $\mathcal{A}$ is an set of names $a, b, c, \ldots$. Then $\overline{\mathcal{A}}$ is the set of *co-names* $\overline{a}, \overline{b}, \overline{c}, \ldots$; $\mathcal{A}$ and $\overline{\mathcal{A}}$ are disjoint and are in bijection via $(^-)$; we declare $\overline{\overline{a}} = a$. $\mathcal{L} = \mathcal{A} \cup \overline{\mathcal{A}}$ denotes the set of labels. We also introduce a distinguished silent action $\tau \notin \mathcal{L}$. We set $Act = \mathcal{L} \cup \{\tau\}$.

Definition 5 *The set of processes is defined by*

$$P \; ::= \; Nil \mid \alpha.P \mid P \; + \; Q \mid P\|Q \mid P\backslash L$$

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

Definition 6 *The labeled transition relation $\overset{\alpha}{\to}$ between two processes is defined by the following rules. In the following rules, $\alpha \in Act, l \in \mathcal{L}$, and $L \subseteq \mathcal{L}$.*

$$[Prefix] \frac{}{\alpha.P \overset{\alpha}{\to} P}$$

$$[Choice] \frac{P \overset{\alpha}{\to} P'}{P + Q \overset{\alpha}{\to} P'} \quad \frac{Q \overset{\alpha}{\to} Q'}{P + Q \overset{\alpha}{\to} Q'}$$

$$[Parallel] \frac{P \overset{\alpha}{\to} P'}{P\|Q \overset{\alpha}{\to} P'\|Q} \quad \frac{Q \overset{\alpha}{\to} Q'}{P\|Q \overset{\alpha}{\to} P\|Q'} \quad \frac{P \overset{l}{\to} P, \,'Q \overset{\overline{l}}{\to} Q'}{P\|Q \overset{\tau}{\to} P'\|Q'}$$

$$[Restriction] \frac{P \overset{\alpha}{\to} P'}{P\backslash L \overset{\alpha}{\to} P'} \; where \; \alpha \notin L \cup \overline{L}$$

Definition 7 *Given processes P and P', and $\alpha \in \mathcal{L}$, we say that $P \overset{\alpha}{\Rightarrow} P'$ if $P(\overset{\tau}{\to})^* \overset{\alpha}{\to} (\overset{\tau}{\to})^* P'$, where $(\overset{\tau}{\to})^*$ is the transitive reflexive closure of $\overset{\tau}{\to}$.*

Definition 8 (Valid Trace) *A string $s \in \mathcal{L}^*$, of length n, is said to be a valid trace of a process P, if there exist processes $P_0, P_1, \ldots, P_n$, such that $P \equiv P_0$, and $P_{(i-1)} \overset{s^{(i)}}{\Rightarrow} P_i$, for all $i \in \{1, \ldots, n\}$*

Then, the Trace Validity Problem is formally defined as follows:
Input   A process P and a string $s \in \mathcal{L}^*$.
Output   Is s a valid trace of P?

Theorem 1 *The trace validity problem is* **NP**-*complete.*

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

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

$$X_i \stackrel{\text{def}}{=} \tau.F_i + \tau.T_i$$
$$F_i \stackrel{\text{def}}{=} \overline{f_i}.F_i$$
$$T_i \stackrel{\text{def}}{=} \overline{t_i}.T_i$$

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

In addition to these processes, we define another process, P. The idea is that P will deadlock, when run concurrently with the processes X_i, iff the truth assignment defined (as above) by the processes X_i is not a satisfying truth assignment for the formula φ.

In order to define the process P, we assume that $C_i \equiv l_{i,1} \vee l_{i,2} \vee l_{i,3}$ for $i \in \{1, \dots m\}$ and $j \in \{1, 2, 3\}$ in following formulas.

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

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

To prove completeness, we prove that Trace Validity Problem belongs to NP. We can view a process P as a labeled transition graph G_P over a set of label $\mathcal{L}$ rooted at the node n_P. For a given process P and a string $s \in \mathcal{L}^*$, we let a path p corresponding to s from n_P be the certificate. Checking can be accomplished in polynomial time by traversing G_P from n_P following p.

$\square$

The trace validity problem is NP-complete because of the non-determinism of process algebra; if a validator uses a non-deterministic process algebraic specification, the validator must keep track of all possible branchings. For example, suppose that we have the specification of process P as followings.

$$P = a.(a.(b.Nil + c.Nil) + a.(d.Nil + e.Nil))$$

Suppose that we validate a trace `a.a.f`. Then, we have to compare `f` of the trace with the all possible branches `b`, `c`, `d`, and `e`. In general, the number of possible branches is exponential in the length of the trace.

Therefore, we should be careful to define a property specification language so that trace validation against properties is tractable. The low-level specification languages and high-level specification language of the MaC architecture do not allow non-deterministic specification to make the trace validation tractable.

3.3 Abstract View of Program Execution

The behavior of target systems is reported to a monitor through a *filter*. A filter works as a sieve to decrease the amount of information to be analyzed by a monitor. A filter removes information on program activity which does not need to be analyzed concerning given properties by a monitor. A monitor receives and analyzes information reported by a filter. If a filter removes too much information, a monitor cannot evaluate properties correctly. If a filter does not filter enough information, the overhead to report the information to a monitor and to analyze the information reported to a monitor will be significant.

We view the execution of the target program as a sequence of states. First, we define a state of a program execution. A state consists of values to variables and a time stamp.

Definition 9 (State) *A state s of a program is a pair of an environment $\rho_s \subseteq V \to R$ which is a function from a set of variables V to a set of real values $\mathcal{R}$ and a time stamp $t_s \in \mathcal{R}$ such that $t_{s_i} < t_{s_{i+1}}$ for $i \geq 0$.*

In this modeling of execution, a state in the execution indicates something happens/changes at the time instant corresponding to the state. For example, suppose ρ_{s_i} has a variable x as 1 and $\rho_{s_{i+1}}$ has the variable x as 2. Then, we know that the event of updating x as 2 occurs at the time instant $t_{s_{i+1}}$. ρ_{s_i}, however, does not change between time interval starting from a state s_i until the next state s_{i+1}. In other words, the information of a program remains fixed between two states s_i and s_{i+1}. We will discuss this issue fully in Section 4.2.1.

To decrease overhead, a filter performs abstraction on the execution of the target program. We define two abstractions - *variable abstraction* and *value abstraction*. We will use S^∞ denoting $S^* \cup S^\omega$

Variable Abstraction

A program has a set of variables. Only a subset of the variables, however, may be interesting to the run-time formal analysis. We call this subset of variables *monitored variables*. $V_m \subseteq V$ denotes a set of monitored variables. A variable abstraction β_{V_m} abstracts out adjacent states which have the same restricted environment $\rho|V_m$ where $\rho|V_m = \{(x, \rho(x))|x \in V_m\}$ but only different time stamps t.

Definition 10 (Variable Abstraction) *A variable abstraction β_{V_m} with a set of monitored variable V_m is a function $\beta_{V_m} : S^\omega \to S^\infty$. β_{V_m} is defined recursively as follows.*

$$\beta_{V_m}(s_i s_{i+1} \sigma') = \begin{cases} \beta_{V_m}(s_i \sigma') & if \ \rho_{s_i}|V_m = \rho_{s_{i+1}}|V_m \\ s_i \beta_{V_m}(s_{i+1} \sigma') & if \ \rho_{s_i}|V_m \neq \rho_{s_{i+1}}|V_m \end{cases}$$

where σ' is an infinite sequence of states and $i \geq 0$

23

A filter performs variable abstraction by reporting the snapshot of the target program only when an instruction which updates *monitored variables* is executed.

Variable abstraction should distinguish monitored variables from non-monitored variables. This distinction can be made either *statically* or *dynamically*. Some instructions contain their target variables in the target program code. Thus, the distinction can be made statically. Some other instructions specify their target variables as run-time arguments. abstraction. For example, in Java bytecode, `iastore` has an argument *arrayref* and *index* to decide which element of the array is to be updated. A filter has to test whether the *index* points to the monitored variable or not at run-time.

Value Abstraction

A value abstraction $\gamma_{exp_{V_m}}$ abstracts out states which do not affect exp_{V_m}, a set of boolean expressions over the monitored variables V_m.

Definition 11 (Value Abstraction) *A value abstraction* $\gamma_{exp_{V_m}}$ *with* exp_{V_m}*, a set of boolean expressions over monitored variables* V_m *is a function* $\gamma_{exp_{V_m}} : S^\omega \to S^\infty$. $\gamma_{exp_{V_m}}$ *is defined recursively as follows.*

$$\gamma_{exp_{V_m}}(s_i s_{i+1} \sigma') = \begin{cases} \gamma_{exp_{V_m}}(s_i \sigma') & \text{if } \forall e \in exp_{V_m}. [\![e]\!]_{\rho_{s_i}} = [\![e]\!]_{\rho_{s_{i+1}}} \\ s_i \gamma_{exp_{V_m}}(s_{i+1} \sigma') & \text{if } \exists e \in exp_{V_m}. [\![e]\!]_{\rho_{s_i}} \neq [\![e]\!]_{\rho_{s_{i+1}}} \end{cases}$$

where σ' *is an infinite sequence of states,* $[\![e]\!]_{\rho_{s_i}}$ *for* $e \in exp_{V_m}$ *is the result of evaluating an boolean expression* e *using an environment* ρ_{s_i}*, and* $i \geq 0$.

A filter performs value abstraction by reporting the snapshot of the target program only when an instruction which affects at least one boolean expression in exp_{V_m} is executed. Value abstraction can abstract out more states than variable abstraction (see Figure 3.2). This is because value abstraction abstracts out adjacent states which have the same restricted environment $\rho | V_m$ as variable abstraction does, but also states which have different $\rho | V_m$ but the same evaluation result on every boolean expression $[\![e_i]\!]_\rho$ where $e_i \in exp$. Value abstraction, however, requires more computation than variable abstraction because value abstraction should evaluate each boolean expression in exp_{V_m}.

Notice that Definition 11 itself does not impose any restriction on the set exp_{V_m} except that boolean expressions in the exp_{V_m} should be expressions over the monitored variables V_m.[4] Thus, value abstraction can be valid with regard to the set of requirement properties $prop_{req}$ only if exp_{V_m} is related to $prop_{req}$.

Definition 12 (Valid Value Abstraction) *Value abstraction* $\gamma_{exp_{V_m}}$ *is valid with regard to the set of requirement properties* $prop_{req}$ *if and only if*

$$\forall j \geq 0. \left(\forall e \in exp_{V_m}. [\![e]\!]_{\rho_{s_j}} = [\![e]\!]_{\rho_{s_{j+1}}} \longrightarrow \forall p \in prop_{req}. [\![p]\!]_{\rho_{s_j}} = [\![p]\!]_{\rho_{s_{j+1}}} \right)$$

---

[4]This restriction intends to compare the abstraction power of variable abstraction and that of value abstraction easily.

The states which value abstraction abstracts out must not affect the evaluation result of requirement properties. The states which value abstraction does not abstract out, however, may or may not affect the evaluation result of requirement properties.

In one extreme end, exp_{V_m} can be a set of entire requirement properties, i.e., $exp_{V_m} = prop_{req}$. In this case, the abstract view of the target program execution is equal to the result of checking the target program execution by the monitor with regard to the requirement properties. The evaluation computation at the filter is as heavy as that at the monitor, which may cause greater overall overhead than when no abstraction is applied. In the other extreme end, exp_{V_m} can be an empty set, where the overhead of evaluating boolean expressions does not exist but the abstraction results in the empty execution. We can choose modest point between these two extreme ends by setting exp_{V_m} heuristically for decreasing the overall monitoring overheads considering communication overhead between the filter and the monitor and property evaluation cost at the monitor. Section 7.3.2 discusses one such choice of exp_{V_m} in the MaC architecture.

We illustrate these two abstractions in the example of Figure 3.2. In Figure 3.2, a state consists of two variables x and y, and time stamp t. A set of monitored variables V_m is defined as $\{y\}$. A set of boolean expressions used by value abstraction exp_{V_m} is $\{y < 5\}$.

a) Original
Execution

b) after Variable Abstraction
Vm={y}

c) after Value Abstraction
exp_{Vm}={y < 5}
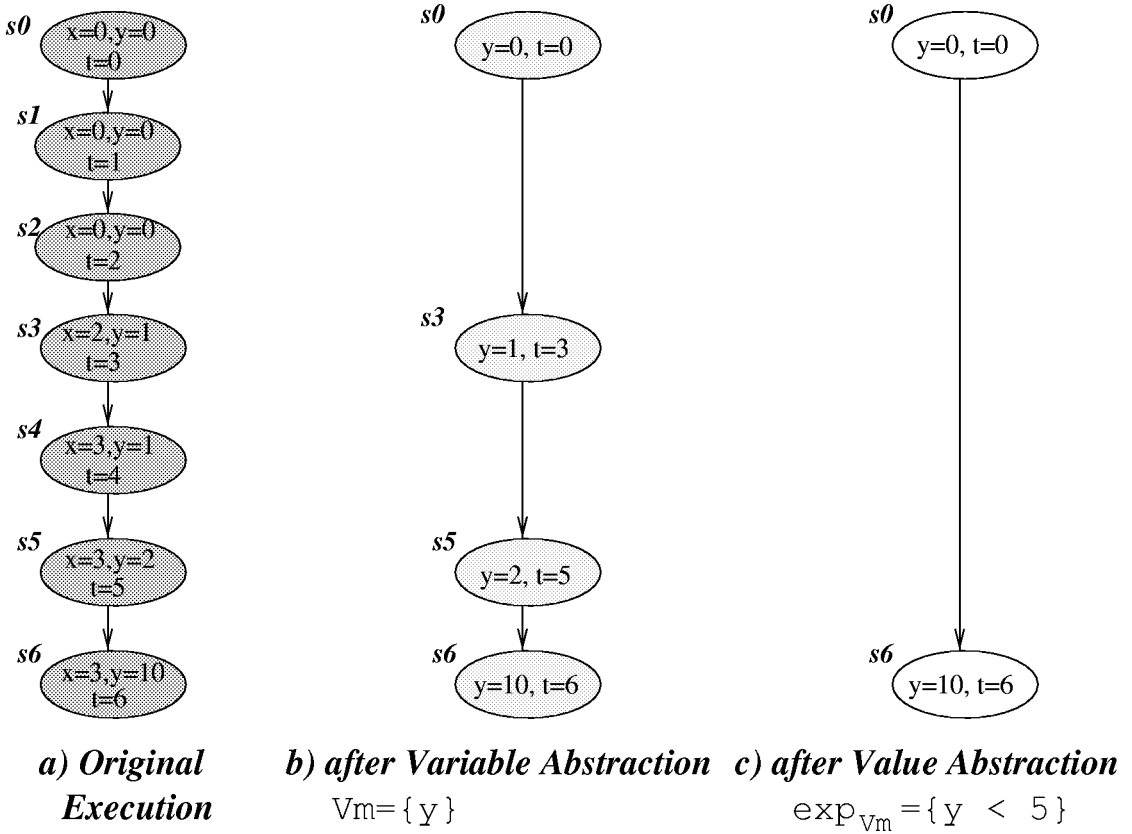
Figure 3.2: Abstract views on the execution of target program

- *Variable abstraction.* Variable x is not the monitored variable. Thus, a filter removes $s1, s2$, and $s4$ by variable abstraction. Variable abstraction abstracts out $s4$

25

additionally compared to time abstraction.

- *Value abstraction.* $s1, s2, s3, s4$, and $s5$ do not affect the evaluation of $y < 5$. Thus, a filter eliminate $s1$ to $s5$. Value abstraction abstracts out $s3$ and $s5$ additionally compared to variable abstraction.

As we see in Figure 3.2, value abstraction has stronger abstraction power than variable abstraction has. Value abstraction, however, requires more computation than variable abstraction.

Chapter 4

Overview of the Monitoring and Checking (MaC) Architecture

This chapter introduces the MaC architecture which we develop as an architecture for run-time formal analysis. Section 4.1 gives an overview of the MaC architecture. Figure 4.1 shows that a formal requirement specification consists of a low-level specification and a high-level specification. Section 4.2 presents the low-level and high-level specification languages of the MaC architecture.

4.1 The MaC Architecture

The architecture is shown in Figure 4.1. The process consists of two terms: a *static phase* and a *run-time phase*. Once a formal requirement specification is written, the run-time components of the architecture, which monitor and check the execution of the instrumented target program, are generated from the specification during the *static phase*. During the *run-time phase*, the architecture monitors and checks the execution of the target program at run-time.

4.1.1 Static Phase of the MaC architecture

A formal requirement specification is written in two separate parts: a high-level specification and a low-level specification. A high-level specification consists of requirement properties. A low-level specification contains the definitions of primitive events and conditions used in the high-level specification in terms of program entities such as program variables and program methods. We can think that a low-level specification assigns high-level meanings to the program entities. For example, when we define a primitive event OpenGate meaning that a controller starts to open a gate as an invocation of a method Control.open(), we give the meaning of start of opening the gate to Control.open(). We have two languages for describing low-level specificaions and high-level specifications. Low-level specifications are written in Primitive Event Definition Language (PEDL) (see Section 4.2.4). High-level specifications are written in Meta Event Definition Language (MEDL) (see Section 4.2.5). PEDL is dependent on the target programming language, but MEDL is not.
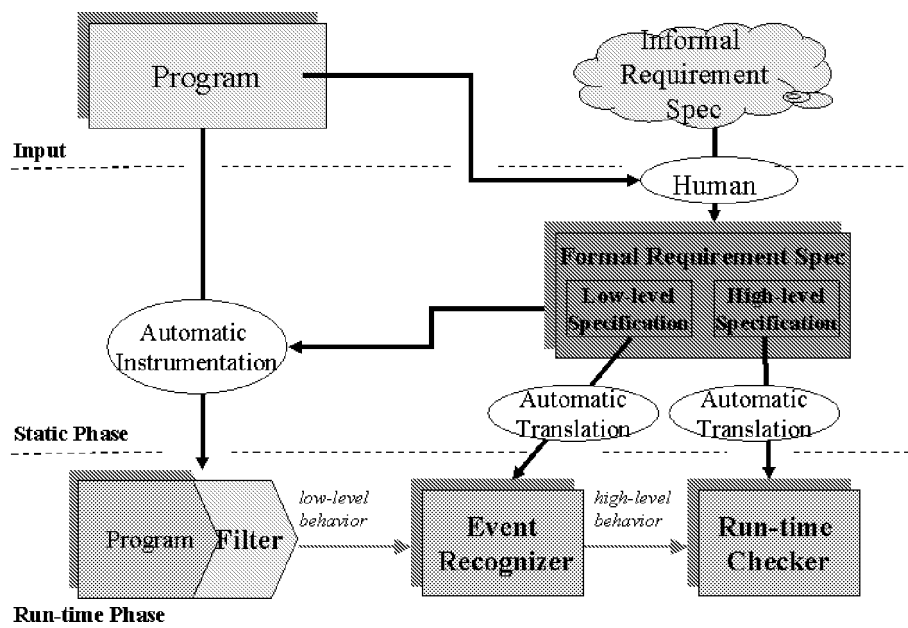
Figure 4.1: Overview of the MaC architecture

Run-time components of the MaC architecture are generated automatically from a target program and a formal requirement specification (see Figure 4.1). The MaC run-time components consist of

- an instrumented target program containing a *filter*

- a low-level behavior monitor, called *event recognizer*

- a high-level behavior checker, called *run-time checker*

A target program is instrumented according to the low-level specification written in PEDL. An event recognizer is generated from the same low-level specification. A run-time checker is generated from a high-level specification written in MEDL.

There are three benefits of separating a low-level specification and a high-level specification. First, different implementations can be monitored using the same high-level specification; only the low-level specification should be modified according to the new implementation. Second, this separation provides a clean specification of requirements by abstracting out the implementation specific details. Third, this separation allows the MaC architecture to extend to different target program languages. For example, a MaC architecture prototype for Java program may extend to analyze C++ program by changing "PEDL for Java" to "PEDL for C++" with consideration of C++ specific issues such as pointer arithmetics. An instrumentor and an event recognizer need to be modified for accepting specifications written in "PEDL for C++". MEDL and a run-time checker, however, do not need to be modified.

4.1.2 Run-time Phase of the MaC architecture

Run-time components of the MaC architecture consist of an instrumented target program containing a filter, an event recognizer, and a run-time checker. They are separated so that they can run on separated hosts. During the run-time phase, the instrumented target program reports a low-level behavior such as the updates of program variables to an event recognizer through a filter. The event recognizer receives this report of the low-level behavior and maps this low-level behavior to primitive events and conditions according to the definitions of the primitive events and conditions in the low-level specification. These primitive events and conditions constitute a high-level behavior of the target program. Then, the event recognizer reports this high-level behavior of the target program to a run-time checker. The run-time checker checks the correctness of the target program execution based on this high-level behavior.

The separation of the run-time components of the MaC architecture has following advantages. First, the modularity of the architecture makes the architecture extendable for various application areas by incorporating different tools from outside. Section 8.3 describes an example of such incorporation - incorporating the NS2 simulator [FV00] in the MaC architecture for the analysis of a network routing protocol. Second, this separation can decrease the monitoring and checking overhead to the target program by distributing the overhead to several hosts/processors. This is because we can locate the event recognizer

and the run-time checker at different hosts/processors other than the host/processor on which the target program is running.[1]

The overview of the run-time components are as follows.

- **Filter.** The essential functionality of a filter is to keep track of every change to monitored entities and send the snapshots of the target program to the event recognizer after abstracting out irrelevant snapshots. A filter consists of

  - a communication channel through which the snapshots of the target program are sent from the target program to the event recognizer

  - probes inserted into the target program which extracts the snapshots of the target program

  - a filter thread sending the extracted snapshots to an event recognizer.

- **Event recognizer.** An event recognizer detects event occurrences according to a PEDL script, which is a low-level specification, from the snapshots received from the filter. Recognized events and conditions are delivered to the run-time checker. PEDL has limited expressive power to ensure fast event recognition by the event recognizer.

- **Run-time checker.** A run-time checker checks that the current execution satisfies requirement properties in a MEDL script, which is a high-level specification, based on events and conditions received from the event recognizer.

More details on the run-time components are in Section 6.2.

4.2   The MaC Languages

Before presenting the two languages, PEDL and MEDL, we discuss some key issues in the semantics of these languages. The first issue is how to reason about temporal behavior and data behavior of the target program execution using these languages. For that purpose, the languages are designed based on instant *events* and durational *conditions*. Section 4.2.1 illustrates the distinction between events and conditions. The second issue is how the languages may handle the presence of variables that are not defined due to scoping rules. This issue is discussed in Section 4.2.2. We then formalize our intuitions on events and condition into a logic in Section 4.2.3. This logic provides the formal foundations for PEDL (in Section 4.2.4) and MEDL (in Section 4.2.5).

4.2.1   Events and Conditions

The filter reports the snapshot of the target program to the event recognizer whenever an "interesting" state change occurs in the running system. Based on the reports from the filter, the event recognizer matches the trace of the current execution against the low-level specification. In order to do this, we distinguish between two kinds of state information underlying the notifications.

---

[1]Increased communication overhead due to the distribution of the run-time components, however, should be considered.

An *event* occurs instantaneously during the system execution, whereas a *condition* is information that holds for a duration of time. Figure 4.2 shows a sequence of snapshots which are states of the target program execution and event `gateOpen` and a condition `gatePos == 2`. An event `gateOpen` denoting starting of the method `open()` occurs at the instant (a time instant 15) the control invokes the method, while a condition `gatePos ==` 2 holds as long as the variable `gatePos` does not change its value from 2, i.e., between a time instant 20 and 30. Distinction between events and conditions is very important in terms of what the event recognizer can infer about the execution based on the information it gets from the filter. For an event, the event recognizer can conclude that the event does not occur at any moment except when it receives an update from the filter. For example, the event recognizer can conclude that no event happened betwen time instant 15 and 20 or between a time instant 20 and 30. By contrast to an event, once the event recognizer receives a message from the filter that variable `gatePos` has been assigned the value 2, the event recognizer can conclude that `gatePos` retains this value until the next update. In other words, the event recognizer can conclude that gatePos == 2 holds between a time instant 20 and 30.
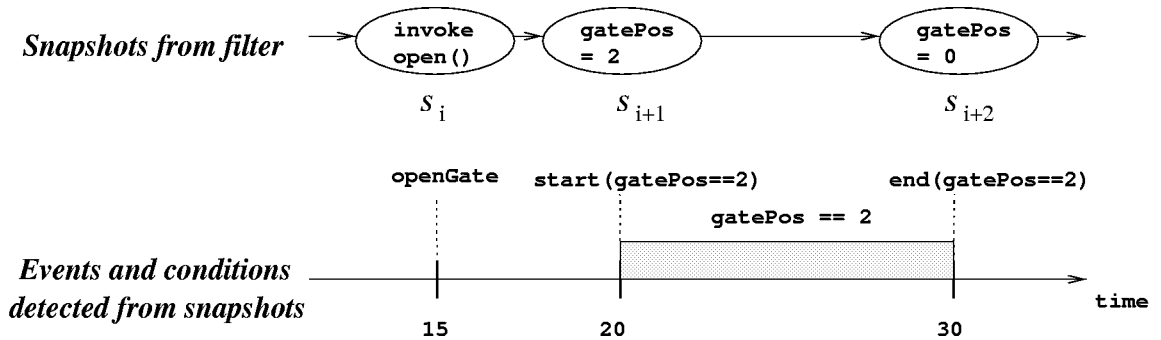
Figure 4.2: Example of an event and a condition

Since events occur instantaneously, we can assign to each event the time of its occurrence. Timestamps of events allow us to reason about timing properties of monitored systems. A condition, on the other hand, has *duration*, an interval of time when the condition is satisfied. There is a close connection between events and conditions: the start and end of a condition's interval are events, and the interval between any two events can be treated as a condition. This relationship is made precise later when we present the logic in Section 4.2.3.

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.

4.2.2 Presence of Undefined Variables

Reconsider the condition `gatePos == 2` that was used previously. When the variable `gatePos` has some integer value, it is very clear what this condition means. However, before the variable `gatePos` is initialized at the start of the execution, it is not clear whether this condition should be considered to be true or false. This problem is not just

31

confined to the start. During any execution, variables routinely become undefined when they are out of scope, and if we want to reason about such variables then we need a consistent way of interpreting logical formulae having undefined variables. The problems associated with defining the semantics of logics in the presence of partial functions[2] are well-understood [Par93]. There have been some approaches to defining logics with partial functions where the formulae are interpreted over boolean values, i.e., true and false. However, these approaches do not work when the logic has primitive relations, like "<" and "≥", which have some "natural" interpretation. Another traditional approach towards handling undefined expressions, has been to move to a three-valued logic, where the third value is taken to represent undefined. We choose to take this later approach, and so interpret the truth of conditions over a three-valued logic.

We now formalize the issues presented above, in a two-sorted logic that defines the operations on events and conditions. In this logic, we shall interpret conditions over three values and not over booleans. PEDL and MEDL are subsets of this logic with added means of definition of primitive events and conditions.

4.2.3 Logic for Events & Conditions

This section describes the syntax and semantics of events and conditions formally.

Syntax

We assume a countable set $C = \{c_1, c_2, \ldots\}$ of primitive conditions. For example, in PEDL (Section 4.2.4), these primitive conditions will be boolean expressions built from the values of the monitored variables. In MEDL (Section 4.2.5), these will be conditions that were recognized by the event recognizer and sent to the run-time checker or conditions that will be boolean expressions built from the auxiliary variables defined in the MEDL script. We also assume a countable set $\mathcal{E} = \{e_1, e_2, \ldots\}$ of primitive events. When an event occurs, it can have an attribute *value* and an attribute *timestamp*. Concrete examples of these two attributes will be given in Section 5.2.2. The logic has two sorts: conditions and events. The syntax of conditions $\langle C \rangle$ and events $\langle E \rangle$ is in Table 4.1.

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

Table 4.1: The syntax of conditions and events

---

[2]Variables can be thought of as partial functions over time

Semantics

The models for this logic are sequences of worlds, similar to those used for linear temporal logic [MP92]. 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, τ, L_C, L_E), where $S = \{s_0, s_1, \ldots\}$ a set of states, τ 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 $\{true, false, \Lambda\}$, and L_E is a partial function from $S \times \mathcal{E}$ to the value domain $\mathcal{D}_e$. Intuitively, L_C assigns to each state the truth values of all the primitive conditions; since we interpret conditions over a 3-valued logic, the truth value of primitive conditions can be $true$, $false$ or Λ (undefined). 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 τ 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.

[c_k primitive]	$\mathcal{D}_M^t(c_k) = L_C(s_i, c_k)$, where $\tau(s_i) \leq t$ and for all s_j $(j > i)$ $\tau(s_j) > t$
[defined]	$\mathcal{D}_M^t(\text{defined}(c)) = \begin{cases} true & \text{if } \mathcal{D}_M^t(c) \neq \Lambda \\ false & \text{otherwise} \end{cases}$
[pair]	$\mathcal{D}_M^t([e_1, e_2)) = \begin{cases} true & \text{if there exists } t_0 \leq t \text{ such that } M, t_0 \models e_1 \\ & \text{and for all } t_0 \leq t' \leq t, \ M, t' \not\models e_2 \\ false & \text{otherwise} \end{cases}$
[negation]	$\mathcal{D}_M^t(!c) = \begin{cases} true & \text{if } \mathcal{D}_M^t(c) = false \\ \Lambda & \text{if } \mathcal{D}_M^t(c) = \Lambda \\ false & \text{if } \mathcal{D}_M^t(c) = true \end{cases}$
[disjunction]	$\mathcal{D}_M^t(c_1 \| c_2) = \begin{cases} true & \text{if } \mathcal{D}_M^t(c_1) \text{ or } \mathcal{D}_M^t(c_2) \text{ is } true \\ false & \text{if } \mathcal{D}_M^t(c_1) = \mathcal{D}_M^t(c_2) = false \\ \Lambda & \text{otherwise} \end{cases}$
[conjunction]	$\mathcal{D}_M^t(c_1 \&\& c_2) = \mathcal{D}_M^t(!(!c_1 \| !c_2))$
[implication]	$\mathcal{D}_M^t(c_1 \Rightarrow c_2) = \mathcal{D}_M^t(!c_1 \| c_2)$

Table 4.2: Denotation for conditions

In order to define what we mean by a condition c being true in model M at time t $(M, t \models c)$, we need to define what we mean by its denotation $(\mathcal{D}_M^t(c))$. This is defined in Table 4.2. Using this we define the meaning of $M, t \models c$, and of an event e occurring in a model M at time t $(M, t \models e)$. The formal definition is given in Table 4.3[3].

As stated before, we interpret conditions over three values, $true$, $false$, and Λ (undefined). The denotation of a primitive condition, c at time t is given by c's truth value in the last state before time t. The predicate defined(c) is true whenever the condition c has a well-defined value, namely, $true$ or $false$. The denotation of 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

---

[3]Notice, that the definition of $\mathcal{D}_M^t$ refers to the definition of $\models$, and vice versa. However, the definitions are well-defined.

$M, t \models c$	iff	$\mathcal{D}_M^t(c) = true$
$M, t \models e_k$ (e_k primitive)	iff	there exists state s_i such that $\tau(s_i) = t$ and $L_E(s_i, e_k)$ is defined.
$M, t \models \text{start}(c)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models c$ and $M, \tau(s_{i-1}) \not\models c$. i.e., start($c$) occurs when condition c changes from false or undefined to true.
$M, t \models \text{end}(c)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models !c$ and $M, \tau(s_{i-1}) \not\models !c$. i.e., end($c$) occurs when condition c changes from true or undefined to false.
$M, t \models e_1 \| e_2$	iff	$M, t \models e_1$ or $M, t \models e_2$.
$M, t \models e_1 \text{ \&\& } e_2$	iff	$M, t \models e_1$ and $M, t \models e_2$.
$M, t \models e$ when c	iff	$M, t \models e$ and $M, t \models c$. i.e., event e occurs when condition c is true.

Table 4.3: Semantics of conditions and events.

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 primitive events, once again, the truth value is given by the labels on the states. Conjunction ($e_1 \& \& e_2$) and disjunction ($e_1 \| e_2$) 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)). Figure 4.2 shows an example of these two events related to the condition `gatePos==2`: an event `start(gatePos==2)` and an event `end(gatePos==2)`. 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 every condition can be identified with the events corresponding to when it becomes $true$, when it becomes $false$ and when it becomes Λ. This is the reason why the languages in the MaC architecture, are called "event definition languages".

4.2.4 Primitive Event Definition Language (PEDL)

PEDL is the language for writing the definitions of primitive events and conditions used in high-level specifications. PEDL is based on the logic for events and conditions described in Section 4.2.3. Design of PEDL is based on the following two principles. First, we encapsulate all implementation-specific details of the monitoring process in a PEDL script. 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 of the language reflects the fact that the main purpose of

a PEDL script is to define primitive events of a high-level specification.

PEDL scripts can refer to the objects of the target system. This means that declarations of monitored entities are by necessity specific to the implementation language of the system. In the current prototype for Java programs called Java-MaC (see Chapter 6), all updates of monitored variables and all invocations/returns of monitored methods are monitored. Details on PEDL for Java are in Section 5.2.

4.2.5   Meta Event Definition Language (MEDL)

The safety requirements are written in MEDL. Like PEDL, MEDL is also based on the logic for events and conditions, described in Section 4.2.3. Primitive events and conditions in MEDL scripts are imported from PEDL scripts; hence the language has the adjective "meta".

The overall structure of a MEDL script is given in Figure 4.3. `<e>` is a name of an event. `<c>` is a name of a condition. `<aux_v>` is an auxiliary variable.

A MEDL script consists of five sections:

- *import section* declares a list of events and conditions to be imported from an event recognizer.

- *auxiliary variable declaration section* declares a list of auxiliary variables.

- *event and condition definition section* defines events and conditions based on the imported events and conditions and auxiliary variables.

- *property and violation definition section* defines safety properties and violations.

- *auxiliary variable update section* defines rules how and when the auxiliary variable is updated

Auxiliary variables.   The logic described in Section 4.2.3 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 the user 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, `OpenGate -> t' := time (OpenGate)` records the time of occurrence of event `OpenGate` in the auxiliary variable `t`. Expression `e1 -> count_e1' := count_e1 + 1` counts occurrences of event `e1`. A special auxiliary variable `currentTime` can be used to refer to the current time of the system. Precisely, it is set to be the last timestamp received from the filter.

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 by boolean expressions using the auxiliary variables. More complex events and conditions are then built up using the various connectives described in Section 4.2.3. These events and conditions are then used to define the safety properties and alarms.

35

```
ReqSpec <...>                   // the title of a MEDL script

  /* Import section */
  import event <e>;             // imported event declarations
  ...
  import condition <c>;         // imported condition declarations
  ...

  /* Auxiliary variable declaration section */
  var int <aux_v>;              // auxiliary variable declarations
  ...

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

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

  /* Auxiliary variable update section */
  <e> -> {  <aux_v'> := ... ; }  // auxiliary variable updation
  ...

End
```

Figure 4.3: Structure of MEDL

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 the same thing. The reason we have both of them is because some properties are easier to think of in terms of conditions, while others are easier to think of in terms of alarms.

Section 4.2.6 will give more detail on features of MEDL through a railroad example. A complete BNF syntax of MEDL is given in Appendix D.

4.2.6 Example

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

The following code shows a fragment of the gate controller implemented as a Java class. The state of the gate is represented as variable `gatePos`, which can assume constant values `GATE_UP`, `GATE_DOWN`, or `IN_TRANSIT`. The controller controls the gate by means of methods `open()` and `close()`. For simplicity, we assume that there is only one instance of class `Control` in the system.

```
class Control {
    public static final int GATE_UP    = 0;
    public static final int GATE_DOWN  = 1;
    public static final int IN_TRANSIT = 2;
    int gatePos;
    public void open()  { ... }
    public void close() { ... }
      ...
};
```

In this example, we monitor the controller of the gate. A safety requirement concerning the gate is that the gate is down within 30 seconds after signal *CloseGate* is sent, unless signal *OpenGate* is sent before the time elapses. Precisely, we check that if there is a signal *CloseGate*, not followed by either signal *OpenGate* or completion of gate closing, is present in the execution trace, then the time elapsed since that signal is less than 30.

Figure 4.4 shows PEDL script for the railroad crossing. The PEDL script introduces high-level events `OpenGate`, `CloseGate` and `Gate_Down` in lines 3 and 4. These events and condition are defined from lines 11, 12, and 14 using program variables and program

```
01:MonScr RailroadCrossing
02:
03:  export event OpenGate, CloseGate;
04:  export condition Gate_Down;
05:
06:  monmeth void Control.open();
07:  monmeth void Control.close();
08:  monobj  int Control.gatePos;
09:  monobj  int Control.GATE_DOWN;
10:
11:  event OpenGate = startM(Control.open());
12:  event CloseGate= startM(Control.close());
13:
14:  condition Gate_Down =(Control.gatePos
15:                     == Control.GATE_DOWN);
16:End
```

Figure 4.4: PEDL script for the gate controller

method in lines 6 to 9. The MEDL script in Figure 4.5 uses the events and conditions imported from the PEDL script in lines 3 to 4. Lines 9 to 12 specify the safety requirement. The time of the last occurrence of event CloseGate is recorded by the auxiliary variable lastClose in lines 14 to 16.

```
01:ReqSpec SafeCrossing
02:
03:   import event OpenGate, CloseGate;
04:   import condition Gate_Down;
05:
06:   var float lastClose;
07:   var float currentTime;
08:
09:   property GateClosing =
10:      [ CloseGate when !Gate_Down,
11:        OpenGate || start(Gate_Down)
12:      ) => lastClose+ 30*1000 >currentTime;
13:
14:   CloseGate -> {
15:     lastClose' = time(CloseGate);
16:   }
17:End
```

Figure 4.5: MEDL script for the gate controller

Chapter 5

Monitoring Java Programs

This chapter describes issues on monitoring Java programs. First, we discuss the object-orientation of the Java programming language in Section 5.1.1. We describe the problems we confront when we specify and monitor variables in a complex object graph which changes dynamically. Then, we propose a solution to the problems with restrictions in Section 5.1.2. Second, we describe "PEDL for Java" in which low-level specifications for Java programs are written. We will use uppercase letters from the beginning of the alphabet for indicating classes and lowercase letters for indicating objects in this chapter.

5.1 Monitoring Objects

5.1.1 Object Orientation in Java

A Java program is an evolving collection of objects. Java handles an object via *references* pointing to the object. Many references can point to the same object. An object contains variables of primitive types such as `int` and `double`, and variables of reference type. A member variable of an object is accessed through a reference pointing to the object.

It is non-trivial to specify and monitor an object in a complex object graph. Suppose we want to specify and monitor the variable x inside the object pointed by `a.b2`. The variable x is pointed by an arrow in Fig 5.1. First, we specify x's location (parent object) in the object graph such as `a.b2` to distinguish this x from x in another object such as an object pointed by `a.b1`. Second, we need to monitor updates of references which possibly point to the parent object of x. A monitored variable can be updated through several alias references pointing to the parent object of the variable. Thus, references which possibly point to the parent object needs to be monitored at run-time to see whether they are actually pointing to the parent object. We have to test all references of type B such as `a.b1`, `a.b1.b'`, `a.b2`, and `a.b2.b'` whenever these references are updated because the x can be updated through these references.

These reference tests are problematic. A reference to the parent object may not be visible to locations where other references of the same type are updated due to Java scoping rules.[1] Table 5.1 shows member visibility according to different visibility modifiers in Java [Fla99].

---

[1] In addition, local variables are not visible outside of the method which declares the variables.
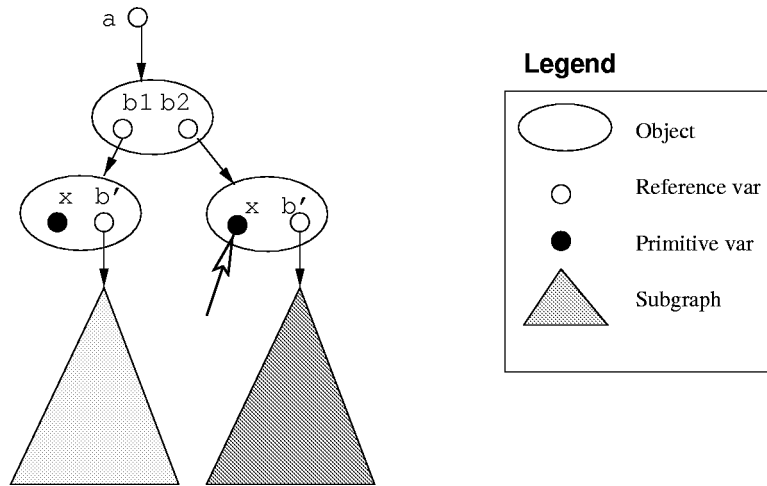
Figure 5.1: An object graph

Accessible to:	public	protected	package	private
Same class	yes	yes	yes	yes
Class in same package	yes	yes	yes	no
Subclass in different package	yes	yes	no	no
Non-subclass, different package	yes	no	no	no

Table 5.1: Class member accessibility

For example, suppose that b2 is declared as private in the class A. Then, we cannot test whether a.b1 is equal to a.b2 outside of the class A. Therefore, we need a globally accessible table which contains the addresses of objects containing monitored variables. We call this table *address table*. We will describe the detail of the address table in Section 5.1.2.

The cost of maintaining the address table, however, can be expensive. Whenever a reference pointing to the object containing the monitored variable changes, the address table need to be updated accordingly. One reference change such as a.b2 = a.b1 may cause the substitution of all a.b2's descendent nodes in the address table. For example, suppose a.b2.b' has 10 monitored objects a.b2.b'.c1,..., a.b2.b'.c10 in its subgraph. After a.b2 = a.b1, all addresses of a.b2.b'.c1,..., a.b2.b'.c10 in the address table should be replaced by the addresses of a.b1.b'.c1,..., a.b1.b'.c10

Figure 5.2 illustrates these problems concretely. Suppose that we would like to monitor a1.b1.x and a1.b2.x. in A.main().

```
01:class A {
02:    B b1, b2;
03:    A(int x, int y) { b1 = new B(x); b2 = new B(y);}
04:    public static void main(String[] args) {
05:        A tmp = null;
06:        A a1 = new A(1,1);
07:        A a3 = a1;
08:        A a2 = new A(2,2);
09:
10:        a3.b1.x = 10;
11:        updateAs10(a1);
12:        tmp = a1;              // swap a1 and a2
13:        a1 = a2;
14:        a2 = tmp;
15:        updateAs10(a2);
16:    }
17:    static void updateAs10(A a) {
18:        a.b1.x = 10; a.b2.x = 10;
19:    }
20:}
21:class B {
22:    int x;
23:    B(int x) { this.x = x;}
24:}
```

Figure 5.2: Example showing aliasing and reference changing

On line 6, an object of type A which has two objects of type B containing x as 1 is created and the local reference variable a1 points to the object. Two aliases to the object are created in this program.

- on line 7, the object is pointed by a reference a3.

42

- on line 12, the object is pointed by a reference `tmp`.

These aliases are shown in Figure 5.3.a). On line 10, we have to test whether a reference

after line 12   after line 8   after line 14

a) Aliasing   *b) Reference Changing*
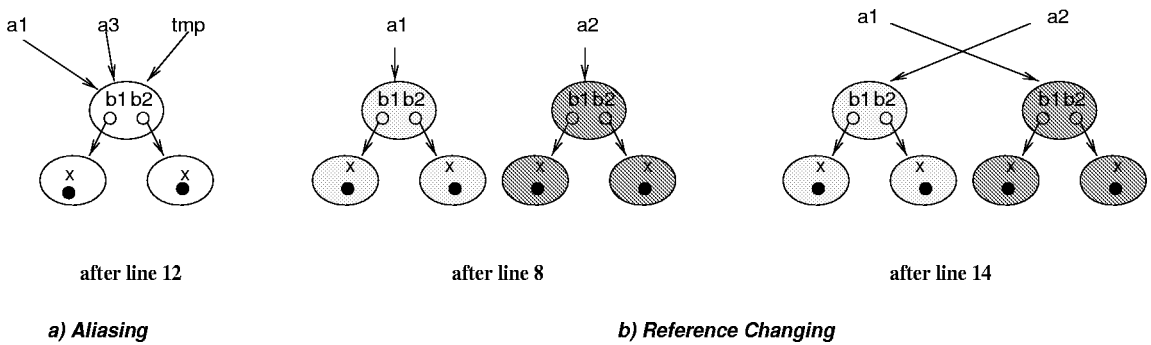
Figure 5.3: Aliasing and reference changing

`a3.b1` is equal to a `a1.b1` or `a1.b2`. In other words, we have to test whether `a3.b1` points
to the same object pointed by `a1.b1` or `a1.b2`. We know that `a3.b1` is equal to `a1.b1`
because `a3` is equal to `a1` in line 10 because line 7 assigns `a1` to `a3`. Similarly, on line 18, we
have to test whether a formal parameter `a` of `updateAs10()` is equal to `a1` or not. If the
parameter `a` is equal to `a1`, `updateAs10()` updates the monitored variables. Otherwise,
not.[2] For example, on line 11, `updateAs10()` updates the monitored variables because its
actual parameter `a` is equal to `a1`. On line 15, however, `updateAs10()` does not update
the monitored variables because its actual parameter `a` is not equal to `a1` but `a2`.[3] On line
13, `a1` is assigned with `a2`. Accordingly, the address table should update its entry for all
subnodes of `a1` such as `a1.b1` and `a1.b2` with the address of `a2.b1` and `a2.b2`.

The elimination of pointer and strong typing in Java, however, lessen the degree of
aliasing compared to other conventional language such as C/C++. Java does *not* provide
pointers to variables. Therefore, a direct alias to a variable cannot be made while an
indirect alias to a variable can be still made through a reference to the parent object of the
variable. Arithmetic on references, however, is prohibited. In addition, Java is strongly
typed. Java has orthogonal sets of similar instructions working on different data types such
as `istore,lstore,fstore` and `dstore` for `int,long,float`, and `double`, respectively. A
reference variable of type T cannot be assigned with a value of numeric type such as
`int` or a value of different type T' unless T' is a subclass of T. These features narrow
down candidate update instructions to be monitored, which lessens the difficulty caused
by aliasing and reference changing.

The benefit of these features for monitoring can be illustrated by showing the difficulties
of monitoring programs in C which does not have these features. Indirect access to variables
through pointer makes detecting updates of the monitored variables difficult. Suppose an
integer variable `x` in `main()` is monitored in Figure 5.4. `x` can be updated directly by
variable `x` or indirectly by pointer `px`. `x` can be updated, however, through a variable
`px2` of type `int` at line 9 and a variable `px3` of type `long` at line 10, too. In order to

---

[2]If we do not have the address table, we cannot perform this test because `a1` is not visible on line 18.

[3]We monitor the variables x of the objects currently pointed by the *references* `a1.b1` and `a1.b2` regardless
of the values of `a1.b1` and `a1,b2`.

```
01:void main() {
02:    int x = 3;
03:    int *px = &x;
04:    int px2 = px;
05:    long px3 = px;
06:
07:    x ++;                // x == 4
08:    (*px)++;             // x == 5
09:    (*(int *)px2)++;     // x == 6
10:    (*(int *)px3)++;     // x == 7
11:
12:    g(px);               // x == ?
13:    *((int *)rand())=-1;// x == ?
14:}
```

Figure 5.4: Example showing indirect access in C

monitor x, we have to keep watching all variables which can be converted to pointer type. Furthermore, a pointer to x is passed to another function g(). Although x is a local variable declared inside function main(), we have to keep track of the execution of another function g(), too. The worst scenario is demonstrated in line number 13. Depending on the return value of rand() which ranges over integer, line 13 may update x. As we have seen, there are numerous locations which possibly update x. Large degree of aliasing results in high overhead at run-time because we need to frequently test whether an instruction updates a monitored variable or not. Figure 5.4 clearly shows the difficulty caused by pointer arithmetic and weak typing. In contrast to C, Java disallows such indirect accesses to variables, which helps to make monitoring a Java program relatively easier than a C program.

5.1.2 The Address Table

This section describes a solution of using *address table* to the problems of aliasing and reference changing when we monitor objects of a Java program. In Figure 5.5, main(String[]) creates a1 and a2 which are instances of class A. a1 and a2 each have b1 which is of class B. Similarly a1.b1 and a2.b1 each have a c1 and a c2 of class C.

We can distinguish a1.b1.c1 and a1.b1.c2 by comparing its address at run-time. Since an object is located at an unique address in the heap, comparing the addresses of these two objects allows us to distinguish one object from another. The *address table* contains pairs of name and address of monitored objects. Suppose that we want to monitor a1.b1.c1 where a1.b1.c1 is located at heap address 8200 as Figure 5.6. The address table contains pairs of addresses and monitored objects' names, such as (8200, a1.b1.c1).[4] At run-time, we check whether an object of C has address 8200 or not. If the address of the

---

[4]In general, an address can match more than one monitored object's name due to aliasing. We will assume, however, that an address can match only one monitored object's name. We will later see that the validity of this assumption follows from the fact that we do not allow reference assignments.

```
01:class A{
02:    B b1;
03:    A(int x) {
04:        b1 = new B(x);
05:    }
06:    public static void main(String[] args) {
07:        A a1 = new A(10);    // instance of A created at address 8000
08:        A a2 = new A(20);    // instance of A created at address 9000
09:    }
10:}
11:class B{
12:    int x;
13:    private C c1;
14:    private C c2;
15:    B(int x) {
16:        c1 = new C(x);
17:        c2 = new C(x);
18:    }
19:}
20:class C {
21:    int x;
22:    C(int x) { this.x = x;}
23:}
```

Figure 5.5: Example Java code for monitoring objects
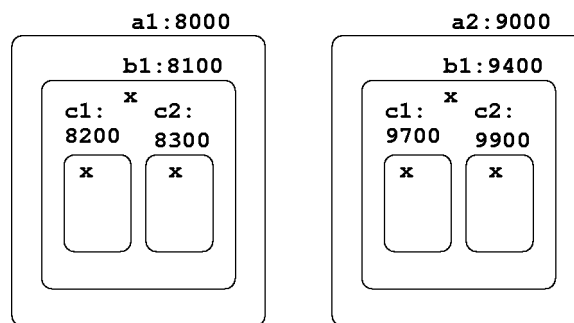
Figure 5.6: Objects created in the program of Figure 5.5

object is 8200, it is `a1.b1.c1` and is reported to an event recognizer.[5] Otherwise, it is not reported.

Maintaining the Address Table

The address table should be updated whenever a reference to a monitored object is updated. One difficulty in maintaining the address table is that object graphs can change arbitrarily. Figure 5.7 (different representation of Figure 5.6) illustrates how to maintain the address table when an object graph has changed.

Address Table

Variable Name	Addr	Desc Ptrs
a1	8000	
a1.b1	8100	
a1.b1.c1	8200	
a1.b1.c2	8300	

`a1.b1 = a2.b1;`

Address Table

Variable Name	Addr	Desc Ptrs
a1	8000	
a1.b1	9400	
a1.b1.c1	9700	
a1.b1.c2	9900	
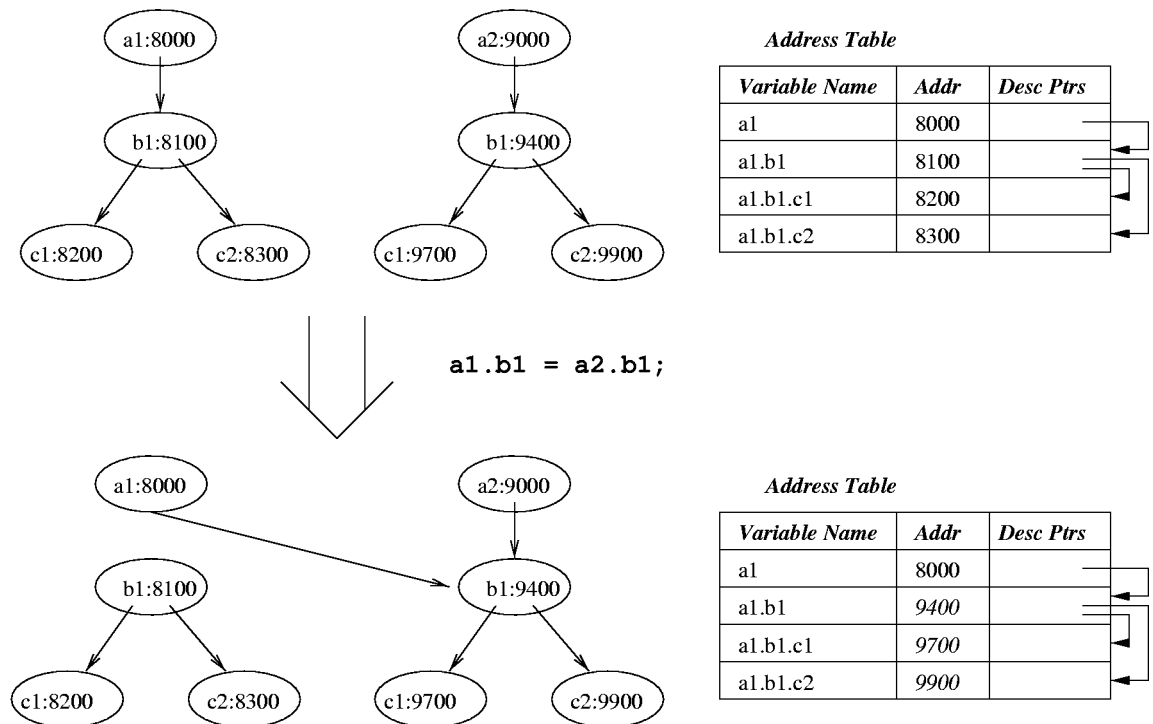
Figure 5.7: Update of address table

Suppose that `a1.b1.c1` and `a1.b1.c2` are monitored. Also suppose that the address table contains monitored objects and their ancestors. In addition, each entry in the address table has pointers to its descendants. We can detect an assignment of a monitored object's ancestor such as `a1.b1` = `a2.b1` by finding the object assignment instruction, such as `astore` and testing whether the instruction assigns an ancestor of a monitored object. Once we detect that an ancestor is assigned, we should update the address table with the addresses of the newly assigned objects. Figure 5.7 shows changes in the address table caused by ancestor assignment `a1.b1` = `a2.b1`. Once we detect `a1.b1` is assigned with a new value `a2.b1`, we find the descendants of `a1.b1` such as `a1.b1.c1` and `a1.b1,c2` by following the pointers to the descendant. Then updates the addresses of these descendants in the address table with new addresses which are the addresses of `a2.b1.c1` and `a2.b1.c2`. We, however, may not be able to get these new addresses directly due to the Java scoping

---

[5]The reference to a monitored object contains the address of the object.

rule as we have seen in Section 5.1.1. For example, we cannot get the addresses of `a2.b1.c1` and `a2.b1.c2` from outside of class B because `c1` and `c2` are declared as *private*. There are two ways to get these new addresses.

1. Traverse an object graph by *reflection* mechanism. Reflection mechanism allows dynamic traversal of object graphs ignoring of scoping rules only if it is allowed by a security manager of the Java program.[6] If a reflection mechanism is allowed, we can traverse `a2.b1`, `a2.b1.c1`, and `a2.b1.c2` and get the addresses for these three objects.

2. Maintain an *extra* address table which contains all pairs of addresses and names of object graphs which are assignable to any ancestor of the monitored object. We call this table *extra* because this table does not have addresses of monitored objects but addresses of un-monitored objects. Whenever the assignment to the un-monitored objects occurs, this extra address table need to be updated in similar way of updating the address table to keep the extra address table consistent.

Unfortunately, both methods pose unacceptable performance penalties at run-time. First, all ancestors of the monitored object must be watched to detect if they are assigned with other objects. Second, a subgraph can be arbitrarily large; it can be an object graph of the entire program. The cost of updating the address table is proportional to the size of newly assigned subgraph. There are additional problems specific to each method. For dynamic traversal by reflection mechanism, reflection may not be allowed by a Java security manager. For the approach using an extra address table, the extra address table can take memory space as large as the original space of the program.

Because of these difficulties in updating the address table, Java-MaC put one assumption into its target program.

> The ancestors of the monitored object does not change so that the address table need not be updated.

This assumption seems very restrictive at the first glance. This assumption is, however, reasonable one considering the following fact. When we specify a monitored object in a PEDL script, we describe it with the name of reference such as `a.b.c` pointing to the object. If an ancestor of the monitored object changes by an assignment to the ancestor, the object we intended to monitor is not being monitored anymore. In many cases we already have this assumption that the object pointed by the reference we specify would not change with another object. Therefore, we believe that the assumption is natural and not so prohibitive in most cases.

There are two garbage collection related issues, however, concerning the maintenance of the address table.

1. An object may be relocated and therefore the address of the object can be changed.

2. Created objects may not be reclaimed because the address table contains a reference to the object all the time.

---

[6]A reflection mechanism [REF97] is supported in Java 1.1 or later.

The first issue does not cause a problem. A reference to a monitored object stored in the address table is automatically updated to a new address by a garbage collector when the garbage collector relocates the monitored object to the new address. Concerning the second issue, a reference to a monitored object contained in the address table needs to be distinguished from normal references used in the target program so that when the object is not reachable except from a reference inside the address table, the object should be reclaimed and the address table should remove corresponding entries for the object. *Weak references* [REF99] provided by the Java 2 platform satisfies this requirement. A weak reference is a wrapper class for a normal reference. A weak reference is the same as a normal reference except that it does not prevent a garbage collector from reclaiming an object it points to. In addition, a user can implement user's own notification method and enroll this method to a weak reference so that the method is invoked when the object pointed by the weak reference is reclaimed. Using this notification mechanism, Thus, Java-MaC can know when the monitored object is reclaimed and set values for the object as undefined.

Creating the Address Table

The assumption that the ancestors of the monitored object does not change implies that an object graph containing the monitored object should be created in a *top-down* way.[7] For example, suppose a linked list containing three nodes is created at line 13 in a top-down way in Figure 5.8.

```
01:class Node {
02:     int value;
03:     Node(int[] values) {
04:         if (values.size == 1) next = null;
05:         else next = new Node( tail(values))
06:         value = head(values);
07:     }
08:     void insert(Node n) {
09:         if(next == null) next = n;
10:         else next.insert(n);
11:     }
12:     public static void main(String[] args) {
13:     Node head = Node({1,2,3});
14:     head.insert(new Node({4}));
15:     }
16:     ...
17:}
```

Figure 5.8: Linked list containing three nodes

---

[7]"top-down" means that child objects should be created directly from the constructor of their parent object.

Figure 5.9 shows the process of creating the object graph rooted at **head** at line 13. An object indicated by a dotted circle means that a constructor for the object is invoked but not yet finished. An object of solid circle indicates that a constructor for the object returns, i.e., the object is completely created.
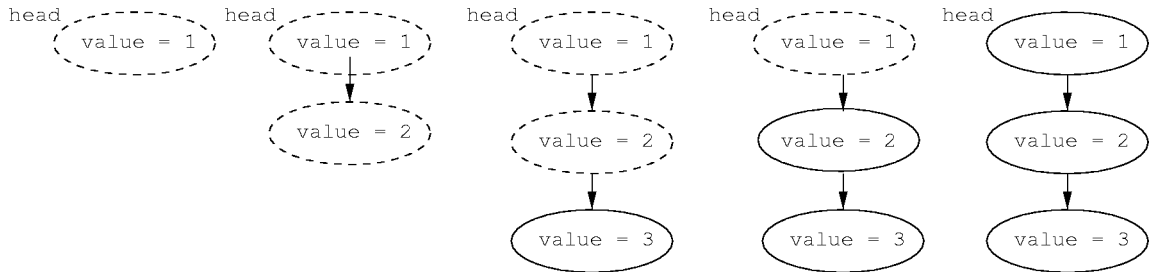
Figure 5.9: Top-down creation of an object graph

head, **head.next**, and **head.next.next** are created in a top-down fashion. Thus, Java-MaC can monitor these objects and check whether the linked list is sorted or not by comparing these three objects. Java-MaC, however, cannot monitor the newly inserted node in line 14, because it is not created in top-down fashion.

When an object is created in top-down way, we can discover the complete name of the object by looking at the method call stack containing constructors. Java does not, however, provide direct access to the call stack. Thus, we create our own constructor stack per thread by inserting probes before and after each constructor invocation. Figure 5.10 illustrates the process of creating an object graph in top-down way.
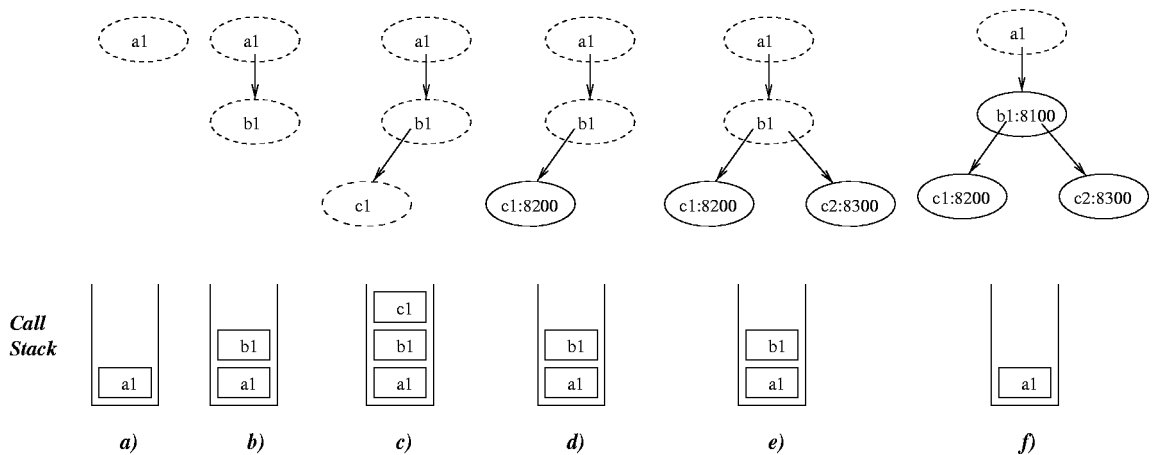
Figure 5.10: Process of creating object graph

In Figure 5.10 a dotted circle means that an object is not yet completely created.

1. Figure 5.10.a) shows the object graph state of line 7 of Figure 5.5. A constructor of class **A** has been invoked, but not yet returned.

2. b) displays the state at the beginning of line 4.

49

3. c) provides a snapshot at the beginning of line 16.

4. d) provides a snapshot at the end of line 16.

5. e) represents the end state of line 17.

6. f) occurs at the end of line 4.

We use two methods to manipulate constructor stack: `pushObjName(String objName)` and `popObjName()`. `pushObjName(String objName)` pushes the name of object currently being created into the stack. `popObjName()` pops up the name of an object completely created from the stack. For example, in Figure 5.5, `pushObjName("a1")` should be called just before line 7, and `popObjName()` should be called right after line 7. Similarly, `pushObjName("b1")` should be called before line 4, and `pushObjName()` should be called directly after line 4. When line 16 begins ((c) in Figure 5.10), the call stack contains `a1.b1.c1`. We know that `a1.b1.c1` is a monitored object. Thus, we can add a pair of (`8200,a1.b1.c1`) to the address table.

5.2 PEDL for Java

PEDL must be closely related to the target programming language, because definitions of primitive events are based on entities of programming language such as variables and methods. In this section, we will describe PEDL for Java. We will simply use PEDL as PEDL for Java in this section. The overall structure of a PEDL script is given in Figure 5.11. `<e>` is a name of an event. `<c>` is a name of a condition. `<var>` is a variable in a target Java program. `<meth>` is a method in a target Java program. A PEDL script consists of four sections.

- *export section* declares a list of events and conditions to be exported to a runtime checker. Notice that not all events/conditions are exported. A user can define events/conditions which are not exported. A user uses these unexported events/conditions as subexpressions of exported events/conditions.

- *overhead reduction section* provides the following four flags to turn on and turn off to reduce the monitoring overhead:

  - `timestamp <period_in_ms>`: sending timestamps from a filter to an event recognizer in every `<period_in_ms>` millisecond so that event recognizer can evaluate timing properties

  - `multithread`: executing lock/unlock operations to make an update and a probe atomic so that race condition in multi-threaded program can be avoided (see Section 6.2.1)

  - `deltavalue`: sending the value of a monitored variable in compact representation to decrease the size of a snapshot

  - `valueabstract`: sending a snapshot of the target program only when that snapshot affects the evaluation of properties (see Section 7.3.2)

50

```
MonScr <...>                    // the title of a PEDL script

   /* Export section */
   export event <e>;        // exported event declarations
   export condition <c>;    // exported condition declarations

   /* Overhead reduction section */
   [timestamp <period_in_ms>;]
   [multithread;]
   [deltavalue;]
   [valueabstract;]

   /* Monitored entity declaration section */
   monobj <var>;            // declaration of monitored variables
   ...
   monmeth <meth>;          // declaration of monitored methods
   ...


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

End
```

Figure 5.11: Structure of PEDL

When a flag is present, the corresponding option is enabled. Otherwise, the option is disabled. The default values for these flags are *false*. The overhead is reduced either when `timestamp` or `multithread` is set false, or when `deltavalue` or `valueabstract` is set true.

Chapter 7 discusses these options in detail.

- *monitored entity declaration section* declares what variables and methods in the target program are monitored. Section 5.2.1 describes the details of these declarations.

- *event and condition definition section* defines events and conditions based on primitive events and primitive conditions constructed from the declared monitored variables and methods. Section 5.2.2 describes the details of these definitions.

A complete BNF syntax of PEDL is in Appendix C.

51

5.2.1 Declared Monitored Entities

Declared Monitored Variables

Since Java is an object oriented language, PEDL for Java needs to define events and conditions based on objects such as `a1.b1.c1` where `a1` is a parent object of `b1` and `b1` is a parent object of `c1`. A prefix of the monitored object name should be that of *root object*. A *root object* is an object created statically. A root object is either[8]

- a static object.

  - For example, `A.b1` is the root object of `A.b1.c1` where `A` is a class name and `b1` is declared as a static object in `A`.

- a local object declared inside of a static method

  - For example, `A.main(String[]).c1` is the root object of `A.main(String[]).c1.d1` where `A` is a class name, `main(String[])` is declared as a static method in `A` and `c1` is declared as a local object in `main(String[])`.

PEDL, however, does not allow objects to be monitored. Rather, PEDL declares member/local *primitive* variables of objects to monitor.[9] The rationale for monitoring only primitive variables is the following. Suppose we monitor an object including an object graph rooted at that object. Java-MaC must always watch whether any field of the object graph is changed. In some case, an object graph can be arbitrarily large even to the size of the entire object graph of the target program. Also, when Java-MaC detects the object has changed (i.e., some field of the object graph rooted at object has changed), the entire object graph should be delivered to the event recognizer. Both of these behaviors pose an unacceptable performance penalty. Therefore, it is for efficiency that PEDL allows Java-MaC to monitor only primitive variables.

PEDL can define an event or a condition related to contents of an object represented by primitive variables. PEDL, however, cannot define events or conditions based on information specific to Java objects such as types of objects. For example, Java provides `instanceof` keyword to determine if an object is of a given type. Suppose `A a = new A();`. Then, `a instanceof A` is true. PEDL cannot define a condition such as this because PEDL does not handle an object directly but only primitive variables of the object.

Although PEDL does not handle object specific information, we believe that handling primitive variables constituting an object is expressive enough to define events and conditions for many purposes considering what really defines an object is its primitive variables. Henceforth, whenever we say monitoring an object, it means monitoring primitive variables of the object.

---

[8]PEDL provides another way to declare monitored objects. A user can use a class name to indicate all instances of the class. For example, for the declaration `monobj A.b1.c1`, `A` indicates all instances of class `A` if `A.b1` is not a root object, i.e., `b1` is not a static variable of `A`. This way of declaration is convenient when a class creates only one instance or all instances of the class are supposed to be monitored collectively. See Section 5.1.2.

[9]A *local variable* is a variable declared inside of a method body. Primitive types in Java are `boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, and `double`.

Declared Monitored Methods

The finest level of execution points in Java programs is bytecode instruction. We may specify an execution point such as `Address 24 in Foo.bar()` which means that we would like to detect when the target program executes the bytecode instruction located at the address 24 of the method `Foo.bar()`. It is not technically difficult to detect an execution point specified as the *address* of bytecode instruction; we can insert a probe at the address of any bytecode instruction and detect the execution of that instruction. A user may want to indicate monitored execution points in bytecode addresses because the user can express *any* execution points of the target program in this way. The instruction level execution point, however, is inconvenient to use because it is difficult for a user to know what addresses to monitor in order to check whether the execution of the target program is correct. It is difficult for two reasons. First, a user needs to inspect the bytecode, which is hard to understand because of its low-level, to specify the address to monitor. Second, the address of an instruction does not have inherent meaning in the program. Given the same source code, the bytecode generated by a different compiler may have different sequences of bytecodes.

Therefore, PEDL does not provide a way to specify execution points by directly specifying the *addresses* of instructions. Rather, PEDL makes a user specify an execution control instruction whose target address is a monitored execution point. There are three types of execution control instructions in Java -*jump, exception,* and *method beginning/ending.*

- *jump*
  An unconditional *jump* instruction has a target address in a method as the only operand. A conditional *jump* instruction has several target addresses. The instruction selects the next move depending on its condition value.

- *exception*
  When an exception occurs, an *exception handler* handles the exception. In other words, when an exception occurs, the control moves to the beginning of the corresponding exception handler. An exception handler is declared by a *range* indicated by a pair of beginning/ending addresses in a method and *a type of exception* which might happen inside the range.

- *method beginning/ending*
  Methods are considered as basic building blocks for a program. Thus, important execution points are frequently specified using method beginning/ending.

To specify jumps and exceptions to be monitored, a user has to look at the details of the code inside a method and specify which jumps/exceptions in the method to monitor. This is a difficult task because a user needs to understand the code inside a method. In contrast, to specifying a jump or exception, specifying a method beginning/ending to be monitored is easy because a user does not need to know the code inside method. For this reason and the popular usage of methods, currently, PEDL supports *method invocation/return.*

5.2.2 Defining Events and Conditions

Basic building blocks of events and conditions in PEDL script are primitive variables and methods declared as monitored entities as we have seen in Section 5.2.1.

Defining conditions. Primitive conditions in PEDL, are constructed from boolean-valued expressions over the monitored variables. An example of such condition is `condition TooFast = Train.calculatePosition().trainSpeed>100`. In addition to these constructed boolean expressions, we have the primitive condition `InM(f)`. This condition is true as long as the execution is currently within method `f`. Complex conditions are built from primitive conditions using boolean connectives.

Defining Events. The primitive events in PEDL correspond to updates of monitored variables and invocations and returns of monitored methods. Each event has an associated timestamp and may have a tuple of values such as $(V_1, V_2, ..., V_n)$.

The event `update(x)` is triggered when variable x is assigned a value. The value associated with this event is the new value of `x`. Events `startM(f)` (`endM(f)`) are triggered when control enters method `f` (respectively, returns from `f`). For example, `event OpenGate = startM(Control.open())` defines an event meaning a controller starts opening a gate in Figure 4.4. The value associated with `startM` is a 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 formal parameters at the time control returns from the method.

All operations on events defined in the logic can be used to construct more complex events from these primitive events. In PEDL, we also have two attributes defined for events, `time` and `value`. As mentioned in Section 4.2.3, events have associated with them attribute values, and the time of their occurrence. These can be accessed using the attributes `time` and `value`.

Time. `time(e)` gives the time of the last occurrence of event `e`. `time(e)` refers to the time on the clock of the monitored system (which may be different from the clock of the monitor) when this event occured.

Value. `value(e,i)` gives the ith value in the tuple of values associated with `e`, provided `e` occurs.[10] We define values of primitive events as follows:

- A value of `update(var)` is a tuple containing the current value of `var`. For example, suppose that `int Foo.x` is a monitored variable and the target program assigns 10 to `Foo.x`. Then an event `update(Foo.x)` occurs. The value of the event `value(update(Foo.x),0)` is 10.

- A value of `startM(method)` is a tuple containing parameters of the `method` when the `method` starts. For example, suppose that `int Foo.sum(int,int)` is a monitored method. When the target program invokes `Foo.sum(int,int)` with a pair of parameters 1 and 2, the event `startM(Foo.sum(int,int))` occurs. The event has two values because `Foo.sum(int,int)` has two parameters.

  - `value(startM(Foo.sum(int,int)),0) = 1`
  - `value(startM(Foo.sum(int,int)),1) = 2`

---

[10] i starts from 0.

54

- A value of `endM(method)` is a tuple containing parameters of the `method` and a return value (if any) when the `method` ends. For example, suppose that `int Foo.sum(int,int)` is a monitored method and `Foo.sum(int,int)` has been invoked with a pair of parameters 1 and 2. When `Foo.sum(int,int)` returns, the event `endM(Foo.sum(int,int))` occurs. The event has three values.

    - parameters
        * `value(endM(Foo.sum(int,int)),0) = 1`
        * `value(endM(Foo.sum(int,int)),1) = 2`
    - return value
        * `value(endM(Foo.sum(int,int)),2) = 3`

We do not define the values of combined events. The only exception is `pe when cond` where `pe` is a primitive event. The value of `pe when cond` is associated with `value(pe)` when `cond` is true.

PEDL does not allow recursive definitions such as `condition c = x > 3 && c` in order to prevent diverging computation. In addition, PEDL does not provide quantifiers in defining events and condition. Therefore, a PEDL script is evaluated in linear time to the size of the specification (details on evaluation of a PEDL script, see Section 6.2.2). Fast event recognition is key requirement in PEDL. PEDL is designed to be an interface language to describe a *thin* interface between low-level implementation and high-level requirement. Complex specification of requirement properties should be written in MEDL.

Chapter 6

Java-MaC: a MaC Prototype for Java

The MaC architecture described in Chapter 4 is a general architecture not limited to any specific programming language. To demonstrate the effectiveness of the MaC architecture, however, we implement a MaC prototype for Java programs. This chapter describes a MaC prototype for Java programs, called *Java-MaC*. Figure 6.1 shows the overall structure of Java-MaC. Section 6.1 describes the details in the components of static phase of Java-MaC. Section 6.2 describes the details in the run-time components of Java-MaC.
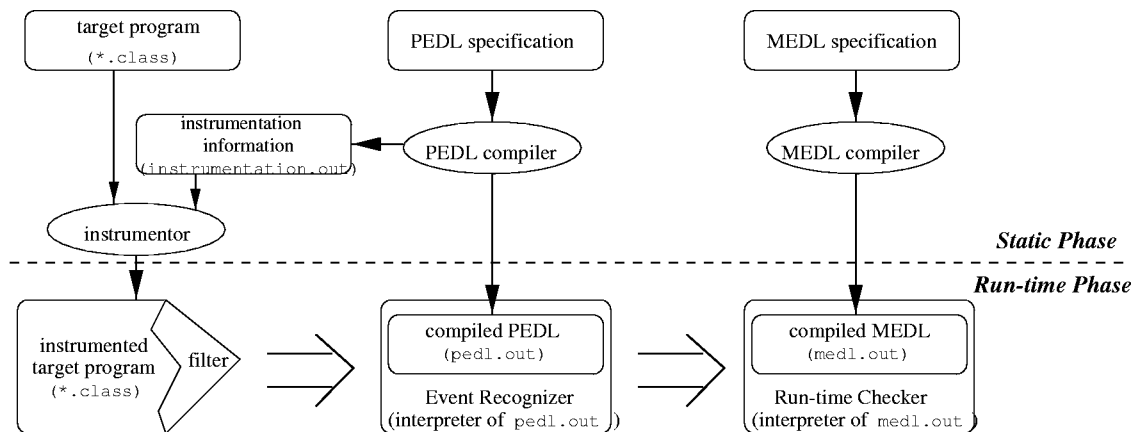
Figure 6.1: Overview of Java-MaC

6.1 Components of Static Phase

Java-MaC has three static phase components: a *PEDL compiler*, a *MEDL compiler*, and an *instrumentor*. A PEDL compiler compiles a PEDL script and generates a file `pedl.out`, which contains an abstract syntax tree (AST) of the PEDL script, which is evaluated by an event recognizer at run-time. At the same time, the PEDL compiler generates a file `instrumentation.out` containing instrumentation information which is used by the instrumentor. Similarly, a MEDL compiler compiles a MEDL script and generates

56

`medl.out` containing an AST which is evaluated by a run-time checker at run-time. A Java-MaC instrumentor takes Java classfiles (`*.class`) and instrumentation information (`instrumentation.out`) containing a list of monitored variables/methods and monitoring flags in a PEDL script. Based on these two inputs, a Java-MaC instrumentor inserts a filter into the target program.

Section 6.1.1 describes the structure of PEDL/MEDL AST and the algorithm of compiling PEDL/MEDL scripts. Section 6.1.2 explains the instrumentation process in high-level. Section 6.1.3 describes the instrumentation rules in detail.

6.1.1 PEDL/MEDL Compilers

`pedl.out` consists of three parts: *variable/method table*, *event/condition table*, and *event/condition trees*.

An entry of the variable/method table consists of *variable/method name*, *variable/method type*, *variable/method value*, and *updated flag* which indicates that a variable is updated or a method is invoked/returned at the time instant represented by a current snapshot. Values of entries in the variable/method table are initialized as undefined (see Section 4.2.2). The entry of the event/condition table consists of an *event/condition name*, an *exported flag* which indicates whether this event/condition is declared as `exported`, and a *pointer* to the root of event/condition tree. An event/condition tree contains an expression defining the corresponding event/condition based on program variables and methods whose values are in the variable/method table and other events/conditions whose trees are pointed from the event/condition table. There are four different types of nodes in event/condition trees.

- An *event* node contains the value of the event, a flag indicating that the event is present, and a timestamp.

- A *condition* node contains its old truth value and a current truth value. The old truth value is necessary for evaluating `start` and `end` construct. For example, in order to evaluate `start(A.x == 0)`, we need to know whether `A.x == 0` was false previously.

- A *reference* node contains a link to an event/condition in the event/condition table or a monitored variable/method in the value/method table.

- An *expression* node represents a constant or an arithmetic operator such as `+` or `-`. An expression node contains a value field.

A root of an event/condition tree has a *mark* which indicates whether this node is already evaluated so that an event/condition tree is not evaluated more than once (see the compilation algorithm in Figure 6.4).

Figure 6.3 shows the PEDL AST of Figure 6.2. There are two entries in the variable/method table - one for `A.x` and one for `A.y`. PEDL does not allow recursive expressions as we have stated in Section 5.2.2. Thus, an event/condition expression forms a tree. $n1$ is the root of a tree indicating the condition expression `A.x > 3`. Similarly, $n5$ is the root of a tree which represents the event expression `start(c1 && A.y < 10)`. $n3$ refers to the variable `A.x` in the variable/method table. $n8$ points to the root to the condition `c1`

57

```
MonScr test
  export event e1;
  monobj int A.x;
  monobj int A.y;

  condition c1 = A.x > 3;
  event e1 = start(c1 && A.y < 10);
end
```

Figure 6.2: A simple PEDL script

pedl.out
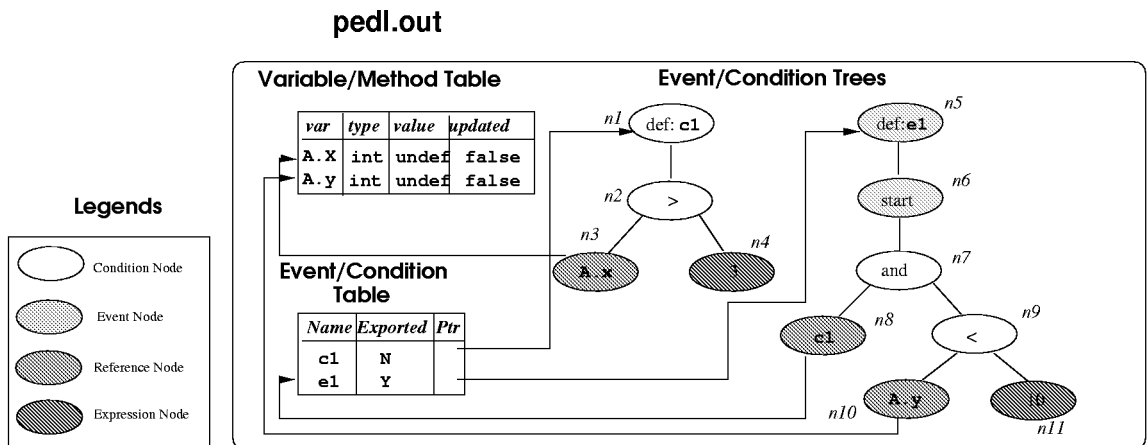
Figure 6.3: A PEDL AST of Figure 6.2

(a node $n1$) which represents the condition c1. The leaves of an event/condition tree are either reference nodes or expression nodes of constants.

Figure 6.4 describes an algorithm for compiling a PEDL script. A MEDL compiler compiles MEDL scripts in a similar way.[1]

```
Step 1. create the list of exported events/conditions
Step 2. create the variable/method table for monitored variables/methods
        and the event/condition table for event/condition definitions
Step 3. make entries for monitored variables/methods to the variable/method
        table
Step 4. for each event/condition definition,
        4.1 create an event/condition definition node
        4.2 if the event/condition table does not have an entry for the
            event/condition definition, create an entry for the definition
            node
        4.3 make a link to the node at the corresponding entry in the
            event/condition table and set an exported flag of the entry
            if the event/condition is declared as exported.
        4.4 create a child node corresponding to the sub-expression
            - for an operator/connective, generate a corresponding node.
              Then, create children nodes for the operands of the operator
              by repeating step 4.4.
            - for a reference to a monitored variable/method, generate a
              reference node which has a link to the corresponding entry
              in the variable/method table.
            - for a reference to an event/condition, generate a reference
              node. Then
                * if the event/condition table does not contain an entry for
                  the event/condition, create a corresponding entry in the
                  event/condition table.
                * make a link from the node to the corresponding entry in
                  the event/condition table
            - for a constant, generate a node containing the constant
```

Figure 6.4: Algorithm of compiling a PEDL script

6.1.2 Instrumentor

An executable Java program is instrumented by the Java-MaC instrumentor. An executable Java program consists of a set of *classfiles*. One classfile contains the definition of one class. A Java classfile is loaded into a running Java virtual machine at run-time. Thus, classfiles are linked dynamically at run-time rather than statically. In order to link classfiles dynamically, a classfile contains symbolic information such as string constants,

---

[1]PEDL/MEDL compilers uses Java Compiler Compiler [Web]

class names, field names, method names, local variable names[2], and other constants that are referred to within the classfile. This symbolic information in a classfile helps the Java-MaC instrumentor to recognize the instructions updating monitored variables and method invocations in the classfile. For example, suppose we would like to monitor a class variable x of type int in a class Foo.[3] putstatic, an instruction updating a class variable, has the symbolic name and type of the variable and the type of the parent class as operands. We can find an instruction putstatic Foo/x I ("I" means integer type) which updates the variable x by inspecting classfiles.

Java-MaC monitors three different entities of a Java program: primitive field variables, primitive local variables, and method beginnings/endings. Java-MaC instrumentor detects instructions which update monitored variables or instructions located at the beginnings/endings of methods and put probes which report new values of monitored variables or monitored execution points.

- *Primitive field variables*
  A field variable is either a *class variable* or a *instance variable*. A class variable is updated by putstatic instruction. An instance variable is updated by putfield instruction. Both instructions have a variable name and a parent class name. Both instructions take a top stack operand as a new value for the variable. The Java-MaC instrumentor inserts probes to instructions which update a monitored class variable by inspecting the instructions and its operands.[4] A probe obtains the new value for the monitored variable from the operand stack at run-time. putfield takes a reference to the parent object containing the variable as a stack operand additionally.

- *Primitive local variables*
  Local variables are updated by <T>store, <T>store_<n> and iinc where <T> is a primitive type and <n>∈ {0, 1, 2, 3}. These instructions contain an index to a local variable as an operand. A symbolic name to a local variable is available in a constant pool of classfile.

- *Execution points*
  There is only one starting point in a method - the beginning of a method definition. A method, however, can have several ending points where a return instruction exists. Parameters and return variable are obtained from operands stack.

Once an instruction (call it i) is recognized as updating a variable which may be a monitored variable, monitorenter Filter.lock is inserted right before i, and monitorexit Filter.lock is inserted right after i for making update of a variable and the report of that update an atomic session (see Section 6.2.1). Then, a probe invoking void sendObjMethod(Object parentAddress, <T> value, String varName) (parentAddress is an address of an object whose member field varName is monitored) is inserted right before i. An example of instrumented Java bytecode is in Figure 6.5.

---

[2]Local variable names are available in a classfile when source code is compiled with the -g debug flag

[3]A *class variable* is a variable statically declared for a class, not for an instance of a class. An *instance variable* is a variable declared for an instance of a class.

[4]We use JTREK [Dig] for modifying bytecode of the target program.

60

```
aload_1
aload_0
getfield CHARON/simulator/classes/DigitalVar/v I
    getstatic mac/filter/Filter/lock Ljava/lang/Object
    monitorenter
    dup2
    ldc "val"
    invokestatic mac/filter/SendMethods/sendObjMethod(
            Ljava/lang/Object;ILjava/lang/String;)V
putfield CHARON/simulator/classes/Var/val I
    getstatic mac/filter/Filter/lock Ljava/lang/Object
    monitorexit
```

Figure 6.5: Inserted probe in bytecode (indented lines indicate a probe)

sendObjMethod() checks whether a variable is actually the monitored variable (or beginning/ending of a method) by testing whether parentAddress matches the address of a monitored object in the address table (see Section 5.1).[5] If a variable is the monitored variable, sendObjMethod() sends the monitored variable (or beginning/ending of a method) and new value of the variable to an event recognizer.

6.1.3 Instrumentation Rules

This section describes instrumentation rules for monitored variables in order to illustrate the instrumentation process clearly. First, we define a classfile mathematically. Then, we describe instrumentation rules based on the definition of a classfile.

Definition of Classfile. A classfile c is a tuple of a qualified class name Id_c, a set of access modifiers $\mathcal{P}(Acc_c)$, the name of the direct super class (optional) $[Id_c]$, the names of direct super interfaces $\mathcal{P}(Id_c)$, field declarations FD, constant values CV, method declarations MD, method implementations MI, and constant pool CP. Method implementations MI map the signatures of methods Sig to the implementations $Impl$ which consists of the stack limit, the local variable limit, the exception handlers H, and the code array for that method $Code$. $Code$ maps a set of addresses PC to a set of instructions $Instr$. An instruction may have operands which are indicated by indexes to the constant pool CP of the class which contains symbolic names of classes, methods, and fields referred in the class or indexes to local variables. This structure of a classfile is in Table 6.1.[6]

Instrumentation Rules. We define the instrumentation of programs as function $\rightarrow_P$ from a program and monitored variables to the instrumented program. A program is a set of classfiles, where each classfile contains a set of method implementations. We define

---

[5]A probe checks whether a new value affects requirement properties when a value abstraction is enabled for reducing monitoring overhead (see Section 7.3.2).

[6]The definition of classfile is from [Ber97]

$$
\begin{aligned}
C &= Id_c \times \mathcal{P}(Acc_c) \times [Id_c] \times \mathcal{P}(Id_c) \times FD \times CV \times MD \times MI \times CP \\
MI &= Sig \rightarrow Impl \\
Impl &= N \times N \times H \times Code \\
H &= PC \times PC \rightarrow \mathcal{P}(Id_c) \rightarrow PC \\
Code &= PC \rightarrow Instr \\
Instr &= Opcode \times N^* \\
Opcode &= \{\texttt{aaload}, \texttt{aastore}, \texttt{aconst\_null}, ...\} \\
Sig &= Id_m \times Desc^* \\
PC &= N^+ \cup \{0\} (\text{a set of program counters}) \\
Id_c &= \text{a set of class names} \\
Id_m &= \text{a set of method names} \\
Desc &= \text{a set of parameter descriptions}
\end{aligned}
$$

Table 6.1: Classfile definition

$\rightarrow_P$ hierarchically. We define the function $\rightarrow_P$ in terms of the classfile instrumentation $\rightarrow_C$ which maps from a classfile and monitored variables to the instrumented classfile. We define $\rightarrow_C$ in terms of the method instrumentation $\rightarrow_M$ which maps from a method implementation and monitored variables to an instrumented method implementation which contains probes for monitored variables. We will use $vars \subseteq V$ as a set of monitored variables declared in a PEDL script where V is a set of program variables.

Definition 13 (Instrumentation of a program : $\rightarrow_P$) *For a program $p \subseteq C$, instrumentation of p with a set of monitored variables $mvars$ is defined as a mapping*

$$
\rightarrow_P \in \mathcal{P}(C) \times \mathcal{P}(V) \rightarrow \mathcal{P}(C)
$$

such that $(p, mvars) \rightarrow_P p'$ where $p' = \{c' | \exists c \in p.(c, mvars) \rightarrow_C c'\} \cup C_{Filter}$ where C_{Filter} is a set of Java-MaC filter classfiles.

Definition 14 (Instrumentation of a classfile : $\rightarrow_C$) *For a classfile $c \in C$, instrumentation of c with $mvars$ is defined as a mapping*

$$
\rightarrow_C \in C \times \mathcal{P}(V) \rightarrow C
$$

such that $(c, mvars) \rightarrow_C c'$ where $c = (id_c, acc, super, interfaces, fd, cv, md, mi)$ and $c' = (id_c, acc, super, interfaces, fd, cv, md, mi')$. mi' is an instrumented method implementation such that $mi' = \{(s, impl') \in Sig \times Impl \mid \exists (s, impl) \in mi. ((id_c, s, impl), mvars) \rightarrow_M (id_c, s, impl')\}$

Instrumentation of a method implementation $\rightarrow_M$ is defined by following three steps. First, the Java-MaC instrumentor creates a table containing the positions and the sizes

of probes in the method, which is defined as a mapping $\to_K$. Second, the instrumentor adjusts the target address of jump instructions in the method and an exception table using the table created by $\to_K$, which is defined as a mapping $\to_J$. Finally, the instrumentor inserts probes into a method implementation, which is indicated as a mapping performed by $\to_I$.

Definition 15 (Instrumentation of a method implementation: $\to_M$) *For a method implementation $m \in MI$, instrumentation of m with mvars is defined as a mapping*

$$\to_M \in MI \times \mathcal{P}(V) \to MI$$

such that $\to_M$ is defined as composition of the following three mappings

- *probe table creation $\to_K$*

- *jump address adjustment $\to_J$*

- *probe insertion $\to_I$*

In other words

$$\to_M \stackrel{def}{=} \to_I \circ \to_J \circ \to_K$$

The following paragraphs define $\to_I$, $\to_J$, and $\to_K$. We start with clarification of the notations we will use in the definitions.

Notations

- A data type $<T> \in \{\texttt{i}, \texttt{l}, \texttt{f}, \texttt{d}, \texttt{a}\}$.

- A set of opcodes for storing local variable $L\_STR = \{<T>\texttt{store}, <T>\texttt{store\_<n>}, \texttt{iinc}\}$ where $<\texttt{n}> \in \{0, 1, 2, 3\}$

- A set of opcodes for storing field variable $F\_STR = \{\texttt{putfield}, \texttt{putstatic}\}$.

- A set of opcodes for jump $JMP = \{\texttt{goto}, \texttt{goto\_w}, \texttt{if<cond>}, \texttt{ifnull}, \texttt{ifnonnull}, \texttt{if\_acmp\_<cond>}, \texttt{if\_icmp\_<cond>}, \texttt{jsr}, \texttt{jsr\_w}\}$ where $<\texttt{cond}> = \{\texttt{eq}, \texttt{ne}, \texttt{lt}, \texttt{le}, \texttt{gt}, \texttt{ge}\}$.

- For $i \in Instr$, $opc_i \in Opcode$ indicates operation code of instruction i, $op_{ij} <T>$ is jth operand of i which has a type $<T>$.

- $c; i$ is a code which starts with instructions c and ends with instruction i.

We use several shorthand notations.

- $sig\_m \stackrel{def}{=} (id_c, id_m, desc)$ for the signature of a method.

- $hd\_m \stackrel{def}{=} (n_0, n_1, h)$ for a header of a method where n_0 is the size of the operand stack, n_1 is the number of local variables and h is the exception table.

- $meth \stackrel{def}{=} (sig\_m, hd\_m)$ for all method implementation components except code.

- pc_i is the location of instruction i.

Creating a table for probes' positions and sizes: $\rightarrow_K$

We define a mapping

$$\rightarrow_K \in Id_c \times Sig \times Impl \rightarrow Id_c \times Sig \times Impl \times (PC \rightarrow N)$$

in order to create a set of instrumentation positions and size of the probes. $\rightarrow_K$ is defined inductively in Table 6.2. *Init* is a rule for empty code as the base case in induction. *NoChange* is a rule for instructions where no probe is inserted. *LocalStore*, *wide* and *FieldStore* are rules for instructions where probes are inserted. In Table 6.2, s_{probe} is the size of an inserted probe.

Jump address adjustment rules: $\rightarrow_J$

We define a mapping

$$\rightarrow_J \in Id_c \times Sig \times Impl \times (PC \rightarrow N) \rightarrow Id_c \times Sig \times Impl$$

in order to adjust target addresses of jumping instructions and the exception table. We use a utility function

$$getBytes \in PC \times PC \rightarrow N.$$

$getBytes(bound_0, bound_1)$ returns a number of bytes newly added by probes inserted between $bound_0$ and $bound_1$ based on the table generated by $\rightarrow_K$. h' is an adjusted exception table of h, i.e, $h' = \{(pc'_0, pc'_1, e, pc'_2) \mid \exists (pc_0, pc_1, e, pc_2) \in h.\ pc'_0 = pc_0 + getBytes(0, pc_0)$. Similarly for pc'_1 and $pc'_2\}$.

$\rightarrow_J$ is defined inductively in Table 6.3. *Init* is a rule for empty code as the base case in induction. *NoChange* is the rule for non-jump instructions. *JMP* is the rule for jump instructions whose target address should be adjusted.

Instrumentation rules: $\rightarrow_I$

We define a mapping

$$\rightarrow_I \in Id_c \times Sig \times Impl \rightarrow_I Id_c \times Sig \times Impl$$

in order to obtain an instrumented method. We will use shorthand notations for probes.

- $probe_s(< vn >, <T>) \stackrel{def}{=}$

```
getstatic mac/filter/Filter/lock Ljava/lang/Object;
monitorenter;
dup2;
ldc <vn>
invokestatic mac/filter/SendMethods/sendObjMethod(
    Ljava/lang/Object;<T>Ljava/lang/String;)V
```

  where $< vn >$ is a string representing a monitored variable name and $<T>$ is a type of a monitored variable.

64

$$[Init](meth, \epsilon) \rightarrow_K (meth, \epsilon, \emptyset) \tag{6.1}$$

$$[LocalStore] \frac{\begin{array}{c} opc_i \in L\_STR \\ id_c.id_m.op_{i1} \prec T \succ \in mvars \\ (meth, c) \rightarrow_K (meth, c, l) \end{array}}{(meth, c; i) \rightarrow_K (meth, c; i, l \cup \{(pc_i, s_{probe})\})} \tag{6.2}$$

$$[wide] \frac{\begin{array}{c} opc_i = wide \\ op_{i1} \in \{\texttt{iinc}, \prec T \succ \texttt{store}\} \\ id_c.id_m.op_{i2} \prec T \succ \in mvars \\ (meth, c) \rightarrow_K (meth, c, l) \end{array}}{(meth, c; i) \rightarrow_K (meth, c; i, l \cup \{(pc_i, s_{probe})\})} \tag{6.3}$$

$$[FieldStore] \frac{\begin{array}{c} opc_i \in F\_STR \\ op_{i1} \prec T \succ \in mvars \\ (meth, c) \rightarrow_K (meth, c, l) \end{array}}{(meth, c; i) \rightarrow_K (meth, c; i, l \cup \{(pc_i, s_{probe})\})} \tag{6.4}$$

$$[NoChange0] \frac{\begin{array}{c} opc_i \notin L\_STR \cup \{\texttt{wide}\} \cup F\_STR \\ (meth, c) \rightarrow_K (meth, c, l) \end{array}}{(meth, c; i) \rightarrow_K (meth, c; i, l)} \tag{6.5}$$

$$[NoChange1] \frac{\begin{array}{c} opc_i \in L\_STR \\ id_c.id_m.op_{i1} \prec T \succ \notin mvars \\ (meth, c) \rightarrow_K (meth, c, l) \end{array}}{(meth, c; i) \rightarrow_K (meth, c; i, l)} \tag{6.6}$$

$$[NoChange2] \frac{\begin{array}{c} opc_i = \texttt{wide} \\ \left(\begin{array}{c} op_{i1} \in \{\prec T \succ \texttt{store}, \texttt{iinc}\} \\ id_c.id_m.op_{i2} \prec T \succ \notin mvars \end{array} \right) \bigvee op_{i1} \notin \{\prec T \succ \texttt{store}, \texttt{iinc}\} \\ (meth, c) \rightarrow_K (meth, c, l) \end{array}}{(meth, c; i) \rightarrow_K (meth, c; i, l)} \tag{6.7}$$

$$[NoChange3] \frac{\begin{array}{c} opc_i \in F\_STR \\ op_{i1} \prec T \succ \notin mvars \\ (meth, c) \rightarrow_K (meth, c, l) \end{array}}{(meth, c; i) \rightarrow_K (meth, c; i, l)} \tag{6.8}$$

Table 6.2: Rules for $\rightarrow_K$

$$[Init](sig\_m, n_0, n_1, h, \epsilon, l) \rightarrow_J (sig\_m, n_0, n_1, h', \epsilon) \tag{6.9}$$

$$[NoChange0] \frac{opc_i \notin JMP \quad (meth, c, l) \rightarrow_J (meth, c')}{(meth, c; i, l) \rightarrow_J (meth, c'; i)} \tag{6.10}$$

$$[JMP] \frac{opc_i \in JMP \quad (meth, c, l) \rightarrow_J (meth, c')}{(meth, c; i, l) \rightarrow_J (meth, c'; i[op_{i1}/addr]) \ where} \tag{6.11}$$
$$addr = op_{i1} + getBytes(pc_i, pc_i + op_{i1})$$

Table 6.3: Rules for $\rightarrow_J$

- $probe_e \overset{def}{=}$

```
getstatic mac/filter/Filter/lock Ljava/lang/Object;
monitorexit;
```

$\rightarrow_I$ is defined inductively in Table 6.4 describing rules which insert probes and Table 6.5 describing rules which do not insert probes. *Init* is the rule for empty code as the base case in induction. *Init* also increases the size of the stack by three because a parent object address, a value of monitored variable, and a variable name are passed to `sendObjMethod()`. *NoChange* is the rule for instructions where no probe is inserted. *LocalStore*, *wide* and *FieldStore* are the rules for instructions where probes are inserted.

6.1.4 Side Effects of Java-MaC

Java-MaC instrumentor does not change the architecture of the target program. In other words, Java-MaC instrumentor does not change the interfaces between classes consisting of variables and methods declarations. We can prove this using the instrumentation rules $\rightarrow_P$, $\rightarrow_C$, and $\rightarrow_M$ defined in Section 6.1.3. The main idea of proof is to show that Java-MaC instrumentor does not modify the contents of classfiles except the instructions of methods.

We call a class c and another class c' *equivalent* if and only if all the element of c and c' are equal except method implementation mi . Similarly, we call a method implementation mi and another method implementation mi' *equivalent* if and only if the signature of mi is equal to the signature of mi'. A class c' is a new class to the program π if π does not have a class equivalent to c'. Similarly, a method implementation mi' is a new method to π if π does not have a method implementation equivalent to mi'.

Theorem 2 *Java-MaC instrumentor does not change the architecture of the target program by prohibiting the following activities.*

Class *addition of new classes (except C_{filter}) to the the target program or removal of any classfiles from the target program.*

66

$$[Init](sig\_m, n_0, n_1, h, \epsilon) \rightarrow_I (sig\_m, n_0 + 3, n_1, h, \epsilon) \tag{6.12}$$

$$[LocalStore]\frac{\begin{array}{c} opc_i \in \{\texttt{<T>store}\} \\ id_c.id_m.op_{i1} :\!\!<\!T\!\!> \in mvars \\ (meth, c) \rightarrow_I (meth, c') \end{array}}{(meth, c; i) \rightarrow_I (meth, c'; probe_s(op_{i1},\texttt{<T>}); i; probe_e)} \tag{6.13}$$

$$[Wide0]\frac{\begin{array}{c} opc_i = \texttt{wide} \\ op_{i1} \in \{\texttt{<T>store}\} \\ id_c.id_m.op_{i2} :\!\!<\!T\!\!> \in mvars \\ (meth, c) \rightarrow_I (meth, c') \end{array}}{(meth, c; i) \rightarrow_I (meth, c'; probe_s(op_{i2},\texttt{<T>}); i; probe_e)} \tag{6.14}$$

$$[Wide1]\frac{\begin{array}{c} opc_i = \texttt{wide} \\ op_{i1} = \texttt{iinc} \\ id_c.id_m.op_{i2} :\!\!<\!T\!\!> \in mvars \\ (meth, c) \rightarrow_I (meth, c') \end{array}}{(meth, c; i) \rightarrow_I (meth, c'; probe_s(op_{i2},\texttt{<T>}); i; probe_e)} \tag{6.15}$$

$$[FieldStore0]\frac{\begin{array}{c} opc_i = \texttt{putfield} \\ op_{i1} :\!\!<\!T\!\!> \in mvars \\ (meth, c) \rightarrow_I (meth, c') \end{array}}{(meth, c; i) \rightarrow_I (meth, c'; probe_s(op_{i1},\texttt{<T>}); i; probe_e)} \tag{6.16}$$

$$[FieldStore1]\frac{\begin{array}{c} opc_i = \texttt{putstatic} \\ op_{i1} :\!\!<\!T\!\!> \in mvars \\ (meth, c) \rightarrow_I (meth, c') \end{array}}{(meth, c; i) \rightarrow_I (meth, c'; probe_s(op_{i1},\texttt{<T>}); i; probe_e)} \tag{6.17}$$

Table 6.4: Rules for $\rightarrow_I$ which insert probes

$$[NoChange0] \frac{opc_i \notin L\_STR \cup \{\texttt{wide}\} \cup F\_STR \cup JMP \quad (meth, c) \rightarrow_I (meth, c')}{(meth, c; i) \rightarrow_I (meth, c'; i)} \quad (6.18)$$

$$[NoChange1] \frac{\begin{array}{c} opc_i \in L\_STR \\ id_c.id_m.op_{i1} \preceq T \succ \notin mvars \\ (meth, c) \rightarrow_I (meth, c') \end{array}}{(meth, c; i) \rightarrow_I (meth, c'; i)} \quad (6.19)$$

$$[NoChange2] \frac{\begin{array}{c} opc_i = \texttt{wide} \\ \left(\begin{array}{c} op_{i1} \in \{\preceq T \succ \texttt{store, iinc}\} \\ id_c.id_m.op_{i2} \preceq T \succ \notin mvars \end{array} \right) \; \bigvee \; op_{i1} \notin \{\preceq T \succ \texttt{store, iinc}\} \\ (meth, c) \rightarrow_I (meth, c') \end{array}}{(meth, c; i) \rightarrow_I (meth, c'; i)} \quad (6.20)$$

$$[NoChange3] \frac{\begin{array}{c} opc_i \in F\_STR \\ op_{i1} \preceq T \succ \notin mvars \\ (meth, c) \rightarrow_I (meth, c') \end{array}}{(meth, c; i) \rightarrow_I (meth, c'; i)} \quad (6.21)$$

Table 6.5: Rules for $\rightarrow_I$ which do not insert probes

Field *addition/removal of field variables to/from the classfiles*

Method *addition/removal of method declarations or method implementations to/from the classfiles*

Local *addition/removal of local variables to/from the methods of the classfiles*

Instr *removal of any existing instructions in the methods of the classfiles nor adds instructions other than ones defined as $probe_s(< vn >, < T >)$ and $probe_e$.*

Proof of [Class]: $\rightarrow_P$ applies $\rightarrow_C$ to all the classfiles of the target program π and generates an instrumented target program π'. Definition 14 shows that $\rightarrow_C$ preserves all elements of a classfile except mi. Thus, for any instrumented classfile $c' \in \pi'$, there is an equivalent classfile c in pi.

Proof of [Field]: Similar to the proof of [Class] because $\rightarrow_C$ preserves field declarations fd.

Proof of [Method]: $\rightarrow_C$ preserves method declaration md. $\rightarrow_C$ applies $\rightarrow_M$ to every method implementation mi. Definition 15 shows that $\rightarrow_M$ is defined as $\rightarrow_I \circ \rightarrow_J \circ \rightarrow_K$. Table 6.2 shows that $\rightarrow_K$ preserves the signature of mi. Similarly, Table 6.3, Table 6.4, and Table 6.5 show that $\rightarrow_J$ and $\rightarrow_I$ do not change the signature of method implementation mi.

Thus, for any instrumented method implementation mi' in instrumented classfile $c' \in \pi$, there is an equivalent method implementation mi in a classfile $c \in \pi$

Proof of [Local]: Similar to the proof of [Method] because none of $\rightarrow_I, \rightarrow_J$, and $\rightarrow_K$ changes the local variables.

Proof of [Instr]: Similar to the proof of [Method] because none of $\rightarrow_I, \rightarrow_J$, and $\rightarrow_K$ removes an instruction nor add instructions other than $probe_s(<vn>, <T>)$ and $probe_e$.

$\square$

Although the architecture of the target program is preserved as shown in Theorem 2, there are side effects to the target program by Java-MaC. The following list is possible side effects caused by Java-MaC.

- **Timing of execution.** A speed of the target program can be slowed due to probes' execution. A real-time application may violate temporal requirements because Java-MaC slows down the application.

- **Thread scheduling.** The execution order among the threads of the target program may change due to slowed execution speed. This changed execution order may cause the violation of requirements. It should be noted, however, that altered execution order does not affect the correctness of a target program when the target program implements synchronization mechanism correctly.[7]

- **Resource Limitation.** The amount of resources that JVM can consume is finite. Java-MaC consumes resources, effectively reducing the available resource to the target program. [LY99] specifies the limit on several resources. The resources are:

  - **Operand stack.** Each method specifies the maximum size of the operand stack. Inserted instructions by the Java-MaC put additional operands onto operand stack to pass to sendObjMethod(...). Therefore, the size of operand stack increases, thus reducing the remaining space for the original program (The operand stack size must be less than 65536 ($= 2^{16}$).)

  - **Method stack.** Each thread has one method stack. When a method is invoked, a new frame for the method is created and loaded on the method stack. Each probe invokes sendObjMethod(...). Therefore, the method stack size increases, which can reduce the remaining space for the original program. (The Java stack size must be less than 65536.)

  - **Constant pool.** The Java-MaC inserts instructions referring to methods and variables of Java-MaC. Therefore, the constant pool additionally must contain symbolic names of Java-MaC. (The constant pool must be less than 65536.)

  - **Size of a method and a class file.** Java-MaC inserts probes into a method implementation. Thus, the size of the instrumented method increases. (The size of method must be less than 65536 instructions.)

---

[7]A program has a *synchronization error* if the program behaves incorrectly when extraneous delays are introduced in threads [Gai86].

– **Size of heap.** The size of the heap increases because of Java-MaC data structures.

- **Target address of a jump instruction and exception handler.** A target address of a jump instruction should be adjusted if probes are inserted between a jump instruction and its target address. An exception handler is declared by a range indicated by a pair of beginning/ending locations and a type of exception which might happen inside the range. A range declaration of an exception handler should be modified, too.[8]

6.2 Components of Run-time Phase

6.2.1 Filter

A filter extracts snapshots from the target program and sends these snapshots to the event recognizer in correct order. A filter consists of following three parts:

- *a communication channel*
  A target program is not originally designed to communicate with an event recognizer. A communication channel from the target program to the event recognizer is created by a filter. The type and the destination of communication is decided when the target program is instrumented.

- *probes*
  Probes are inserted into the *all* locations of monitored variables updates (or beginnings/endings of monitored methods). A probe extracts the new value of a monitored variable and sends the value (or beginning/ending signals of monitored methods) to the event recognizer through the communication channel of a filter.

- *a filter thread*
  A filter thread flushes the content of the communication buffer to the event recognizer through the communication channel.

A filter might report updates of monitored variables of a multi-threaded target program differently from what really happens in the target program due to preemptions in thread scheduling. Consider the following example in Figure 6.6. `ldc` is a bytecode instruction to load a constant into an operand stack. `putfield` x is a bytecode instruction to store a top operand in the operand stack into a field variable x. `sendObjMethod()` reports monitored variable updates to an event recognizer. The value `sendObjMethod()` sends is obtained from a top operand in the operand stack.

In Figure 6.6, `y` is updated earlier than `x`. The update of `x`, however, is reported earlier than that of `y` because a preemption occurs between `sendObjMethod()` and `putfield x`.[9] There are two conceivable solutions for this problem. The first solution is making an update instruction and a probe an atomic session using a global lock. This solution guarantees

---

[8]A jump instruction uses a 16 bit offset which can express any address in a method whose size is limited to less than 65536. Therefore, the increased offset due to probe insertion does not cause problem.

[9]Even when `sendObjMethod()` is inserted right after `putfield`, there still can be incorrect ordering of reporting snapshots.

```
              Thread 1              Thread 2

                  |
              ldc 10
preemption    sendObjMethod("x",10)
by scheduler ............................................|..................

                              ldc 20
                              sendObjMethod("y",20)

                              putfield y

preemption                        |
by scheduler ............................↓..................
              putfield x

                  ↓
```
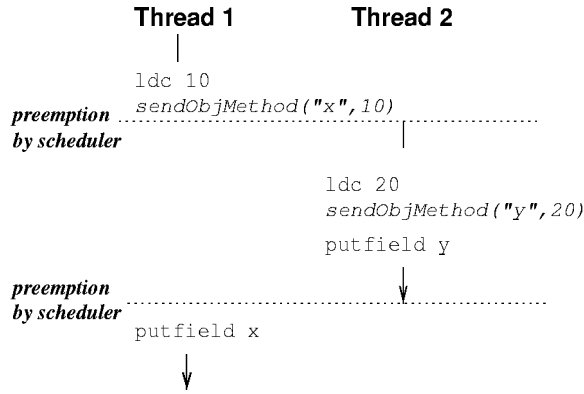
Figure 6.6: Incorrect ordering of reporting snapshots

correct order among snapshots. This solution, however, incurs two overhead costs. The first overhead is making threads serialized while the threads are updating monitored variables. The second overhead is an acquiring and releasing the lock at run-time.

The second solution is to use two timestamps, one to assign immediately before an update of a monitored variable (say x), and the other to assign immediately after the probe for the variable. If the time interval between these two timestamps of update of x overlaps with the interval between the timestamps of updating y, the event recognizer can recognize x may precede y or the other way. We, however, cannot know whether x precedes y or not in this case.

We choose the atomic session implementation because the atomic session solution guarantee the correct ordering of snapshots. A thread i acquires a global lock right before an update instruction. After finishing the update and report, thread i releases the lock. When preemption happens before thread i finishes the report, no other thread j can make a report because thread i holds the lock. Thread j has to wait until thread i finishes the report and release the lock.

We observe that locking is unnecessary when monitored variables are updated by one thread only. PEDL for Java provides a keyword multithread (see Section 5.2). When this keyword is used, probes performing lock/unlock are inserted. Otherwise, probes performing lock/unlock are not inserted. Figure 6.7 shows the structure of a filter.

6.2.2 Event Recognizer

An event recognizer evaluates events and conditions defined in a PEDL AST when it receives a snapshot from a filter. Figure 6.8 describes an algorithm for evaluating a PEDL AST. The evaluation process will be illustrated using Figure 6.3 which is an AST of Figure 6.2. Suppose that an event recognizer has A.x as 2 and A.y as 5. Then, suppose a filter sends a snapshot of A.x as 5. When the event recognizer receives the snapshot, it first updates A.x as 5 in the value/method table. Then the event recognizer cleans up marks of all roots of event/condition trees. Then, the event recognizer starts evaluating events and condition expressions in a bottom-up way. The order of evaluating event/condition trees does not affect the evaluation result. In other words, whether the event recognizer starts evaluation from $n1$ or $n5$ does not change the result of evaluation. Suppose that
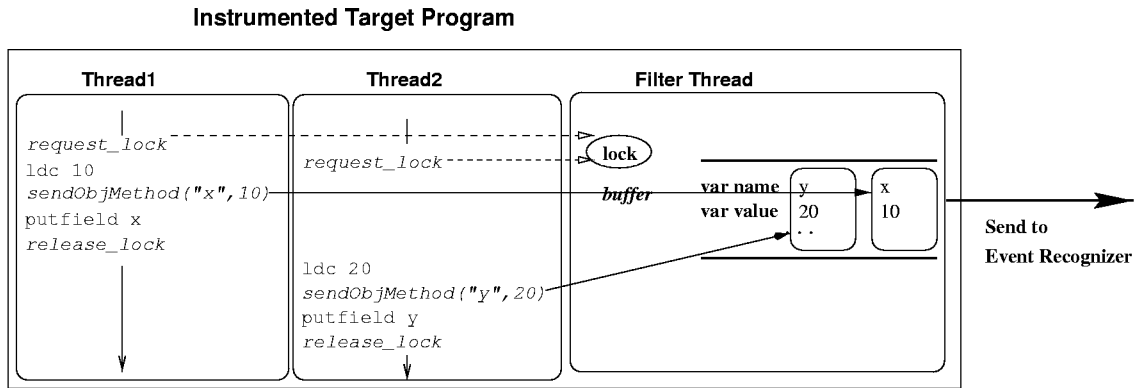
Figure 6.7: Structure of a filter

$n1$ is evaluated first. First, a mark of $n1$ is set. Then, a truth value field of $n1$ (**false**) is copied into its old truth value field and the value of $n1$ changes to *true* because a new value of **A.x** (5) is greater than 3. Although **c1** has changed its value, **c1** is not exported because it is not declared as exported. Next, $n5$ is evaluated. When $n8$ is evaluated, the event recognizer recognizes that $n1$ has been already evaluated by looking at the mark of $n1$. Thus, the event recognizer does not evaluate $n1$ again, but just uses the flag of $n1$. Finally, $n5$ is evaluated as present because $n9$ is true (**A.y** (5) < 10). **e1** is declared as exported. Thus, **e1** is exported to a run-time checker.

PEDL expressions are evaluated in linear time to the size of expression. This is for two reasons. First, PEDL AST contains event/condition definitions as trees, not cyclic graphs. Remember that PEDL/MEDL do not allow recursive expressions. Second, an event recognizer traverse the nodes of PEDL AST only once, by using *marks* of event/condition definition nodes.

6.2.3 Run-time Checker

A run-time checker evaluates events and conditions defined in an MEDL AST (**medl.out**) when it receives an event or a condition from an event recognizer. MEDL AST has similar structure as PEDL AST. MEDL AST, however, additional structure for auxiliary variables and auxiliary variable updates, which do not exist in PEDL. [Vis00] has more details for an evaluation process of a run-time checker. MEDL expressions can be evaluated in linear time to the size of MEDL script similarly to PEDL expressions are evaluated. If the evaluation detects a violation defined by **alarm** or **property**, the run-time checker raises a signal.

6.2.4 Connection of the Run-time Components

A connection among the Java-MaC run-time components is established before running a target application. The communication medium through which connection is established is important because of the following reasons. First, a communication medium affects the correctness of checking. Java-MaC assumes that the communication medium guarantees delivery of snapshots in order. If a communication medium does not provide that

```
Step 1. Set the current time with the timestamp of the received snapshot
Step 2. Update the variable/method table according to the snapshot
Step 3. Clear all the marks of the roots of event/condition trees
Step 4. For each event/condition tree
        4.1 Set the mark of the root
        4.2 Evaluate children nodes
            - For an operator/connective node, getting the value of the node
              by performing corresponding operation on the values of the
              children nodes which are obtained by repeating step 4.2
              For example:
                * For an <start> event node, set the present flag and set
                  the timestamp of the node with the current time if a
                  condition corresponding to the child node has changed to
                  true.  Reset the present flag otherwise.
            - For a reference node for a variable/method, getting the value
              of the node by obtaining the value of the corresponding entry
              in the variable/method table
            - For a reference node for an event/condition, getting the
              present flag/truth value of the node by obtaining the present
              flag/truth value of the root of the corresponding
              event/condition tree
                * if the root has mark set, getting the flag/value of the root
                * if not, repeat step 4.1 and 4.2 for the children of the root
            - For a constant node, the value of the node is the constant value
Step 5. Export events which are declared as exported and whose present flags
        are set (such events can be found through the event/condition table)
Step 6. Export conditions which are declared as exported and whose old
        values and new values are different (such conditions can be found
        through the event/condition table)
```

Figure 6.8: Algorithm of evaluating a PEDL AST

guarantee, the correctness of checking cannot be guaranteed either. In other words, the order of snapshots affects the evaluation of properties. For example, suppose `event e = update(x) when y == 2`. Suppose snapshots are arrived in the following order: `y=2`, `x=1`, and `y=1`. Then, the event recognizer detects `e`. The event recognizer, however, does not detect `e` if the arrival order of snapshots is `y=2`, `y=1`, and `x=1`. Second, application specific constraints may be given to the communication medium. For example, if Java-MaC monitors a program handling security information such as credit card of customers, snapshots of the target system should be delivered to the event recognizer securely.

Another important issue in communication is *buffering*. We can send snapshots as soon as possible without buffering in the communication channel (*unbuffered communication*). Or we can store snapshots into a buffer and send the content of the buffer when the buffer becomes full (*buffered communication*).

- *Unbuffered communication* has minimum delay of sending snapshots to the event recognizer. Unbuffered communication, however, causes high communication overhead if a read/write operation to the communication channel between a filter and an event recognizer has high per-read/write overhead.

- *Buffered communication* has delay of sending snapshots to the event recognizer because a filter postpones sending snapshots to the event recognizer until the buffer becomes full. Buffered communication, however, decreases the communication overhead compared to unbuffered communication if a read/write operation to the communication channel between a filter and an event recognizer has high per-read/write overhead.

  Buffered communication can use *blocking buffer* which blocks writing snapshots into the buffer when the buffer is full or *non-blocking buffer* which allows overwriting to the buffer even when the buffer is full. Blocking buffer may slow down the target program by blocking. Non-blocking buffer does not slow down the target program. However, non-blocking buffer can lose its content by overwriting (see Sentry in Section 2.2.3).

A user can implement his/her own communication channel to satisfy application specific need and connect the Java-MaC run-time components using this channel. For example, a user can write a code to make Java-MaC components communicate with each other through Secure Socket Layer [SSL96]. A filter uses an `OutputStream` provided by a user to send snapshots to an event recognizer. Similarly, an event recognizer uses an `InputStream` and an `OutputStream` provided by a user to communicate with a filter and run-time checker. `InputStream` and `OutputStream` are provided to the Java-MaC run-time components by following Java-MaC APIs.

- In `mac.filter.Filter`,

  ```
  static void hook(java.lang.String targetClass, java.lang.String[] args,
                   java.io.OutputStream os)
  ```

  This method is used to invoke the instrumented target program and connect the target program with an event recognizer through `OutputStream os` provided as a parameter.

74

- In mac.eventRecognizer.interpreter.EventRecognizer

  ```
  EventRecognizer(java.io.InputStream inStream, java.io.OutputStream outStream
                  ParserErObject pedl_out, PedlInstrObject objMonEntities)
  ```

  This method creates and executes an event recognizer which communicates with a filter and a run-time checker through InputStream inStream and OutputStream outStream provided as parameters.

- In mac.runtimeChecker.interpreter.Checker

  ```
  Checker(java.io.InputStream inStream, ParserChkObject medl_out)
  ```

  This method creates and executes an run-time checker with InputStream inStream.

Figure 6.9 and Figure 6.10 show an example of connecting a target program and an event recognizer using SSL channel. Assume that there exists a SSL package containing SSLOutStream and SSLInputStream. Assume further that a main class of the target program is TargetPgm.class and it does not get any argument. Lines 5 to 6 of Figure 6.9 creates an output stream to cis.upenn.edu:8040 secured by SSL. Line 7 invokes the instrumented target program TargetPgm and connects the target program to an event recognizer through SSL secured outStream. Line 4 in Figure 6.10 creates a SSL input stream inStream at port 8040 through which the event recognizer receives snapshots from the filter. Line 7 creates a SSL output stream outStream to ee.upenn.edu:8050. pedl.out are read in lines 10 and 11. Line 13 executes an event recognizer with inStream and outStream.

```
01:public class TargetProgramSSL {
02:    public static void main(String[] args){
03:        // Create SSL connection to the event recognizer
04:        // at cis.upenn.edu:8040
05:        OutputStream outStream =
06:            new SSLOutStream("cis.upenn.edu",8040);
07:        mac.filter.Filter.hook("TargetPgm", new String[] {}, outStream);
08:    }
09:}
```

Figure 6.9: A target program sending snapshots through SSLOutputStream

The standalone run-time components of Java-MaC can use one of two pre-implemented communication methods for the convenience of users.

- TCP socket communication with a blocking buffer

- file communication with a blocking buffer

```
01:public class EventRecognizerSSL{
02:     public static void main(String[] args){
03:            // Create SSL connections from filter at port 8040
04:            InputStream inStream = new SSLInputStream(8040);
05:            // Create SSL connection to the run-time checker at
06:            // ee.upenn.edu:8050
07:            OnputStream outStream = new SSLOutputStream("ee.upenn.edu", 8050);
08:
09:            // Read PEDL AST
10:            ParserErObject pedl_out =
11:                (ParserErObject)(new ObjectInputStream(...));
12:
13:            new EventRecognizer(inStream, outStream, pedlOut).start();
14:     }
15:}
```

Figure 6.10: An event recognizer receiving snapshots through SSLInputStream

TCP socket communication is flexible because it can connect the run-time components running on separate but network-connected hosts or different processes on same host. It poses, however, high communication overhead. The run-time components can communicate each other through a FIFO file if the run-time components of Java-MaC run on hosts which share a file system. Furthermore, a trace containing the execution of the target program or a trace containing a history of detected events/condition can be stored into files using this communication method. These trace files can be analyzed later for checking different requirements without running the system again (see Section 8.3). The command line usage of the Java-MaC standalone components are described in Appendix E.

Chapter 7

Overhead Reduction Techniques

Java-MaC runs together with the target program. The overhead caused by Java-MaC should be modest so that the execution of the target program is not slowed down significantly. In this chapter, we analyze the overhead, describe overhead reduction techniques, and show the experimental results using these techniques. First, we build an overhead model. Second, we will develop techniques to reduce the overhead based on the model. Finally, we perform experiments measuring the overhead caused by Java-MaC with and without these overhead reduction techniques.

7.1 Modeling of the MaC architecture

7.1.1 An Example of Product Distribution/Sales System

We start with an illustration of a real-world distribution/sales system to give intuitive idea about the performance of Java-Mac. Suppose we receive a request from a baseball company to distribute their baseballs from their factory to a retail store and sell the balls at the store. Person A in Figure 7.1 represents the company and produces baseballs. B puts the baseballs in boxes one by one and load these boxes into a truck C. When C is filled completely, C delivers the boxes to the retailer D. D checks whether a ball is faulty or not and sell the ball to customers. We are in charge of the process B, C, and D.

Our *goal* is to increase the overall speed of the distribution/sales system by improving processes B, C, and D. The processes B, C, and D form a pipeline. Thus, the slowest one among B, C, and D congests the whole processes. For example, let us see the following situation.

- D checks 1 ball per 1 min.

- C's capacity is 100 boxes. C takes 10 min from B to D.[1]

- B put 1000 balls into boxes and load these boxes into C per 10 min.

The bottleneck is D now; we cannot sell more than 1 ball per 1 min. If D's speed is increased to check 1000 balls per 1 min. Then the bottleneck is C; we cannot sell more

---

[1]Assume that C can return to B from D in no time.
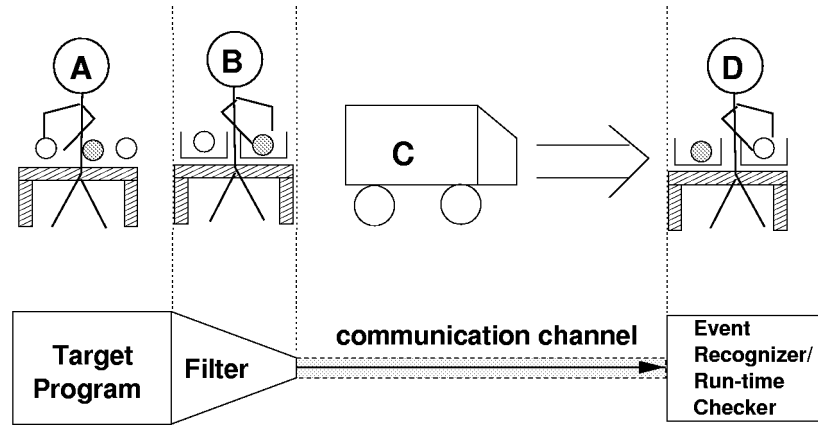
Figure 7.1: Comparison between a distribution/sales system and the MaC architecture

than 10 balls per 1 min (= 100 boxes/10 min). Therefore, we need to improve all processes B, C, and D so that any of these processes may not be a significant bottleneck. At D, we can improve the checking procedure so that checking takes less time. At C, we can increase the velocity of the truck or expand the capacity of the truck. At B, we have several ways to improve the speed. First, we can make B check whether a ball is faulty or not *briefly*. If a ball is proved faulty by this brief checking, B throws it away so that B can load more sellable balls in C. This also helps D to sell balls faster because D will handle more sellable balls in the same amount of time due to the elimination of faulty ones by B. The speed of B itself is decreased by this extra work of checking. This check at B, however, can reduce the whole processing time, if C or D is a bottleneck. Second, B can use smaller boxes so that B can load more boxes to C. Third, we can improve the wrapping speed of B. As we have described, B has more room to improve the performance than C and D. In addition, the improvement of B can increase the speed of B and C, too. Thus, B should be the first spot to improve.

There exists an analogy between this distribution/sales system and Java-MaC. A is a target program. B is a filter. C is a communication channel between the filter and the event recognizer. D is an event recognizer and a checker. A box is a snapshot and the size of box is the size of snapshot. Checking of a ball is to check whether properties are affected by this snapshot. Faulty ball is a snapshot which does not affect the requirement properties. We will concentrate on techniques applying at the filter because the filter has significant potentials to improve the whole system as we have seen in the example.

7.1.2 Overhead Model of the MaC architecture

We make an overhead model consisting of a target program and a monitor which consists of an event recognizer and a run-time checker.[2] Figure 7.2 shows the model. The following six parameters constitute overheads in the model.

- p is the overhead of executing a probe inserted into the target program.

---

[2]This model does not consider a communication behavior between an event recognizer and a run-time checker for the sake of simplicity in the analysis.

78

Instrumented Target Program   **Monitor**

Execution of Original Target Program

Execution for Monitoring
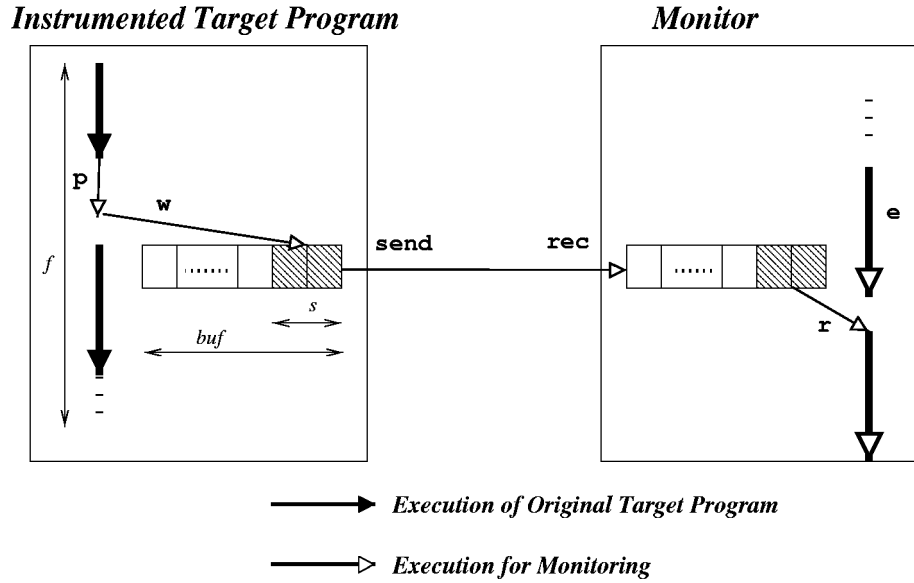
Figure 7.2: Model of the overhead to the target system

- w is the overhead of writing a snapshot to the buffer.

- *send* is the overhead of sending the content of the buffer at the filter to the monitor when the buffer becomes full.

- *rec* is the overhead of receiving snapshots into the buffer at the monitor.

- r is the time taken for a monitor to read a snapshot from the buffer.

- e is the overhead of evaluating properties upon arrival of a snapshot by the monitor.

We will use $P, W, Send, Rec, R$, and E as sums of $p, w, send, rec, r$, and e, respectively, over the whole target program execution. We will concentrate our discussion to the overhead reduction techniques applied to a filter. There are three factors we can control to reduce the overheads at the filter - a snapshot size s, a frequency of taking snapshots f, and a buffer size buf. The effects of these three factors on the overhead parameters are as follows.

1. Reducing snapshot size s decreases the writing overhead W, the reading overhead R, and the overheads of sending and receiving operations $Send$ and Rec.

2. Reducing snapshot frequency f decreases

   - the overhead of executing probes P and writing snapshots W.

   - the amount of snapshots delivered from the target program to the monitor, which reduces sending overhead $Send$ and receiving overhead Rec.

   - the overhead of reading snapshots R and evaluating properties by the monitor E.

79

3. Increasing a buffer size *buf* decreases the sending overhead *Send* and overhead *Rec* if send/receive operations over the communication channel have high per-operation overhead.

Simply decreasing s and f may result in incorrect evaluation of properties. In the extreme case, we do not take any snapshot from the target program which makes f as 0. Obviously, this way of decreasing f is useless because monitoring results in nothing. Therefore, we have to come up with reduction techniques which decrease s or f while they conserve the evaluation of properties correctly. Increasing *buf*, however, does not affect the correctness of the property evaluations although increased *buf* delays the delivery of snapshots which may result in late detection of violations. We will discuss details for techniques of reducing s and f which conserve the results of property evaluations correctly. Then, we will describe a result of experiment measuring the overhead using these techniques.

7.2 Reducing Snapshot Size

A snapshot sent from a filter to an event recognizer consists of an ID of the monitored variable (1 byte), the value of the variable (from one byte (`boolean` or `byte` type) to eight bytes (`long` or `double` type)), and a timestamp (8 byte `long` type).

7.2.1 Reducing the Value Field in a Snapshot

We observe that a difference between two consecutive values of a numerical variable is usually small. Thus, just sending the difference in smaller representation can save the amount of snapshot taken by the value of a variable. We call this reduced representation of a value as a *delta value* and this reduction technique as *delta abstraction*. The type of a delta value for `long` is `int`. The type for `int` and `short` is `byte`. The type for `double` is `float`. To distinguish a delta value from a normal value, negative ID is used. For example, suppose `monobj long A.x` has id 10. A snapshot containing the delta value of this variable has id -10 and type `int`.

A probe should test whether the difference can be represented in a delta type. If the difference is small enough to be represented in a delta type, the probe sends a delta value with negative ID. Otherwise, a regular value and an ID is exported. This testing may cause extra overhead at run-time. Section 7.4 shows the experimental results on the overhead when delta abstraction is performed.[3]

7.2.2 Reducing Timestamps

If we attach a timestamp (8 byte `long` type) to every snapshot, the timestamp would take a large portion (47%[4] to 80%[5]) of the total space taken for the snapshot. To reduce the snapshot size, we can send a timestamp periodically instead of attaching a timestamp

---

[3]The worst case is that all snapshots need to be exported after being testing. Experimental results shows that even in the worst case, however, the overhead is increased less than 1%.

[4]$47 = 8/(1+8+8)$

[5]$80 = 8/(1+1+8)$

to every snapshot. This method can reduce the size of snapshots when events happen frequently.

The timestamping error caused by periodic timestamps is as follows. Suppose that a period is p. Let us use $time_p(e)$ to indicate a periodic timestamp of an event e and $time(e)$ to indicate the exact timestamp of e. Suppose e happens between time instant ip and $(i + 1)p$ where $i \in N$. The minimum timestamping error caused by periodic timestamps is 0 when e happens exactly at the instant ip. I.e., $time(e) - time_p(e) = ip - ip = 0$. The maximum error caused by periodic timestamps is p when e happens near to the instant $(i + 1)p$. I.e., $time(e1) - time_p(e1) < (i + 1)p - p = p$. For measuring time difference between two events $e1$ and $e2$, there are two cases depending on whether $e1$ and $e2$ are in the same interval or not as depicted in Figure 7.3.
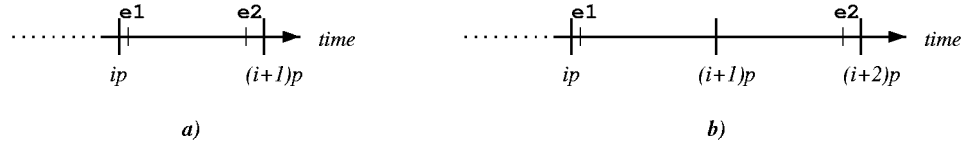
Figure 7.3: Maximum error caused by periodic timestamp

- In Figure 7.3.a), events $e1$ and $e2$ have the same periodic timestamp ip because these two events occur in the same interval between ip and $(i + 1)p$. The minimum error for time difference caused by periodic timestamps is 0 when $e1$ and $e2$ happen at the same instant. I.e., $(time(e2) - time(e1)) - (time_p(e2) - time_p(e1)) = 0 - 0 = 0$. The maximum error for time difference caused by periodic timestamps is p when $e1$ happens at the time instant ip and $e2$ happens near to the time instant $(i + 1)p$. I.e., $(time(e2) - time(e1)) - (time_p(e2) - time_p(e1)) = p - 0 = p$.

- In Figure 7.3.b), $time_p(e1) = ip$ and $time_p(e2) = (i + 1)p$. Thus, $time_p(e2) - time_p(e1) = p$. The minimum error for time difference caused by periodic timestamps is 0 when $e2$ happens exactly p time unit later that $e1$. I.e., $(time(e2) - time(e1)) - (time_p(e2) - time_p(e1)) = p - p = 0$. The maximum error for time difference caused by periodic timestamps is p when $e1$ happens at the time instant ip and $e2$ happens near to the time instant $(i + 2)p$. $(time(e2) - time(e1)) - (time_p(e2) - time_p(e1)) = 2p - p = p$.

Therefore, a timestamping error caused by periodic timestamp is less than the period p.

Periodic timestamp contains only an ID (1 byte) indicating it is a timestamp (see Appendix A). When snapshots are frequently sent, saving due to periodic timestamp is large. Suppose p is 100 ms and a snapshot is sent every milisecond. With timestamp attached to every snapshot, each second, timestamp takes 8000 bytes. Periodic timestamps, however, take only 10 bytes. When snapshots are infrequent, periodic timestamp might take more space in snapshots. Java-MaC provides timestamping using a separated thread which sends a timestamp after sleep for p time unit repeatedly.

7.3 Reducing the Frequency of Taking Snapshot

We clarify two terms first. To *extract* a snapshot means that a probe reads the snapshot from the execution of the target system. To *export* a snapshot means that a probe sends the snapshot it reads to the monitor. A probe extracts a snapshot but does not necessarily export the snapshot.

7.3.1 Decreasing the Frequency of Snapshot *extraction*

We restrict the scope of monitoring using `in` keyword in the declaration section in a PEDL script. An observation is that not every update of a variable or every invocation of a method should be monitored for checking properties. Update and invocation in specific context may count for checking properties. This context is expressed by `<constraint>` in the following extended declaration section in PEDL grammar.

```
MonScr <...>
...
/* Monitored entity declaration section */
monobj <type> <var_name> <constraint>;
monmeth <type> <metd_name> <constraint>;
...
End
```

where

```
<constraint> := in <scope> | not in <scope>
<scope>      := <pkg> | <pkg>.<class> | <pkg>.<class>.<method>
```

The `<constraint>` indicates the scope of the target system to observe. The scope can be described using *package*, *class*, and *method* of the target program. A probe is inserted into the part of target system specified by `<constraint>`. A following bank account example shows the usage of `<constraint>`.

```
class Account {
    private double balance;
    ...
    void deposit(double amount) { balance += amount; }
    double withdrawal(double amount) { balance -= amount; }
}
class WireTransfer {
    ...
}
class ATM {
    ...
}
```

- If we want to monitor the amount of money deposited via wire transfer only, we do not always have to monitor `deposit()`. We only need to monitor `deposit()` invoked inside `WireTransfer` class. The constraint

```
monmeth account.deposit(double) in WireTransfer
```

will remove the unnecessary monitoring such as monitoring `deposit()` method invoked in `ATM`.

- Every month, an account statement comes to customers notifying them of the maximum balance for a month. To monitor that amount, we need to monitor `balance` only when updated by the `deposit()` method, not when updated by the `withdrawal()`. This specification is written in the constraint

```
monobj double balance in account.deposit(double)
```

  If a customer withdraws money from ATM often and deposits his payroll once per month, this constraint reduce the overhead significantly.

Note that `<constraint>` is a static constraint, not a dynamic constraint which needs to be evaluated at run-time. This technique of restricting scope to monitor applies when the target program is instrumented, which does not cause run-time overhead.

7.3.2   Decreasing the Frequency of Snapshot *exportation*

An observation is made that not every update of a variable affects properties being checked. Only an update of a variable which changes at least one condition or an event can affect the property. Let us see a following simple example.

```
condition c1 = x <= 10;
...
property safe = !c1 && c2 && ...
```

Suppose `x` is used only in `c1` and has value 1 initially. Unless `x` is changed to be greater than 10, an update of `x` does not change the value of property `safe`. Thus, updates of `x` do not need to be exported except when `x` becomes greater than 10. Only the last update of `x` in Figure 7.4 should be exported.
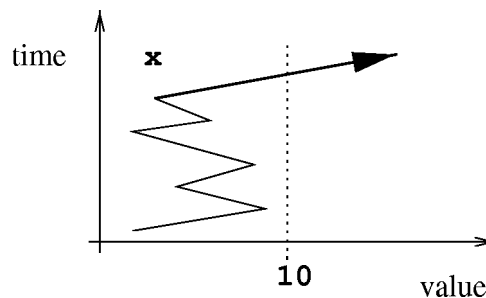
Figure 7.4: The last update changes `c1`

We can reduce the frequency of exporting snapshots by exporting only snapshots which contain updates of variables affecting properties as we have discussed in Section 3.3. We call this technique *value abstraction*. In one extreme end, all requirement properties need to be evaluated at the target program to see whether an update affects any of these

properties. In this extreme case, the overhead at the target program does not decrease because the overhead of evaluating all properties is added to the overhead at the target program although communication overhead decreases. We need to define a set of boolean expressions over the set of monitored variable exp_{V_m} from given a PEDL script satisfying the following two conditions.

1. if an update of a monitored variable does not change the value of any boolean expression in exp_{V_m}, then the update does not change the value of any requirement property either as defined in Definition 12

2. boolean expressions in exp_{V_m} are simple enough that evaluation of exp_{V_m} requires only small amount of computation.

We define such set of boolean expressions heuristically as follows.

Definition 16 (Simple Expression) *A simple expression $sexp_x$ for a monitored variable x is defined as one of the following two forms of boolean expressions*

$$x \; cmp \; c \; or \; c \; cmp \; x$$

where x is a monitored variable, cmp is one of $>, >=, ==, <=, <$, and c is a constant.

$sexp_x$'s are obtained from the event/condition definitions of a PEDL script. Java-MaC does not apply value abstraction for a monitored variable x if any of the following two conditions holds.

- a PEDL script has boolean expressions containing a monitored variable x which are not simple expressions. An update which does not change the value of any $sexp_x$'s can still change the value of requirement properties in this case.

- a PEDL script has `update(x)`. An update which does not change the value of any $sexp_x$'s can still change the value of requirement properties by raising an event `update(x)`.

Otherwise, Java-MaC performs the value abstraction to the variable x. The following example illustrates how simple expressions are obtained from condition definitions.

```
condition c1 = (3 < x && x < 10) || y >10 || z > 10;
condition c2 = x> 5 && w > 2*z + 3;
```

From the above definitions, following five simple expressions are obtained.

$$sexp_{x0} = 3 < x$$
$$sexp_{x1} = x < 10$$
$$sexp_{y0} = y > 10$$
$$sexp_{z0} = z > 10$$
$$sexp_{x2} = x > 5$$

84

x is contained only in $sexp_{x0}$, $sexp_{x1}$, and $sexp_{x2}$ in the PEDL script. Whenever x is updated, a probe checks whether any of the value of $sexp_{xi}$ has changed. If the value of any $sexp_{xi}$ is changed, the probe exports the new value of x to the event recognizer. Otherwise, the probe does not. Situation is similar for y. z and w are, however, different because z and w are contained in the expression $w > 2 * z + 3$ which is not a simple expression. Therefore, all snapshots updating z and w are exported. Figure 7.5 shows the sample execution of the target program and what snapshots are exported and what snapshots are not exported in the execution by value abstraction.
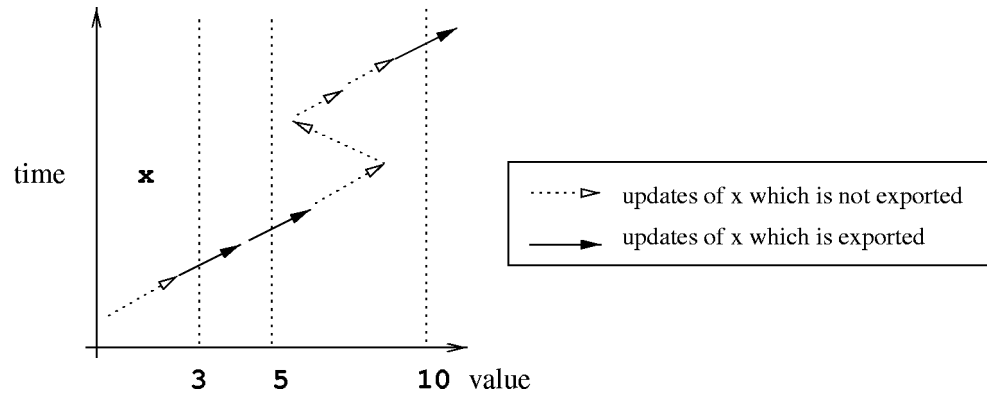
Figure 7.5: Example of updating x

The best overhead reduction is achieved when there is only one $sexp_x$ for a monitored variable x and the $sexp_x$ does not change at all. Then, a probe does not send a snapshot at all. In the worst case when $sexp_{xi}$ does not change for all $i < n$ and $sexp_{xn}$ changes where n is the total number of $sexp_x$, the value abstraction does not reduce the overhead, but increases the overhead due to the overhead of evaluating $sexp_{xi}$'s.

7.4 Overhead Measurement

In this section, we measure the overheads incurred to the target program by Java-MaC. First, we describe the experiment we design to measure the overheads. Second, we measure the effects of overhead reduction techniques to the overhead incurred by probe, communication, and event/condition evaluation. Third, we illustrate the overhead incurred to the Sieve of Eratosthenes by Java-MaC as an example.

7.4.1 Description of Overhead Measurement Experiment

TestMain in Figure 7.6 invokes a nested loop. The outer loop from line 14 to line 17 iterates maxIter times and has a statement assigning a monitored variable TestMain.t.monitoredX at line 16. The inner loop at line 15 iterates maxInst times.

The inner loop at line 15 is compiled into the four bytecode instructions in Fig 7.7. iinc j 1 increases a local variable j by 1. iload j loads the value of j onto the operand stack. Similar for iload maxInst. if_icmplt Label2 compares two top stack operands and jumps to Label2 if a top operand is less than the second top operand.

85

```
01:class TestMain {
02:     public Test t;
03:     TestMain() { t = new Test();}
04:     public static void main(String[] args) {
05:         int maxIter = Integer.parseInt(args[0]);
06:         int maxInst = Integer.parseInt(args[1]);
07:         (new TestMain()).t.testing(maxIter,maxInst);
08:     }
09:}
10:
11:class Test{
12:     int monitoredX;
13:     public void testing(int maxIter, int maxInst) {
14:         for(int i=0; i < maxIter; i++) {
15:             for(int j=0; j < maxInst; j++) {}
16:             monitoredX = i;
17:         }
18:     }
19:}
```

Figure 7.6: A test program for measuring overhead cost

```
Label2:
    iinc j 1
Label3:
    iload j
    iload maxInst
    if_icmplt Label2
```

Figure 7.7: 4 bytecode instructions of line 15 of Figure 7.6

`int TestMain.t.monitoredX` is the monitored variable. A snapshot is taken at line 16 after the inner loop finishes - that is, after the execution of 4×`maxInst` bytecode instructions. The total number of snapshots taken in the whole execution is `maxIter`. By increasing `maxInst`, we can decrease the frequency of taking snapshots.

We performed experiments on the following platforms. `TestMain.class` runs on a Linux 2.2 machine (2X 550Mhz PIII, 1GB memory). An event recognizer and a run-time checker run on the same Windows 2000 machine (1.4Ghz Pentium4, 512MB memory). Both machines are on the same network domain. Messages are delivered using a TCP socket. The size of a communication buffer used by Java-MaC is 512 bytes. First, we executed `TestMain.class` with `maxIter` as 10^6 and `maxInst` as 25 to 2.5×10^5 and measured the execution time. Figure 7.8 shows the execution time of uninstrumented `TestMain.class`. The x axis indicates `instMax`. The y axis is the execution time in logarithmic scale. The execution time increases in linear to `instMax`.
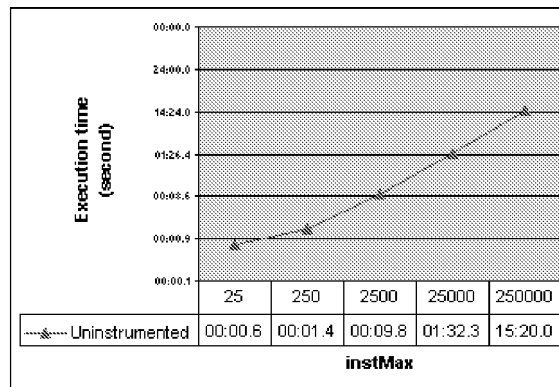
Figure 7.8: Execution time of `TestMain.class`

Then, we instrumented `TestMain.class` according to different probe configurations and measured the execution time. We repeated the experiment 10 times for a fixed `instMax`. We measure the monitoring overhead with and without applying overhead reduction techniques. We performed experiments with the following three different probe configurations.

- "NoAbstract" does not use overhead reduction techniques. NoAbstract does not perform timestamping nor locking/unlocking. A probe sends a snapshot of `TestMain.t.monitoredX` as 5 bytes (ID takes 1 byte and a value takes 4 bytes) after executing 4 × `maxInst` bytecode instructions.

- "DeltaAbstract" uses delta abstraction and sends a snapshot of `TestMain.t.monitoredX` in two bytes (ID takes 1 byte and a delta value takes 1 byte).

- "ValueAbstract" applies value abstraction. ValueAbstract has only a single *sexp* which is `TestMain.t.monitoredX` < `-10`. A probe does not send a snapshot except the first one because a new value of `int TestMain.t.monitoredX` is always greater than 0.

87

7.4.2 Overhead of Probe

We measured the execution time of instrumented `TestMain.class` with the different probe configurations without writing snapshots into the communication buffer and sending the content of the buffer. In other words, this experiment measures P only. In addition to the three probe configurations we described in Section 7.4.1, we used the following probe configurations.

- "Time" is same as "NoAbstract" except it generates timestamps every 20 ms.[6]

- "Thread" is same as "NoAbstract" except it additionally performs locking/unlocking for each probe execution as we described in Section 6.2.1.

- "ValueAbstract$n$" evaluates n *sexp*'s where $n \in \{50, 100, 150, 200\}$. Corresponding configurations are indicated by "ValueAbstract50", "ValueAbstract100" "ValueAbstract150", and "ValueAbstract200" respectively.

Figure 7.9.a) shows the overhead ratio of Java-Mac. The top four lines are ValueAbstract200, ValueAbstract150, ValueAbstract100, and ValueAbstract50 in order. ValueAbstract200 slows down the execution around 28 times when the frequency of taking snapshot is once per 100 bytecode instructions execution. The configurations which do not evaluate *sexp*'s slow down the execution less than 10 times at the frequency of 1/100. The overheads of configurations except the configurations evaluating more than 50 *sexp* become less than 10% at the frequency of $1/10^5$. Figure 7.9.b) shows the overhead per a single snapshot. The top four lines are ValueAbstract200, ValueAbstract150, ValueAbstract100, and ValueAbstract50 in order. The time taken to execute a single ValueAbstract200 probe is around 16 microseconds and the time taken to execute a single ValueAbstract100 probe is around 10 microseconds. Thus, it takes around 60 nanosecond to evaluate 1 *sexp*. The other probes take almost the same overhead of 5 microsecond. The extra overheads caused by delta abstraction, timestamping, and locking/unlocking are not significant.

This experiment alone does not show the advantages of the delta abstraction and value abstraction, but shows only the extra overhead of evaluating *sexp*'s. The advantage, however, will be shown when the communication and event/condition evaluation overhead are measured in Section 7.4.3 and Section 7.4.4. In addition, in realistic setting, one monitored variable has a small number of *sexp*'s, not hundreds of *sexp*'s to evaluate. Furthermore, not all *sexp*'s need to be evaluated because a probe stops evaluating *exps*'s whenever it finds affected *exps*. Thus, the overhead by value abstraction shown in Figure 7.9 indicates the overhead of the worst case.

7.4.3 Overhead of Probe and Communication

We measured the execution time of instrumented `TestMain.class` with the three different probe configurations without condition/event evaluation by using a dummy event recognizer which just receives snapshots but does not evaluate conditions/events. In other words, this experiment measures $P + W + Send$.

---

[6]Java-MaC using the JVM on the target platform does not make *accurate* periodic timestamp if a period is smaller than 20 ms.

a)

	1/100	1/1000	1/10000	1/100000
ValueAbstract	9.59	4.63	1.54	1.06
DeltaAbstract	9.15	4.44	1.53	1.06
NoAbstract	9.22	4.46	1.52	1.06
TimeStamp	9.63	4.75	1.61	1.06
Thread	9.59	4.63	1.55	1.09
ValueAbstract50	14.05	6.66	1.87	1.09
ValueAbstract100	18.11	8.29	2.11	1.13
ValueAbstract150	23.79	10.65	2.41	1.15
ValueAbstract200	27.91	12.39	2.68	1.19

Frequency of taking snapshots

b)

	1/100	1/1000	1/10000	1/100000
ValueAbstract	5.2	5.2	5.3	5.7
DeltaAbstract	4.9	4.9	5.2	5.4
NoAbstract	5.0	4.9	5.1	5.8
TimeStamp	5.2	5.4	5.4	5.4
Thread	5.2	5.2	5.3	6.5
ValueAbstract50	7.9	8.1	8.5	8.8
ValueAbstract100	10.4	10.4	10.9	11.9
ValueAbstract150	13.8	13.8	13.8	14.2
ValueAbstract200	16.3	16.3	16.4	17.7

Frequency of taking snapshots

Figure 7.9: Overhead of probe. (a) Overhead ratio (b) Overhead per a single snapshot

Figure 7.10.a) shows the overhead ratio. The configuration without overhead reduction techniques slows down the execution 13 times when the frequency of taking snapshots is 1/100. Delta abstraction, however, slows down the execution 11 times and the value abstraction slows down the execution less than 10 times at the frequency of 1/100. Compared to the overhead of NoAbstract without communication (see Figure 7.9), the overhead of NoAbstract increases around 30% due to the communication overhead. The overhead of ValueAbstract almost does not increase from Figure 7.9 because ValueAbstract does not send snapshots because snapshots do not affect the property TestMain.t.monitoredX < -10 (TestMain.t.monitoredX is always greater than or equal to 0. See Figure 7.6). Figure 7.10.b) shows the overhead per a single snapshot. The overhead per a single snapshot is almost constant over the frequency of taking snapshots. While NoAbstract takes around 7 micro second per a single snapshot, DeltaAbstract takes around 6 micro second and ValueAbstract takes around 5.2 micro second.

7.4.4 Overhead of Probe, Communication, and Evaluation

We measured the execution time of instrumented TestMain.class with the three different probe configurations over four different condition expressions of different lengths. The condition/event evaluation takes time in linear to the length of an expression as we have seen in Section 5.2.2. We measure the length of an expression as a number of binary operators in the expression. Figure 7.11 has a condition expression c of length 1 and Figure 7.12 has a condition expression c of length 50.

a)

b)
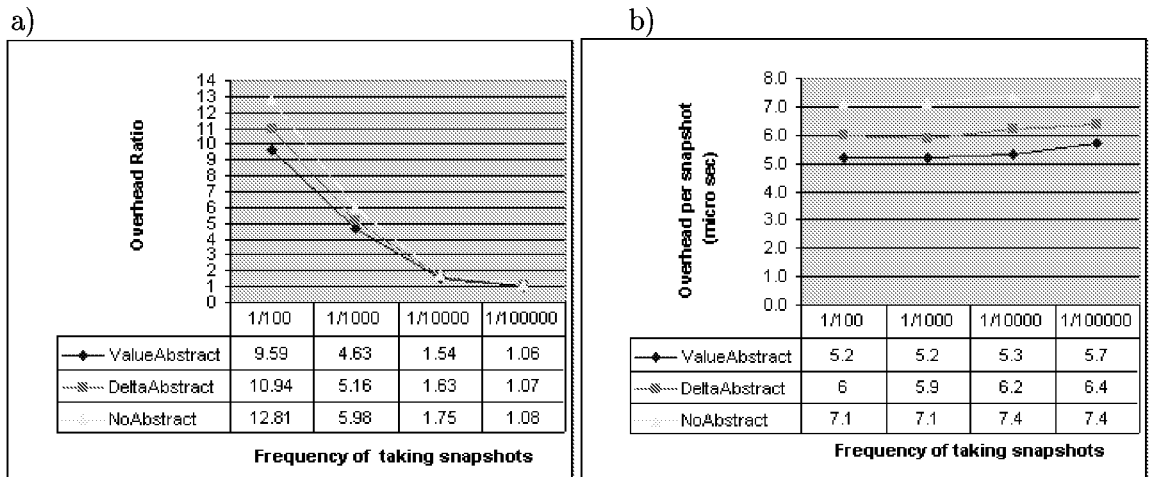
Figure 7.10: Overhead of probe and communication. (a) Overhead ratio (b) Overhead per a single snapshot

```
MonScr
    export condition c;
    monobj int TestMain.t.monitoredX;
    condition c = TestMain.t.monitoredX > 0;
end
```

Figure 7.11: A condition expression of length 1

```
MonScr
    export condition c;
    monobj int TestMain.t.monitoredX;
    condition c =
    (((((((((((((((((((((((((((((((((((((((((((((((((
    TestMain.t.monitoredX
    +1)*1)+1)*1)+1)*1)+1)*1)+1)*1)          // 10
    +1)*1)+1)*1)+1)*1)+1)*1)+1)*1)          // 20
    +1)*1)+1)*1)+1)*1)+1)*1)+1)*1)          // 30
    +1)*1)+1)*1)+1)*1)+1)*1)+1)*1)          // 40
    +1)*1)+1)*1)+1)*1)+1)*1)+1)             // 49
    > 0;                                    // 50
end
```

Figure 7.12: A condition expression of length 50

90

Figure 7.10.a) shows the overhead ratio over the four different expressions with three different probe configurations. NoAbstract indicates the overhead of evaluating Figure 7.11 without any reduction techniques. NoAbstract50 indicates the overhead of evaluating Figure 7.12. NoAbstract100 and NoAbstract150 indicate the overheads of evaluating conditions of length 100 and 150 respectively. Similarly, DeltaAbstract$n$ and ValueAbstract$n$ indicate the overhead of evaluating a condition expression of length n. The top two lines indicate NoAbstract150 and DeltaAbstract150. Similarly, the second top two lines indicate NoAbstract100 and DeltaAbstract100, and the third two lines indicate NoAbstract50 and DeltaAbstract50. The top line (bright color) of NoAbstract150 slows down the execution around 609 times when the frequency of snapshot is 1/100. The second top line (dark color) of DeltaAbstract150 shows that the overhead slows down the execution around 583 times at the frequency of 1/100. Delta abstraction does not reduce the overhead significantly, because delta abstraction does not reduce the condition/event evaluation overhead which is the bottleneck in this experiment. Value abstraction, however, significantly reduces the overhead. The bottomline in Figure 7.13.a) indicates the overhead of ValueAbstract$n$ which slow down the execution 10 times at the frequency of 1/100. The overhead of ValueAbstract$n$ does not increase as the length of the condition is increased.

This large overhead of evaluation is due to the object-oriented implementation of PEDL AST tree (`pedl.out`). The early prototype of Java-MaC aimed to monitor a whole object, which was discarded later for the reasons explained in Section 5.2.1. Thus, `pedl.out` can contain general operators working on objects and objects as operands. For this generality, primitive values are represented as objects in `pedl.out`. For example, a wrapper object `Integer(10)` is stored for an integer 10 in `pedl.out`. To make comparison between an object `Integer(10)` and `int x`, we need to go through several complex steps including identifying the type of the object and unwrapping the object, etc. Re-implementation of `pedl.out` removing object-orientedness can decrease the evaluation overhead significantly.

Figure 7.13.b) shows the overhead per a single snapshot.[7] NoAbstract150 (the top line) takes around 360 microsecond per a single snapshot at the frequency from 1/100 to 1/10000. Considering that the evaluation is the bottleneck in this experiment, we can know that it takes around 360 microsecond for the event recognizer to evaluate a condition expression of length 150. The time taken to evaluate a condition of length 150 (NoAbstract150) is 367 microsecond (NoAbstract150) and the time taken to evaluate a condition of length 50 (NoAbstract50) is around 124 microsecond at the frequency 1/100. Thus, it takes around 2.4 microsecond to evaluate an expression of length 1.[8] NoAbstract150 takes 274 microsecond per a single snapshot at the frequency of 10^{-5}. The number 274 microsecond can be understood as time taken to evaluate a condition of length 150 after evaluating the condition during the interval between two consecutive snapshot.[9] NoAbstract150 takes 20.9 microsecond per a single snapshot at the frequency of 10^{-6}. The reason the overhead per a single snapshot becomes so small at the frequency of 10^{-6} is that the period between two consecutive snapshots at the frequency of 10^{-6} is long enough for the event recognizer

---

[7]The standard deviation of execution time at the frequency of 10^{-6} is greater than the measured overhead caused by Java-MaC. Thus, the number of overhead per a single snapshot itself is not much meaningful.

[8]This 2.4 microsecond does not contain the initialization cost for each evaluation.

[9]Figure 7.8 shows that the interval between two consecutive snapshots at the frequency of 10^{-5} is around 92 microsecond because 10^6 snapshots are taken during 92 second when `instMax` is 25000. $367 - 92 \approx 274$

to evaluate the condition of length 150 completely.[10] ValueAbstract$n$ (the bottom line) takes around 5.2 microsecond per a single snapshot over evaluating conditions of various lengths.

a)   b)
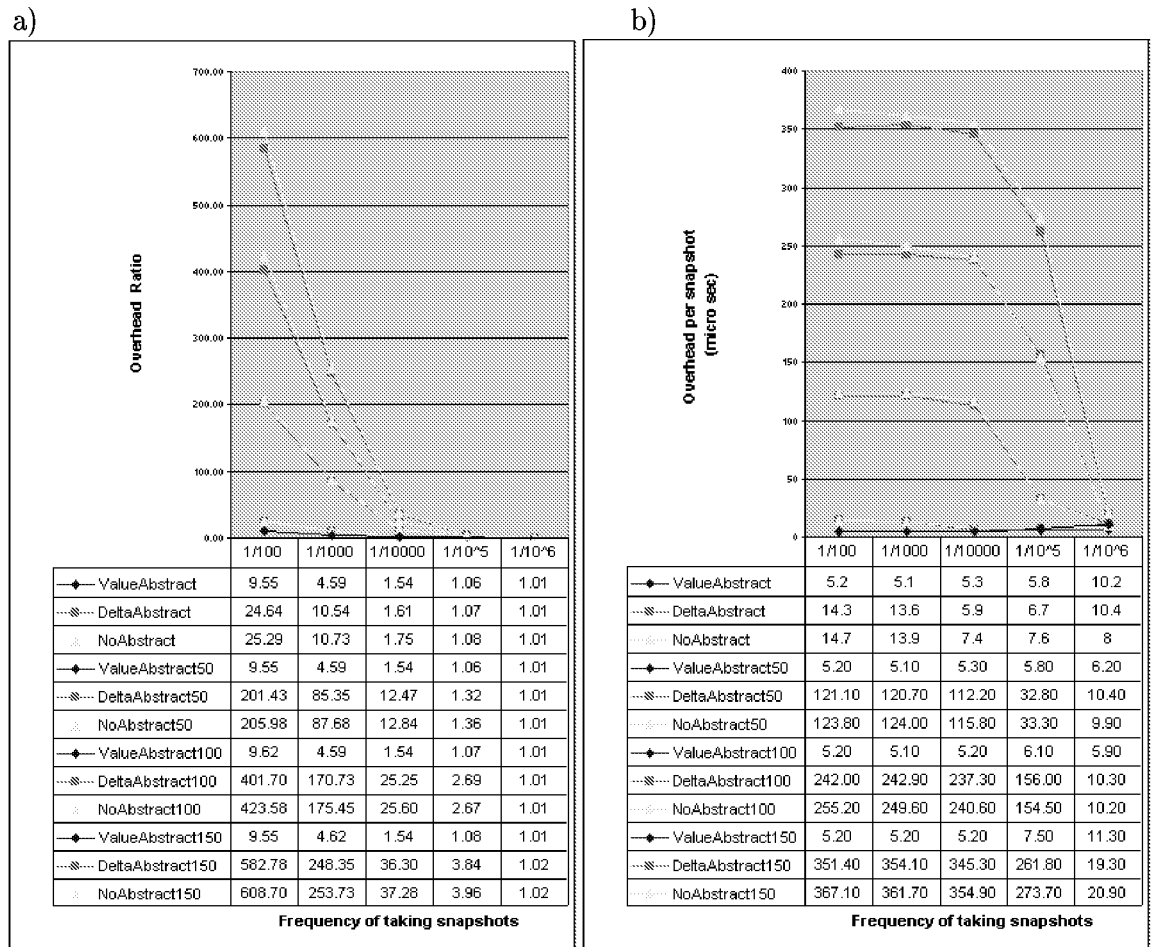
Figure 7.13: Overhead of evaluating a condition expression of different size (a) Overhead ratio (b) Overhead per a single snapshot

7.4.5 Example: the Sieve of Eratosthenes

We illustrate the monitoring overhead and effects of overhead reduction techniques using the Sieve of Eratosthenes program. The Sieve of Eratosthenes generates prime numbers. The algorithm of the Sieve for generating prime numbers less than or equal to n can be described as follows.

Make a list of all the integers less than or equal to n (and greater than one). Strike out the multiples of all primes less than or equal to $\sqrt{n}$, then the numbers

---

[10]Figure 7.8 shows that the interval between two consecutive snapshots at the frequency of 10^{-6} is 920 microsecond.

that are left are the primes.[11]

Figure 7.14 shows a Java code for the Sieve. An integer being tested is declared as numTested in line 14. numPrimes declared in line 15 indicates the total number of prime numbers upto numTested. Main code in execute() from line 22 to line 42 contains a nested loop. Lines 30 to 41 form an outer loop which increases numTested one by one. Lines 33 to 35 make an inner loop which divides numTested with prime numbers less than or equal to $\sqrt{\text{numTested}}$. Lines 37 to 40 store numTested as a prime number and increase numPrimes by 1 if there is no prime number which can divide numTested.

We would like to monitor and check whether there exists a prime number between 99990 and 100000. For that purpose, an event foundPrime is defined in lines 5 to 7 of Figure 7.15 The experiment shows that there exists one prime between 99990 and 100000.

The computational complexity of the Sieve program(Figure 7.14) is $O(n\sqrt{n})$ (the outer loop takes $O(n)$ and the inner loop takes $O(\sqrt{n})$) where n is the maximum number to check. The bottom line of Figure 7.16.a) shows the execution time of the uninstrumented Sieve program. Testing integers from 1 to 200000 takes 2.3 seconds. Testing integers from 1 to 800000, however, takes 22.6 seconds. Java-MaC takes around $1.08 \times n$ snapshots including n numTested's and $0.08 \times n$ numPrimes's. [12] The frequency of taking snapshots decreases as n increases because of $O(n\sqrt{n})$ computational complexity of the Sieve program, which decreases the overhead ratio by Java-MaC as n increases . Figure 7.16.b) shows the overhead ratios of NoAbstract, DeltaAbstract, and ValueAbstract. NoAbstract slows down the Sieve program 3.1 times when n is 200000. The overhead ratio decreases to 1.5 times when n increases to 800000. DeltaAbstract reduces the overhead 19% compared to the overhead of NoAbstract when n is 200000. ValueAbstract reduces the overhead 73% compared to the overhead of NoAbstract when n is 200000 by not sending snapshots of numTested except three snapshots containing 3, 99990, and 100001. All snapshots containing numPrimes are sent because numPrimes is involved in update(). As n increases, the bottleneck of event evaluation diminishes, which decrease the amount of reduction by DeltaAbstract and ValueAbstract.

---

[11]If there exists a prime greater than $\sqrt{n}$ which divides n, there exists a prime less than or equal to $\sqrt{n}$ which divides n, too.

[12]The number of primes less than 800000 is around 64000.

```
01:public class SieveMain{
02:    Sieve sa;
03:    SieveMain() { sa = new Sieve();}
04:    public static void main(String[] args) {
05:        SieveMain sm = new SieveMain();
06:        sm.sa.initialize( Integer.parseInt(args[0]));
07:        sm.sa.execute();
08:    }
09:}
10:class Sieve {
11:    public Sieve sa;
12:    public int primes[];
13:    public int maxCandidate;
14:    public int numTested;      // current number being tested
15:    public int numPrimes;      // number of primes found
16:
17:    public void initialize(int i) {
18:        maxCandidate = i;
19:        primes = new int[maxCandidate];
20:    }
21:
22:    public void execute() {
23:        int k = 0;
24:        int sqrt_i=0;
25:        boolean flag = false;
26:
27:        primes[0] = 1;
28:        primes[1] = 2;
29:        numPrimes = 2;
30:        for (numTested = 3; numTested <= maxCandidate; numTested++) {
31:            k = 1;
32:            sqrt_i = (int)(Math.sqrt(i));
33:            for (flag = true; k < numPrimes && flag; k++)
34:                if ( primes[k] <= sqrt_i && numTested % primes[k] == 0)
35:                    flag = false;
36:
37:            if (flag) {
38:                numPrimes++;
39:                primes[numPrimes - 1] = numTested;
40:            }
41:        }
42:    }
43:}
```

Figure 7.14: The code of the Sieve of Eratosthenes

```
01:MonScr
02:    export event foundPrime;
03:    monobj int SieveMain.sa.numTested;
04:    monobj int SieveMain.sa.numPrimes;
05:    event foundPrime = update(SieveMain.sa.numPrimes) when
06:                            (99990 <= SieveMain.sa.numTested
07:                            && SieveMain.sa.numTested <= 100000);
08:end
```

Figure 7.15: PEDL script for checking the existence of prime between 99990 and 100000

a)

	2E+05	4E+05	6E+05	8E+05
——— ValueAbstract	00:03.6	00:09.2	00:17.4	00:29.0
——— DeltaAbstract	00:06.2	00:12.9	00:21.3	00:32.4
NoAbstract	00:07.1	00:13.8	00:22.2	00:33.6
Uninstrumented	00:02.3	00:06.8	00:13.3	00:22.6

maxCandidate

b)

	2E+05	4E+05	6E+05	8E+05
——— ValueAbstract	1.58	1.36	1.31	1.28
——— DeltaAbstract	2.74	1.91	1.60	1.44
NoAbstract	3.14	2.03	1.66	1.49

maxCandidate
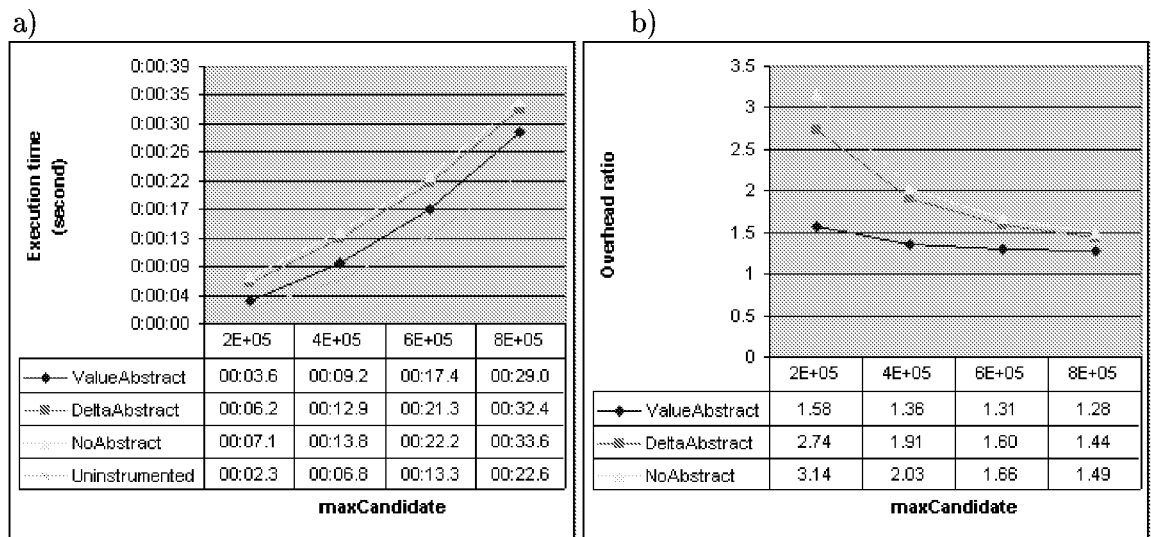
Figure 7.16: Overhead of Java-MaC to the Sieve of Eratosthenes (a) Execution time (b) Overhead ratio

Chapter 8

Examples

This chapter demonstrates the application of Java-MaC to several examples. First, we will apply Java-MaC to two small but illustrative examples - a railroad crossing system and a database client system. Second, we will apply Java-MaC to the emulator of distributed controller for large numbers of mobile agents such as micro-air vehicles. Third case study is on the network routing algorithm. This case study shows the flexibility of the MaC architecture by plugging a NS2 simulator in the MaC architecture. Finally, we monitor and check the execution of an inverted pendulum simulator written in the Charon specification language. We measure the overhead caused by Java-MaC to the inverted pendulum.

8.1 Small Examples

8.1.1 A Railroad Crossing

A railroad crossing system has a train, a gate, and a controller. The goal of the system is operating the crossing gate subject to the safety property. The safety property states that when a train is in the crossing, the gate must be completely down. This example is commonly used as a benchmark in formal methods research [HD96]. The railroad system is illustrated in Figure 8.1.

A PEDL script for the railroad crossing is given in Figure 8.2. The script starts with a list of exported events `startGD` and `endGD` in line 2 and an exported condition declaration containing a condition `IC` in line 3. Then, four variables are declared as monitored variables including `RRC.train_x`, `RRC.train_length`, `RRC.cross_x`, and `RRC.cross_length` from line 5 to line 8. `train_x` is the position of the tail of a train. `train_length` is the length of a train. `cross_x` is the position of the left end of the crossing. In addition, there are two monitored methods declared in lines 10 and 11: `void Gate.gd(int)` and `int Gate.gu()`. A method `void Gate.gd(int)` is used to lower the gate. `int Gate.gu()` is the method for raising the gate. Next, a condition `IC`, which indicates whether a part of the train is in the crossing, is defined in lines 13 and 14. `IC` is true if and only if the head of the train is after the left end of the crossing and the end of the train should be before the right end of the crossing. Finally, an event `startGD` meaning that the gate enters the closed state is defined as the ending of method `Gate.gd(int)` in line 16. An event `endGD` meaning that the gate starts raising is defined as the starting of the method `Gate.gu()` in line 17.

```
        RRC.cross_x      RRC.cross_x
                       + RRC.cross_length
```

Train

Gate

Controller

```
RRC.train_x   RRC.train_x
            + RRC.train_length
```
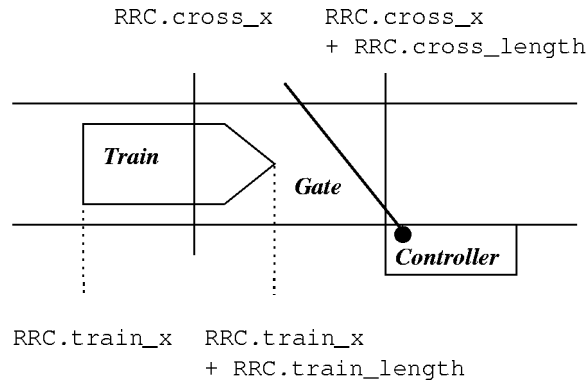
Figure 8.1: A railroad crossing example

A MEDL script for the railroad crossing is given in Figure 8.3. The MEDL script starts with a list of imported events in line 2 and an imported condition list in line 3. A condition meaning that the gate is down is defined as an interval between startGD and endGD in line 5. The safety property indicates that when a train is in crossing (IC), the gate must be down (GD). The safety property is defined in line 7.

8.1.2 A Database Client

A database client queries a database server periodically (choosing it randomly from a list of servers). The pseudo-code for the client appears in Figure 8.4.

The real-time requirement and fault tolerant requirements are as follows:

- *Real-time requirement.* The client is periodic; that is, every few (say 1000 ms) seconds it tries to query a new server.

- *Fault tolerant requirement.* Old data is used only when either the client fails to connect to some server after sufficient number (say three) retries or the client fails to get a response from the server (for the query asked) after trying (say) four times.

A MEDL script describing these requirements is given in Figure 8.5. The requirements for the client can be defined with a signal for the beginning of the computation (startPgm), an event for when a fresh period of 1000 ms has started (periodStart), a signal when the client fails to connect to a server (conFail), a signal when the client resends the query (queryResend), and an event denoting when the client uses old information (oldDataUsed). These events are declared as imported events in line 3. Using these events, we can define the real-time requirement (violatedPeriod) and the fault tolerance requirement (wrongFT). The real-time requirement is violated whenever the time between successive periodStart events (stored in an auxiliary variable periodTime) is not between 900 and 1100 ms as defined in line 12. The fault tolerance requirement is defined in terms of the number of times the client failed to connect to some server (an auxiliary variable numConFail), the number of times a query was resent (an auxiliary variable numRetries) in line 13, and oldDataUsed. All auxiliary variables are initialized in lines 16 to 19 when startPgm occurs. The auxiliary variables periodTime is updated with the elapsed time between two consecutive sessions

97

```
01:MonScr RailRoadCrossing
02:    export event startGD, endGD;
03:    export condition IC;
04:
05:    monobj float RRC.train_x;
06:    monobj int RRC.train_length;
07:    monobj int RRC.cross_x;
08:    monobj int RRC.cross_length;
09:
10:    monmeth void Gate.gd(int);
11:    monmeth int Gate.gu();
12:
13:    condition IC = RRC.train_x + RRC.train_length > RRC.cross_x &&
14:                   RRC.train_x <= RRC.cross_x + RRC.cross_length;
15:
16:    event startGD = endM(Gate.gd(int));
17:    event endGD = startM(Gate.gu());
18:End
```

Figure 8.2: The PEDL script for the railroad crossing system

```
01:ReqSpec RailRoadCrossing
02:    import event startGD, endGD;
03:    import condition IC;
04:
05:    condition GD = [startGD, endGD);
06:
07:    property safeRRC = IC -> GD;
08:End
```

Figure 8.3: The MEDL script for the railroad crossing system

```
Client:
  loop periodically (every p time units do)
      randomly select a URL from a given list
      open an HTTP connection to url with a timeout of To
      if connection is established then
          repeat for R times or until successful
              send a CGI query for data
          if unsuccessful then use old data
          else use the data just received
      else use old data
      process data
  endloop
```

Figure 8.4: Pseudo code of the database client

```
01:ReqSpec StockClient
02:   // Imported event declaration from the Stockclient
03:   import event startPgm, periodStart, conFail, queryResend, oldDataUsed;
04:
05:   // Auxiliary variable declartion
06:   var long periodTime;
07:   var long lastPeriodStart;
08:   var int numRetries;
09:   var int numConFail;
10:
11:   // Requirement definition
12:   alarm violatedPeriod = end((periodTime' >= 900) && (periodTime' <= 1100));
13:   alarm wrongFT = oldDataUsed when ((numRetries' < 4) || (numConFail' < 3));
14:
15:   // Auxiliary variable update rules
16:   startPgm ->     { periodTime' = 1000;
17:                       lastPeriodStart'=time(startPgm)-1000;
18:                       numRetries' = 0;
19:                       numConFail' = 0;}
20:   periodStart -> { periodTime' = time(periodStart) - lastPeriodStart;
21:                       lastPeriodStart'= time(periodStart);
22:                       numRetries' = 0;
23:                       numConFail' = 0;}
24:   queryResend -> { numRetries' = numRetries + 1; }
25:   conFail ->      { numConFail' = numConFail + 1; }
26:End
```

Figure 8.5: The MEDL script for the database client

in line 20 when `periodStart` occurs. An auxiliary variable `numRetries` is increased by 1 in line 24 when `queryResend` happens. Similarly, an auxiliary variable `numConFail` is increased by 1 in line 25 when `conFail` happens.

A run-time checker receives events `startPgm`, `periodStart`, `conFail`, `queryResend`, and `oldDataUsed` from an event recognizer at run-time. These events are defined in the PEDL script in Figure 8.6 based on methods and variable defined in `Client` class. A

```
01:MonScr StockClient
02:    // Exported event declaration
03:    export event startPgm, periodStart, conFail, queryResend, oldDataUsed;
04:
05:    // Monitored methods declaration
06:    monmeth void Client.main(String[]);
07:    monmeth void Client.run();
08:    monmeth void  Client.failConnection(ConnectTry);
09:    monmeth Object Client.retryGetData(int);
10:    monmeth Object Client.processOldData();
11:
12:    // Event definition
13:    event startPgm = startM(Client.main(String[]));
14:    event periodStart = startM(Client.run());
15:
16:    event conFail = startM(Client.failConnection(ConnectTry));
17:    event queryResend = startM(Client.retryGetData(int));
18:    event oldDataUsed = startM(Client.processOldData());
19:end
```

Figure 8.6: The PEDL script for the database client

method `main(String[])` is invoked when the client program starts. `run()` is invoked when a new session begins. `failConnection(ConnectTry)` is invoked when connection fails to be established. `retryGetData(int)` is invoked when the client retries to get response from the server. `processOldData()` is invoked when the old data is used instead of new data. These methods are declared as monitored methods in lines 6 to 10. `startPgm` is defined as starting of `main(String[])` in line 13. `periodStart` is defined as starting of `run()` in line 15. `conFail`, `queryResend`, and `oldDataUsed` are defined as startings of methods `failConnection(ConnectTry)`, `retryGetData(int)`, and `processOldData()`, respectively, in lines 16 to 18.

8.2   Micro Air Vehicles

This example of the utility of the MaC approach is taken from an 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 [GSSL99]. One such task involves arranging MAVs into a hexagonal pattern, illustrated in Figure 8.7. 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

100

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 neighbors, using local information only. Gordon and Speers at NRL devised a distributed algorithm to solve this problem [SG99]. The algorithm is based on the relative positions of the neighbors of an MAV.

Figure 8.7: The hexagonal pattern of MAVs

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 violation of the pattern. Intuitively, we should expect that as the pattern forms, the number of such violations should go down.

An implementation-independent MEDL script of this property is shown in Figure 8.8. The primitive event MAValert, supplied by the event recognizer, denotes a misplacement of some neighbor of an MAV. The auxiliary variable currAlert 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 represented by an auxiliary variable prevAlert. 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

Line 2 of Figure 8.8 imports events init which indicates the starting of MAV and MAValert. Line 8 defines an event endPeriod indicating 1000 ms long period ends. The alarm NoPattern is raised when the number of MAValert in current period is increased more that 10% from the number of MAValert in the previous period as defined in line 10. The auxiliary variables periodStart, numMAValert, prevAlert, and currAlert are initialized in lines 12 to 15 when an event init occurs. The auxiliary variable numMAValert is increased by 1 in line 17 whenever MAValert happens. Lines 19 to 22 update currAlert and prevAlert and reset numMAValert as 0 and periodStart as current time when endPeriod occurs.

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 runs, it continuously executes the positioning algorithm and queries its neighbors for their positions. A local variable

101

```
01:ReqSpec HexPattern
02:  import event init, MAValert;
03:
04:  var int    numMAValert;
05:  var float  prevAlert, currAlert;
06:  var long   periodStart;
07:
08:  event endPeriod = ((time(MAValert) - periodStart) > 1000);
09:
10:  alarm NoPattern = start(currAlert' > (prevAlert *1.1)) ;
11:
12:  init -> {      periodStart' = time(init);
13:                 numMAValert' = 0;
14:                 prevAlert' = 0;
15:                 currAlert' = 0; }
16:
17:  MAValert -> {  numMAValert' = numMAValert + 1; }
18:
19:  endPeriod -> { periodStart' = time(endPeriod);
20:                 prevAlert' = currAlert;
21:                 currAlert' = numMAValert;
22:                 numMAValert' = 0; }
23:End
```

Figure 8.8: The MEDL script for pattern monitoring

distance in the `run()` method of the class is used to hold the distance from the currently queried neighbor. The PEDL script for this implementation is shown in Figure 8.9. An

```
01:MonScr MAVpattern
02:
03:export event MAValert, init;
04:
05:  monobj int Air.DIST;
06:  monmeth void Console.createMAVs(int);
07:  monobj double MAV.run().distance
08:
09:  event init = startM( Console.createMAVs(int) );
10:  event MAValert = start ( MAV.run().distance > 0.25 * Air.DIST &&
11:                           MAV.run().distance < 0.75 * Air.DIST );
12:End
```

Figure 8.9: The PEDL script for pattern monitoring

event `MAVAlert` is defined in terms of the value of the variable `distance`. By declaring the variable as a monitored variable, the specification 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 specification. Line 9 defines an event `init` indicating the beginning of the MAV emulator. Lines 10 and 11 define an event `MAValert` as when the distance between any two MAV's becomes too close, i.e. less than 75% of the specified distance `Air.DIST`.

8.3   Analysis of Network Simulations

Network protocols are often analyzed using simulations. We demonstrate how the MaC architecture can be applied to such simulations to check propositions expressing safety properties of network event traces. We uses an extension of NS2 simulator [FV00] by the CMU Monarch group (`http://monarch.cs.cmu.edu`) together with components of the Java-MaC to provide a uniform architecture to analyze network protocols. We call this tool suite *Verisim*. Figure 8.10 shows the overview of Verisim architecture.
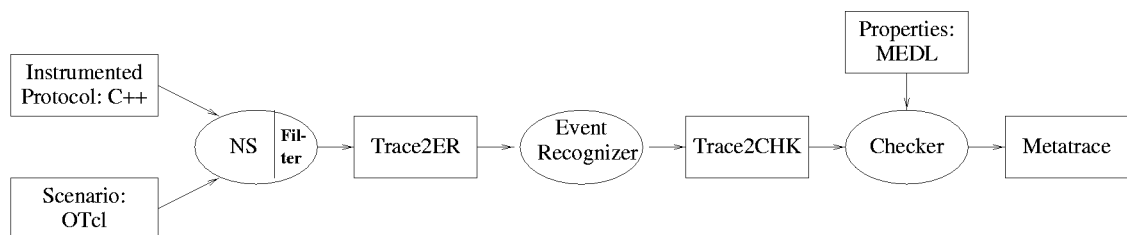
Figure 8.10: The architecture of Verisim

NS2 simulator comes in the place of an instrumented target program. NS2 generates a

trace to an event recognizer through hand-written filter.[1] The execution trace generated by a filter is fed into the event recognizer, which evaluates the events and conditions upon the arrival of snapshots and generates a trace containing events and conditions. Then, the run-time checker receives the trace of events and conditions and generates a *metatrace* consisting of property violations. Based on the meta trace file, we modify MEDL properties to get more bugs detected in a protocol implementation without debugging and running NS2 simulator all over again. This methodology is effective when debugging and running simulation cost high.

The effectiveness of applying Java-MaC to the analysis of network protocols is demonstrated by analyzing simulations of the Ad hoc On-demand Distance Vector (AODV) routing protocol for packet radio networks [Per97]. Our analysis finds violations of significant properties such as loop invariant and destination reply. This case study shows flexibility of Java-MaC in specifying complex network properties and checking the properties. More details are in [BGK$^+$01, BGK$^+$00]

8.4 Hybrid System in Charon Specification Language

Charon [AGH$^+$00] is a specification language for the hierarchical and modular specification of interacting hybrid systems developed in Software Design Research Lab in University of Pennsylvania. Charon is used to specify hybrid systems that exhibit continuous behaviors within modes and discrete jumps between modes. The goals of the Charon specification language include simulating behaviors of hybrid systems, generating a code from the specification. Current Charon system generates only a simulator code in Java. Java-MaC can be applied to monitor a simulator code generated from a specification in Charon.

8.4.1 Background on Charon

A Charon specification defines a hybrid system based on *agents, modes, transitions* and *variables*. An *agent* can create sub-agents and make communication channels between other agents. An agent can have modes and variables. A *mode* defines both algebraic and differential equations on variables. A mode can have its own variables. A mode can create submodes. Agents and modes are similar to Java classes. A *Transition* is defined between modes. A transition has a name, a starting point, ending points and corresponding guards and actions. Action assigns a value to a variable. *Variables* are defined inside of agents and modes. `Real`, `boolean` and `integer` variables are supported.

Entities of Charon specification are translated into entities of Java systematically. Therefore, if a user writes down a PEDL for Charon script whose primitives are agents, modes, variables, and transitions, the PEDL for Charon script can be translated into the PEDL for Java script. Agent definitions and mode definitions in a Charon specification are translated into Java classes through the Charon compiler. Variables defined inside of agents/modes are translated into member fields of classes corresponding the agents/modes. Agents/modes in Charon can be monitored in similar way to monitor objects in Java. A transition, however, is not translated into its own class. Each transition is created by instantiating a class `Transition`. To monitor transitions, Java-MaC monitors `t.name` of

---

[1]NS2 is instrumented manually because Java-MaC is not able to instrument C++ code.

`Simulator.processTransition ( Transition t, int dest)`, which is the name of a transition `t`.

The overall structure of monitoring and checking a simulator code generated from a Charon specification using the MaC architecture is in Figure 8.11. First, a user writes down requirement properties in MEDL. Then, the user writes down a PEDL for Charon script (Charon Layer). Second, a PEDL for Charon script is translated into a PEDL for Java script (Charon/MaC Interface Layer).[2] This translation depends on the design of the Charon compiler. In other words, the way the Charon compiler translates agents, modes, variables, and transitions into Java entities determines the way a PEDL for Charon script is translated into a PEDL for Java script. In addition, a Charon specification is translated into Java program by the Charon compiler. Finally, given Charon program in Java bytecode and the PEDL for Java script, Java-MaC instruments the Charon program (Java-MaC Layer). The rest of monitoring and checking process is the same as usual.
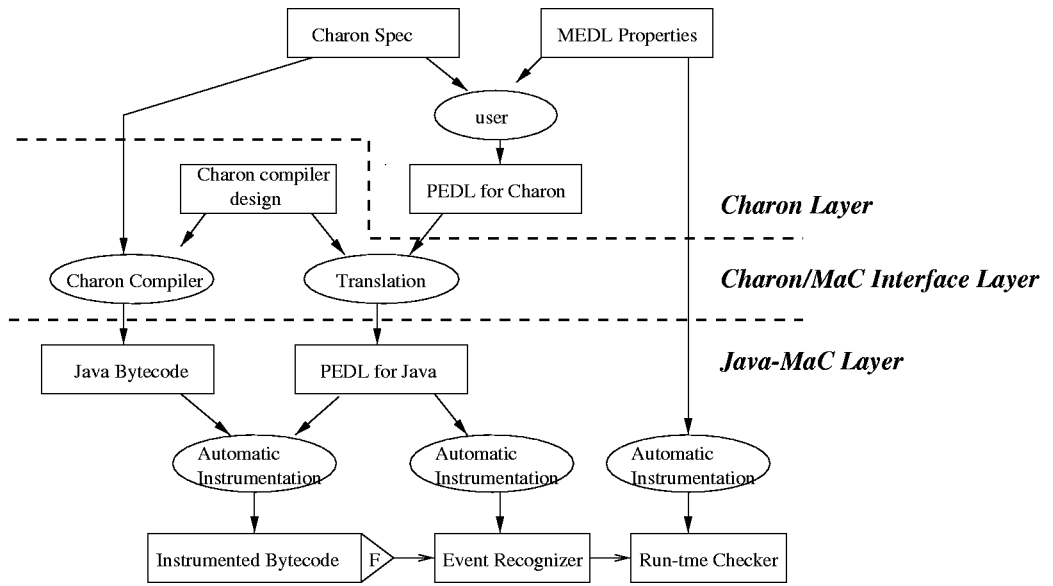
Figure 8.11: Overview of applying Java-MaC for monitoring a Charon simulation

8.4.2 Inverted Pendulum Example

Figure 8.12 illustrates an inverted pendulum (IP). The control objective is to move the cart from one position to a desired goal position maintaining the pendulum at the upright position. In this case study, starting position is 0 and goal position is 0.5.

Figure 8.13 shows a diagram of the inverted pendulum system in Charon [FHLS00], which consists of `IPControllers`, `IPDecisionModule`, and `IPPhysicalSystem`. `IPControllers` provides control signals to `IPDecisionModule` which implements control switching logic of the Simplex Architecture [SKSC98]. `IPControllers` has three different controllers: `controllerEC` (experimental controller), `controllerBC` (baseline controller) and `controllerSC` (safety controller). Initially, experimental controller whose reliability is not trusted controls

---

[2]We have not implemented a translator from a PEDL for Charon script to a PEDL for Java script yet.
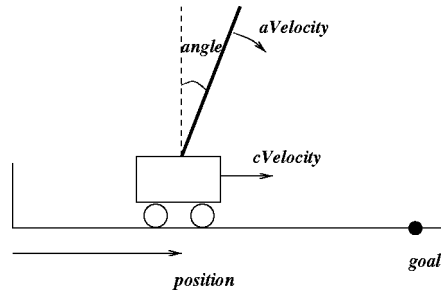
Figure 8.12: The inverted pendulum system

IP = IPControllers ‖ IPDecisionModule ‖ IPPhysicalSystem

VaBC: control output from baseline controller
VaEC: control output from experimental controller
VaSC: control output from safety controlller
position: position of cart   angle: vertical angle
cVelocity: cart velocity   aVelocity: angular velocity
ctrlCmd: control output of IPController
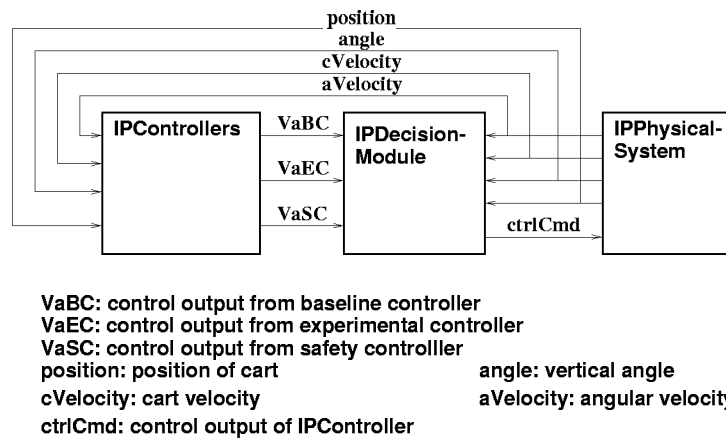
Figure 8.13: The diagram of the inverted pendulum specification

the cart. If the system enters undesirable state, the safety controller which performs reliably but not optimally takes over the control. Once the system enters the operating region of the baseline controller, control is given to a baseline controller. IPDecisionModule reads control signals from IPControllers and generates control command. IPPhysicalSystem reads that command and updates the system states consisting of position (the cart's position), angle (vertical angle of the inverted pendulum of the cart), cVelocity (the velocity of the cart), aVelocity (the angular velocity of the inverted pendulum).

We monitor following events in IP.

- the cart achieves the control objective: to reach the goal position with upright pendulum.

- control is passed from one controller to another. In other words, transitions between controllerEC, controllerBC, and controllerSC are monitored.

- the mission fails: pendulum falls down completely

PEDL script for IP are in Figure 8.14. We allow tolerance of 0.01 for position,cVelocity,angle, and aVelocity. We monitor four variables aVelocity, cVelocity, angle, and position

106

declared in lines 9 to 16.[3] We monitor transitions by monitoring `transName` which contains the name of current transition declared in lines 17 and 18. An event `finish` is defined in lines 21 to 24 as `position` reaches 0.5 which is the goal position, `cVelocity` and `aVelocity` becomes 0, and the vertical angle `angle` becomes 0. An event `fallDown` is defined in lines 33 to 34 as the vertical angle becomes 90 or -90 degree which indicates the pendulum falls down completely. Events indicating that control is given from one controller to another are defined in lines 25 to 30.

MEDL script for IP are in Figure 8.15. Lines 2 to 7 indicate the imported events from the event recognizer. Alarm `fail` is defined as when the pendulum falls down in line 9. Line 11 to 32 display what events are occuring for the convenience of users. For example, if the run-time checker receives an event `finish`, the run-time checker shows the following message to the screen.

```
###########################################################
The cart has reached the goal with rod standing upright!
###########################################################
```

8.4.3 Overhead Measurement

We measured the overhead induced by Java-MaC on IP. IP runs on IBM JVM 1.3.0 with JIT in Linux 2.2 machine (2X 550Mhz PIII 5, 1GB memory). An event recognizer and a run-time checker run on Sun HotSpot JVM 1.3.1 in the same Windows 2000 machine (1.4Ghz Pentium4, 512MB memory). Both machines are on same network domain. A filter has a communication buffer of 512 byte and sends snapshot using TCP socket. Experiments were performed 10 times for each case.

First, we measured size/execution time of uninstrumented IP. IP has total 131 classfiles, which takes 1176 kilobytes. At run-time, IP takes 58M bytes memory.[4]

- elapsed time: 85.2 second (standard deviation 0.9%).

- user time: 84.65 second (standard deviation 4.9%).

- system time: 3.29 second (standard deviation 6.5%).

As we can see, IP is a CPU intensive program.

Second, we measured size/execution time of the instrumented IP. 10 classes are instrumented and increased by total 2224 bytes.

- `IPSimplexTestSimulation.class`: 19811 → 19893 bytes

- `IPSimplexTestgSim.class`(main classfile): 3264 → 3817 bytes

- `controllers.class`: 3542 → 3735 bytes

- `AnalogVar.class`: 2240 → 2456 bytes

- `Bool.class`: 1008 → 1242 bytes

---

[3]PEDL allows a user to use a short alias to the monitored variable/method by using `<-` keyword.
[4]Memory usage is obtained using `top`.

- `DiscreteVar.class`: 1894 → 2118 bytes

- `Real.class`: 966 → 1178 bytes

- `Real.class`: 849 → 1055 bytes

- `RealD.class`: 853 → 1059 bytes

- `Variable.class`: 1374 → 1472 bytes

We performed experiments with and without delta abstraction and value abstraction. As you may see, overhead reduction techniques do not seem to reduce the overhead substantially. This is because the overhead itself caused by Java-MaC is too small to demonstrate the benefit of overhead reduction techniques and the standard deviation of measured execution time masks out the difference.

Without overhead reduction techniques At run-time, IP takes 59M bytes memory, which is 1 M bytes more than memory used by uninstrumented IP. The total elapsed time is 85.9 second (standard deviation 0.61%) which is 0.9% slowed down compared to the original execution. A total number of `sendObjMethod()` invocation was around 6.8×10^5. A total number of snapshots sent from the target program to an event recognizer was around 6300. A total volume of snapshots was around 62k bytes.

With delta values At run-time, IP takes 59M bytes memory, which is 1 M bytes more than memory used by uninstrumented IP. The total elapsed time is 85.7 second (standard deviation 0.8%) which is 0.6% slowed down compared to the original execution.

A total number of `sendObjMethod()` invocation was around 6.8×10^5. A total number of snapshots sent from the target program to an event recognizer was around 6300. A total amount of snapshot was 40k bytes. As you see, the delta abstraction reduces the total amount of snapshots by 37 (= (62-40)/62)%.

With value abstraction At run-time, IP takes 59M bytes memory, which is 1 M bytes more than memory used by uninstrumented IP. The total elapsed time is 86.0 second (standard deviation 1.26%) which is 0.9% slowed down compared to the original execution. A total number of `sendObjMethod()` invocation was around 6.8×10^5. A total number of snapshots sent from the target program to an event recognizer was 35. The value abstraction abstracts out most snapshots. A total amount of snapshot was 364 bytes.

```
01:MonScr IP
02:     export event
03:     finish,         // 1. Goal is achieved
04:     ECtoSC,ECtoBC,   // 2. Control is given from EC to other controller
05:     BCtoEC,BCtoSC,   //    Control is given from BC to other controller
06:     SCtoEC,SCtoBC,   //    Control is given from SC to other controller
07:     fallDown;        // 3. Pendulum falls down
08:
09:     monobj double aVelocity
10:         <- CHARON.simulator.agents.controllers.aVelocity.v;
11:     monobj double cVelocity
12:         <-CHARON.simulator.agents.controllers.cVelocity.v;
13:     monobj double angle
14:         <- CHARON.simulator.agents.controllers.angle.v;
15:     monobj double position
16:         <- CHARON.simulator.agents.controllers.position.v;
17:     monobj String transition
18:         <-CHARON.simulator.Simulation.transName;
19:
20:     // Tolerance : 0.01
21:     event finish = start(0.49 < position && position < 0.51 &&
22:                          - 0.01 < cVelocity && cVelocity < 0.01 &&
23:                          - 0.01 < angle && angle < 0.01 &&
24:                          - 0.01 < aVelocity && aVelocity <0.01 );
25:     event ECtoSC = start(transition == "ECtoSC");
26:     event ECtoBC = start(transition == "ECtoBC");
27:     event BCtoEC = start(transition == "BCtoEC");
28:     event BCtoSC = start(transition == "BCtoSC");
29:     event SCtoEC = start(transition == "SCtoEC");
30:     event SCtoBC = start(transition == "SCtoBC");
31:
32:     // Tolerance : 0.01
33:     event fallDown = start( (89.99 < angle && angle < 90.01) ||
34:                          (- 90.01 < angle && angle < -89.99));
35:End
```

Figure 8.14: PEDL script for IP

```
01:ReqSpec IP
02:    import event
03:    finish,        // 1. Goal is achieved
04:    ECtoSC,ECtoBC, // 2. Control is given from EC to other controller
05:    BCtoEC,BCtoSC, //    Control is given from BC to other controller
06:    SCtoEC,SCtoBC, //    Control is given from SC to other controller
07:    fallDown;      // 3. Pendulum falls down
08:
09:    alarm fail = fallDown;
10:
11:    finish -> {
12:        print "#################################################################";
13:        print "The cart have reached the goal with rod standing upright!";
14:        print "#################################################################";}
15:    ECtoSC -> {
16:        print "An expreiment controller is changed into a safe controller!";
17:        print "";}
18:    ECtoBC -> {
19:        print "An expreiment controller is changed into a basic controller!";
20:        print "";}
21:    BCtoEC -> {
22:        print"A basic controller is changed into a experimental controller!";
23:        print "";}
24:    BCtoSC -> {
25:        print "A basic controller is changed into a safe controller!";
26:        print "";}
27:    SCtoEC -> {
28:        print"A safe controller is changed into an experimental controller!";
29:        print "";}
30:    SCtoBC -> {
31:        print"A safe controller is changed into a basic controller!";
32:        print "";}
33:End
```

Figure 8.15: MEDL script for IP

Chapter 9

Summary and Future Work

9.1 Summary

We have presented an architecture for run-time formal analysis, called Monitoring and Checking (MaC) architecture, as a complementary solution to the formal verification and testing for the assurance of correct target program execution. First, we investigated several fundamental issues including issues on monitorable properties, the complexity of property evaluation in the presence of non-determinism, and abstraction of program execution. Second, we developed the MaC architecture with two specification languages, i.e., Primitive Event Definition Language (PEDL) and Meta Event Definition Language (MEDL). We designed the architecture as a modular architecture consisting of several components for the increased flexibility. Third, we have implemented a MaC prototype for Java programs called Java-MaC solving the platform specific issues such as monitoring objects. Lastly, we showed the effectiveness of the MaC architecture through several examples including the emulator of distributed controller for large numbers of mobile agents, Ad-hoc On Demand Vector (AODV) routing protocol analysis, and monitoring an inverted pendulum simulation.

The thesis of this dissertation is that run-time formal analysis can provide confidence in the correct execution of software systems in a practical manner. The technical contributions in the dissertation are summarized in the followings.

- **Rigorous analysis**
  The MaC architecture uses a formal requirement specification to monitor and check the execution of the target program. In addition, instrumentation, monitoring, and checking are performed automatically following the formal requirement specifications without requiring human interaction, which leads to the accurate analysis.

- **Flexibility**
  The MaC architecture separates monitoring program-dependent, low-level behavior from checking high-level behavior in both its specification languages (PEDL and MEDL) and its run-time analysis components (a filter, an event recognizer, and a run-time checker). Furthermore, the interfaces between the components are well-defined. These features allow users to extend the MaC architecture for their own purpose by plugging third-party tools in the architecture.

111

- **Automation**

  The analysis procedure of the MaC architecture is fully automatic. First, the target program is instrumented automatically according to a low-level specification. Second, the architecture monitors and checks the execution of a target program automatically following formal requirement specifications.

- **Ease of use**

  Users do not need to change their execution environment such as VM/OS/HW to use the MaC architecture. In addition, the analysis procedure including instrumentation, monitoring, and checking are performed automatically without requiring human directions. Finally, users do not need complex source code recompilation because the architecture works on the executable code directly.

- **Generality**

  The MaC architecture is application-independent. In addition, the architecture works on the executable code which has great availability compared to the source code. Furthermore, the openness of the architecture makes the architecture extendible to various application areas.

This dissertation discusses the design of the MaC architecture and its implementation Java-MaC. We have partly shown the effectiveness of run-time formal analysis by presenting the design and implementation of the MaC architecture with several case studies. We did not, however, have a chance to prove the practicality of run-time formal analysis. The main reason for this missing proof of the practicality is that we could not get real-world safety critical programs written in Java. We expected that Java would become a favorable programming language in safety critical fields when we started research on run-time formal analysis. People have been extending Java programming language for such safety real-time systems. However, currently, Java is still not a programming language of the choice in the safety critical areas. As the second best way, we have applied Java-MaC to simulators so that we may show its practicality, at least partially (see Chapter 8). Through several case studies, we found that the MaC architecture could be useful as an extended/supplementary testing architecture. We believe that we can provide this missing proof of practicality as safety critical applications start to be written in Java in near future.

9.2 Future Work

9.2.1 Value Abstraction

The MaC architecture performs value abstraction to reduce overhead. The heuristic for value abstraction used in Java-MaC is, however, primitive because value abstraction is applied only to very simple expressions. We will extend the heuristic for broad applicability.

9.2.2 Monitoring Objects

Java-MaC monitors the execution of target program through monitoring objects. Due to the complexity of dynamic object behavior, however, Java-MaC assumes that an object is not assigned with another object. We will investigate how to resolve this restriction so

that a broader range of programs can be monitored. We may modify JVM so that we can access necessary information to test references from JVM, which eliminate the need of the address table.

9.2.3 Application Area

There has been an continuous effort to add a real-time features to Java so that safety critical application can be written in Java. We believe that we can apply Java-MaC to a safety critical systems written in Java in near future. In addition, Java-MaC has shown its potential as a flexible software testing tool through several examples including AODV protocol analysis. We will continue investigating issues for applying Java-MaC as a software testing tool.

Bibliography

[ACHH93]   R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. Hybrid automata:an algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems I LNCS 736*, 1993.

[AEK+99]   R. Alur, J. Esposito, M. Kim, V. Kumar, and I. Lee. Formal modeling and analysis of hybrid systems: A case study in multirobot coordination. In *FM'99 World Congress On Formal Methods In The Development Of Computing Systems*, Sep 20-24, 1999.

[AEK+00]   R. Alur, J. Esposito, M. Kim, V. Kumar, and I. Lee. Formal modeling and analysis of hybrid systems: A case study in multirobot coordination. 2000. Submitted to Formal Methods in System Design.

[AGH+00]   R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specifications of hybrid systems in CHARON. In *Hybrid Systems: Computation and Control*, LNCS 1790, pages 6–19, 2000.

[AH96]   R. Alur and T. A. Henzinger. Reactive modules. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, 1996.

[AH98]   R. Alur and T. A. Henzinger. *Computer-Aided Verification*. To be published, 1998.

[Ari96]   Ariane 5 flight 501 failure, 1996. `http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html`.

[AS85]   B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[BCD+90]   J. Burch, E. Clarke, D. Dill, L. Hwang, and K. McMillan. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, 1990.

[Ber97]   P. Bertelsen. Semantics of Java byte code. Technical report, Royal Veterinary and Agricultural University, 1997.

[BGK+00]   K. Bhargavan, C. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. In *International Symposium on Software Testing and Analysis*, August 2000.

[BGK+01]   K. Bhargavan, C. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transaction on Software Engineering*, 2001. To be published.

[BGLG93]   P. Brémond-Grégoire, I. Lee, and R. Gerber. ACSR: An Algebra of Communicating Shared Resources with Dense Time and Priorities. In *Proc. of CONCUR '93*, 1993.

[Bow95]   J. P. Bowen. Ten commandments of formal methods. In *IEEE Computer*, April 1995.

[Bry86]   R. E. Bryant. Graph Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Computers*, C35(8):677–691, 1986.

[CG92]   S. E. Chodrow and M. G. Gouda. The Sentry System. In *SRDS11*, October 1992.

[CG95]   S. E. Chodrow and M. G. Gouda. Implementation of the sentry system. In *Software-Practice and Experience*, Apr 1995.

[CG96]   S. E. Chodrow and M. G. Gouda. Sentries for the execution of concurrent program. In *ICDCS*, May 1996.

[CGL94]   E. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In *A Decade of Concurrency - Reflections and Perspectives. Lecture Notes in Computer Science vol 803*, 1994.

[CK96]   E. M. Clarke and R. P. Kurshan. Computer-aided verification. In *IEEE Spectrum 1996*, 1996.

[CL97]   D. Clarke and I. Lee. Automatic test generation for the analysis of a real-time system: Case study. *Proceedings of 3rd IEEE Real-Time Technology and Applications Symposium*, June 1997.

[CW96]   E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[Dig]   Digital Equipement Corp. *DIGITAL JTrek* . http://www.digital.com/ java/download/jtrek/index.html.

[DKM+94]   L. Dillon, G. Kutty, L. Moser, P. M. Melliar-Smith, and Y. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transations on Software Engioneering and Methodology*, April 1994.

[DR96]   L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'96), Software Engineering Notes*, pages 106–117, 1996.

[DS97]   B. Dutertre and V. Stavridou. Formal requirements analysis of an avionics control system. *IEEE Transaction on Software Engineering*, 23(5), May 1997.

115

[DY94]   L. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. *Software Engineering Notes*, December 1994.

[FHLS00]   R. Fierro, Y. Hur, I. Lee, and L. Sha. Modeling the simplex architecture using charon. In *RTSS work in progress*, 2000.

[Fla99]   D. Flanagan. *Java in a Nutshell.* O'Reilly, 1999.

[FMP95]   Formal methods publications, 1995.   `http://www.comlab.ox.ac.uk/ archive/formal-methods/pubs.html.`

[FV00]   K. Fall and K. Varadhan. *ns Notes and Documentation.* The VINT Project, 2000.

[Gai86]   J. Gait. A probe effect in concurrent programs. *Software-Practice and Experience*, 1986.

[GJS00]   J. Gosling, B. Joy, and G. Steele. *The Java Language Specification 2nd ed.* Addison Wesley, 2000.

[GKND97]   M. Gibble, J. C. Knight, L. G. Nakano, and C. Dejong. Formal verification: An evaluation. Technical report, CS Dept. Univ. of Virginia, 1997. Tech Report CS-97-13.

[GSSL99]   D. Gordon, W. Spears, O. Sokolsky, and I. Lee. Distributed spatial control and global monitoring of mobile agents. In *IEEE International Conference on Information, Intelligence, and Systems*, Nov 1999.

[HCRP91]   N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEEE*, 1991.

[HD96]   C. Heitmeyer and D. Mandrioli, *Eds. Formal Methods for Real-Time Systems.* Number 5 in Trends in Software. John Wiley and Sons, 1996.

[HP90]   J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1990.

[Jef93]   C. L. Jeffery. *A Framework for Monitoring Program Execution.* PhD thesis, CS Dept. Univ. of Arizona, 1993.

[JG90]   F. Jahanian and A. Goyal. A formalism for monitoring real-time constraints at run-time. *20th Int. Symp. on Fault-Tolerant Computing Systems (FTCS-20)*, pages 148–55, 1990.

[JVM99]   Java platform debugger architecture, 1999.   `http://java.sun.com/ products/jpda/.`

[JZTB98]   C. L. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. *Program Analysis for Software Tools and Engineering*, July 1998.

[Kah99]   G. Kahn. Building robust software, 1999. Invited Talk at FM'99 World Congress on Formal Methods.

[KKL+01]   M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-Mac: a run-time assurance tool for java programs. *First Workshop on Runtime Verification*, July 2001.

[Kur87]   R. P. Kurshan. Reducibility in analysis of coordination. In *LNCS*, volume 103, pages 19 –39. Springer-Verlag, 1987.

[Kur94]   R. P. Kurshan. *Computer Aided Verification of Coordinating Processes: The Automata Theoretic Approach*. Princeton Series in Computer Science. Princeton University Press, 1994.

[KVBA+98] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. A framework for run-time correctness assurance of real-time systems. Technical Report MS-CIS-98-37, University of Pennsylvania, 1998.

[KVBA+99] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *Proceedings of the European Conference on Real-Time Systems - ECRTS'99*, pages 114–121, June 1999.

[Lev95]   N. G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 1995.

[LKK+99]   I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Runtime assurance based on formal specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications - PDPTA'99*, June 1999.

[LM99]   G. Liu and A. K. Mok. Implementation of Jem - a Java composite event package. In *RTAS*, 1999.

[LMK98]   G. Liu, A. K. Mok, and P. C. Konana. A unified approach for specifying timing constraints and composite events in active real-time database systems. In *RTAS*, June 1998.

[Log00]   X. Logean. *Run-time Monitoring and On-Line Testing of Middleware based Communication Services*. PhD thesis, Communication Systems Dept. Swiss Federal Institute of Technology, Lausanne, 2000.

[LY99]   T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, 2nd ed.* Addison Wesley, 1999.

[McM93]   K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[Mil89]   R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[ML97a]   A. K. Mok and G. Liu. Efficient run-time monitoring of timing constraints. In *IEEE Real-Time Technology and Applications Symposium*, June 1997.

[ML97b]   A. K. Mok and G. Liu. Early detection of timing constraint violation at runtime. In *RTSS*, Dec 1997.

[MP92]   Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[Mye76]   G. J. Myers. *Software Reliability:Principles and Practices*. John Wiley and Sons, 1976.

[Mye79]   G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[OCH90]   R. A. Olsson, R. H. Crawford, and W. W. Ho. Dalek: A gnu, improved programmable debugger. *Proc. Summer 1990 USENIX Conference*, June 1990.

[OCH91]   R. A. Olsson, R. H. Crawford, and W. W. Ho. A dataflow approach to event-based debugging. *Software - Practive and Experience*, February 1991.

[Par93]   D. L. Parnas. Predicate Logic for Software Engineering. *IEEE Transactions on Software Engineering*, SE-19(9):856–861, September 1993.

[Pel94]   D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proceedings of the Computer Aided Verification, Sixth International Workshop (CAV'94)*. Springer Verlag, 1994.

[Per97]   C. Perkins. Ad hoc on-demand distance vector (aodv) routing, November 1997. Internet-Draft Version 00, IETF.

[Pnu77]   A. Pnueli. The Temproal Logic of Programs. In *Proc. of Foundations of Computer Science*, pages 46–57, 1977.

[Pnu99]   A. Pnueli. Deduction is forever., 1999. Invited Talk at Formal Methods '99.

[PO96]   O. Parissis and F. Ouabdesselam. Specification-based testing of synchronous software. *Proceedings of the 4th ACM SIGSOFT SYmposium on the Foundations of Software ENgineering*, 1996.

[Pre99]   President's Information Technology Advisory Commitee. *Information Technology Research: Investing in Our Future*, 1999. http://www.ccic.gov/.

[REF97]   Java core reflection, 1997. http://java.sun.com/products/jdk/1.1/docs/guide/reflection/.

[REF99]   Java reference objects and garbage collection, 1999. http://developer.java.sun.com/developer/technicalArticles/ALT/RefObj.

[Rov97]   Time rover home page, 1997. http://www.time-rover.com/.

[SDL89]   Functional specification and description language recommendations z.100-z.104(blue book), 1989.

[SG99]   W. M. Spears and D. F. Gordon. Using artificial physics to control agents. In *Proceedings of the International Conference on Information Intelligence and Systems*, 1999.

[SKSC98]   D. Seto, B. H. Krogh, L. Sha, and A. Chutinan. Dynamic control system upgrade using the Simplex Architecture. *IEEE Control Systems*, 18(4):72–80, 1998.

[Sno88]   R. Snodgrass. A relational approach to monitoring complex systems. In *ACM Transactions on Computer Systems*, May 1988.

[Sos92]   R. Sosič. Dynascope: A tool for program directing. In *Proceedings of SIG-PLAN'92 Conference on Programming Language Design and Implementation*, 1992.

[Sos95a]   R. Sosič. A procedural interface for program directing. In *Software-Practice and Experience*, July 1995.

[Sos95b]   R. Sosič. The dynascope directing server: Design and implementation. In *Computing Systems*, Spring 1995.

[SS98]   T. Savor and R. E. Seviora. Toward automatic detection of software failures. In *IEEE Computer*, pages 68–74, August 1998.

[SSL96]   Secure socket layer 3.0 specification, 1996. `http://home.netscape.com/eng/ssl3/`.

[TFC90]   J. Tsai, K. Fang, and H. Chen. A noninvasive architecture to monitor real-time distributed systems. In *IEEE Computer*, March 1990.

[TJ98]   K. S. Templer and C. L. Jeffery. A configurable automatic instrumentation tool for ANSI C. *Automated Software Engineering*, October 1998.

[Vis00]   M. Viswanathan. *Foundations for the Run-time Analysis of Software Systems*. PhD thesis, CIS Dept. Univ. of Pennsylvania, 2000.

[Web]   WebGAIN Corp. *Java Compiler Compiler v 1.1*. `http://www.webgain.com/products/metamata/java_doc.html`.

[WTK00]   T. Willemse, J. Tretmans, and A. Klomp. A case study in formla methods: Specification and validation of the om/rr protocol. In *Fifth Intl. ERCIM Worksho on Formal Methods for Industrial Critical Systmes (FMICS 2000)*, 2000.

[YY91]   W. Yeh and M. Young. Compositional Reachability Analysis Using Process Algebra. In *Proc. of Testing, Analysis and Verification*, August 1991.

Appendix A

Interface between Filter and Event Recognizer

After a connection between a filter and an event recognizer is established, the filter sends snapshots to the event recognizer. The connection must guarantee delivery of snapshots in order without losing one. A snapshot consists of an ID field for a monitored variable (1 byte) and a value field of the monitored variable (variable size). The size of value field depends on the type of the monitored variable.[1]. A snapshot reporting a beginning/ending of a method has a boolean value field: `false` means beginning of a method and `true` means ending of the method.[2] A snapshot is either a *special snapshot* which has a reserved functionality for monitoring or a *regular snapshot* which contains a value of the monitored variable.

ID from -9 to 9 are reserved for special snapshots.

- ID 1 is defined for a periodic tick indicating a *period* amount of time has been elapsed since last periodic tick 7.2. This snapshot does not have a value field.

- ID 2 is defined for multiple periodic ticks indicating multiple *period* amount of time has been elapsed since last periodic tick. This snapshot has 1 byte value field containing how many periods has elapsed.

- The remaining range among -9 to 9 are not defined.

A regular snapshot has an ID starting from 10 to 127. Negative ID in the snapshot means that the snapshot contains a delta value of a monitored variable of positive ID (see Section 7.2.1). Therefore, the maximum number of monitored entities is 118. We decided delta types as the followings.

- the delta type of `long` : `int`

- the delta type of `int` and `short` : `byte`

- the delta type of `double` : `float`

---

[1] A `String` value is delivered in Java modified UTF format [GJS00]

[2] Values for a beginning/ending of a method are not yet implemented.

Figure A.1 shows an example of communication between a filter and an event recognizer. The first snapshot is a monitored variable A.b1.x whose value is 35. The second one is A.b1.y whose value is 3.14. The third one is a periodic time stamp. The fourth one is A.b1.x represented as a snapshot containing delta value. The delta value of the snapshot is -1, which means that A.b1.x is 34 (= 35-1).

ID	Var Name	Type
10	A.b1.x	int
11	A.b1.y	double
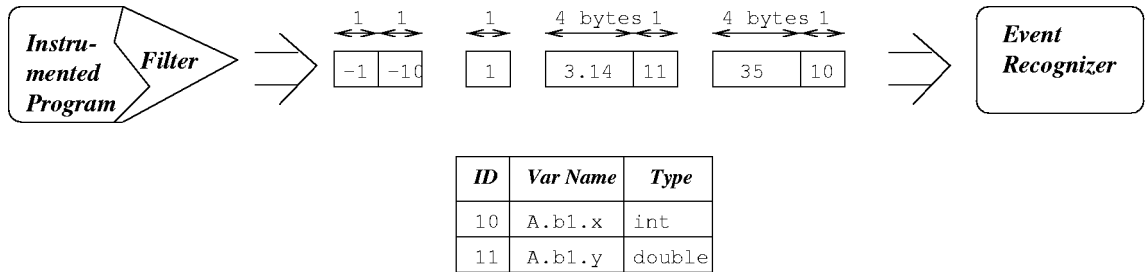
Figure A.1: Example of snapshots from a filter to an event recognizer

Appendix B

Interface between Event Recognizer and Checker

After a connection between an event recognizer and a checker established, the event recognizer sends messages to the checker. The connection must guarantee delivery of messages in order without losing one.

Initially, the event recognizer sends an object of `InterfaceERChecker` containing a table mapping event IDs to event names and mapping condition IDs to condition names. MEDL script and PEDL script are compiled separately. A checker cannot recognize what an ID from the event recognizer means without the table from the event recognizer.

Then, the event recognizer sends events. Also, the event recognizer sends conditions which change its value among **true**, **false** and **undefined**. Multiple events can happen at one instance. Similarly multiple conditions can change their values at one instance. Therefore, the event recognizer sends a group of messages to the checker.

A header of group of message consists of

- Timestamp : 8 bytes **long**

- a number of events: 1 byte

- a number of conditions becoming true: 1 byte

- a number of conditions becoming false: 1 byte

- a number of conditions becoming undefined: 1 byte

After the header is sent, events are sent first, then conditions becoming **true**, conditions becoming **false**, conditions becoming **undef** are sent in order.

- For an event, 1 byte `eventID`, 1 byte `typeEventValue`[1] and a value of the events which can be various size depending on its type are sent.

- For a condition, 1 byte `conditionID` is sent

---

[1] `typeEventValue` is defined in `mac.types.interfaceFilterER.DataTypes`

eventID and conditionID can be any number between -128 to 127. Therefore, maximum number of events and conditions defined should be less than or equal to 256

Figure B.1 shows an example of communication between an event recognizer and a checker. An event recognizer sends a table containing condition/event names and IDs associated with these names. A condition inCrossing has an ID 0. An event enteringCrossing has an ID 1. The first packet is a header indicating that one condition becomes true and one event happens at the time instant represented by 72342400. The second packet indicates that an event enteringCrossing, whose value is int 13, happens. The third packet indicates inCrossing becomes true.

Event Recognizer →

1	4	1	1	1	1	1	1	8 bytes
0	13	1	1	0	0	1	1	72342400

ID	Condition/Event name
0	inCrossing
1	enteringCrossing
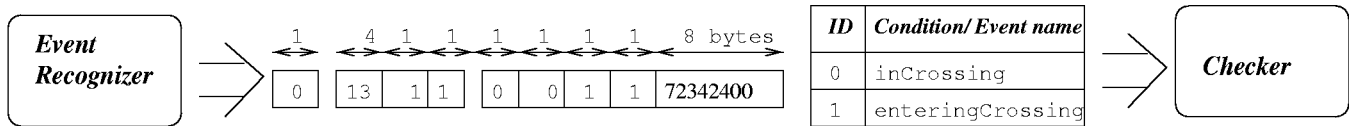
→ **Checker**

Figure B.1: Example of messages from an event recognizer to a checker

Appendix C

PEDL for Java Grammar

BNF for pedl.jj

NON-TERMINALS

```
                addObj::= java code
              location::= java code
       checkMethodObj::= java code
            contained::= java code
              addExec::= java code
             setRelOp::= java code
           setArithOp::= java code
      addToSymbolTable::= java code
   searchSymbolTable::= java code
  connectAndSetName::= java code
        error_skipto::= java code
     MonitoringScript::= <MONSCR> <IDENTIFIER> Declarations
                         Definitions <END>
      TimeDeclaration::= <TIMESTAMP> <INTEGER_LITERAL> (
                         <IDENTIFIER> )?
     ValueAbstraction::= <VALUEABSTRACTION>
 MultiThreadDeclaration::= <MULTITHREAD>
 DeltaValueDeclaration::= <DELTAVALUE>
         Declarations::= ( ( ( EventDeclaration |
                         ConditionDeclaration | TimeDeclaration |
                         ValueAbstraction | DeltaValueDeclaration |
                         MultiThreadDeclaration |
                         MonitoredObjectDeclaration |
                         MonitoredMethodDeclaration ) ) ( ";" )? )*
          Definitions::= ( ( ( EventDefinition | ConditionDefinition
                         ) ) ( ";" )? )*
     EventDeclaration::= <EXPORT> <EVENT> EventNames
 ConditionDeclaration::= <EXPORT> <CONDITION> ConditionNames
```

124

```
                 EventNames::= <IDENTIFIER> ( "," <IDENTIFIER> )*
             ConditionNames::= <IDENTIFIER> ( "," <IDENTIFIER> )*
  MonitoredObjectDeclaration::= <MONOBJ> Type ( AliasVariableName "<-" )?
                                ObjectName ( "," ( <IDENTIFIER> "<-" )?
                                ObjectName )*
  MonitoredMethodDeclaration::= <MONMETH> ( Type | <VOID> ) (
                                AliasMethodName "<-" )? MethodName
                 ObjectName::= ( <IDENTIFIER> "." )* <IDENTIFIER> ( "."
                                <IDENTIFIER> FormalParameters )? "."
                                <IDENTIFIER>
    MethodNameWOparameters::= ( <IDENTIFIER> "." )+ <IDENTIFIER>
                 MethodName::= MethodNameWOparameters FormalParameters
          AliasVariableName::= <IDENTIFIER>
            AliasMethodName::= <IDENTIFIER>
           FormalParameters::= "(" ( Type ( "," Type )* )? ")"
            EventDefinition::= <EVENT> <IDENTIFIER> "=" EventExpression
            EventExpression::= ( SimpleEventExpression ( ( EventOp
                                EventExpression ) )? OptionalWhen )
               OptionalWhen::= ( <WHEN> ConditionalExpression )*
      SimpleEventExpression::= ( <IDENTIFIER> | "(" EventExpression ")" |
                                <START> "(" ConditionalExpression ")" |
                                <END> "(" ConditionalExpression ")" |
                                <UPDATE> "(" ( ObjectName ")" |
                                <IDENTIFIER> ")" ) | <STARTM> "("
                                MethodNameWOparameters FormalParameters ")"
                                | <ENDM> "(" MethodNameWOparameters
                                FormalParameters ")" | <IOM> "("
                                MethodNameWOparameters FormalParameters ")"
                                )
                    EventOp::= "&&"
                              | "||"
        ConditionDefinition::= <CONDITION> <IDENTIFIER> "="
                                ConditionalExpression
                       Type::= ( ( PrimitiveType | Name ) ( "[" "]" )* )
              PrimitiveType::= ( "boolean" )
                              | ( "char" )
                              | ( "byte" )
                              | ( "short" )
                              | ( "int" )
                              | ( "long" )
                              | ( "float" )
                              | ( "double" )
                              | ( "String" )
                       Name::= <IDENTIFIER> ( "." ( <IDENTIFIER> ) )*
                   NameList::= Name ( "," Name )*
                 Expression::= ConditionalExpression
```

```
         ConditionalExpression::= ConditionalOrExpression
       ConditionalOrExpression::= ( ConditionalAndExpression ( "||"
                                   ConditionalAndExpression )* )
      ConditionalAndExpression::= ( InclusiveOrExpression ( "&&"
                                   InclusiveOrExpression )* )
         InclusiveOrExpression::= ( ExclusiveOrExpression ( "|"
                                   ExclusiveOrExpression )* )
         ExclusiveOrExpression::= ( AndExpression ( "^" AndExpression )* )
                 AndExpression::= ( EqualityExpression ( "&"
                                   EqualityExpression )* )
            EqualityExpression::= ( <INM> "(" MethodNameWOparameters
                                   FormalParameters ")" )
                                | ( <DEFINED> "(" ConditionalExpression ")" )
                                | InstanceOfExpression ( ( ( "==" | "!=" )
                                   InstanceOfExpression ) )*
          InstanceOfExpression::= ( RelationalExpression ( "instanceof" Type
                                   )? )
          RelationalExpression::= ShiftExpression ( ( ( "<" | ">" | "<=" |
                                   ">=" ) ShiftExpression ) )*
               ShiftExpression::= AdditiveExpression ( ( ( "<<" | ">>" |
                                   ">>>" ) AdditiveExpression ) )*
            AdditiveExpression::= MultiplicativeExpression ( ( ( "+" | "-" )
                                   MultiplicativeExpression ) )*
      MultiplicativeExpression::= UnaryExpression ( ( ( "*" | "/" | "%" )
                                   UnaryExpression ) )*
               UnaryExpression::= ( ( "+" | "-" ) UnaryExpression )
                                | UnaryExpressionNotPlusMinus
                                | ( <TIME> "(" EventExpression ")" )
                                | ( <VALUE> "(" EventExpression ","
                                   Expression ")" )
 UnaryExpressionNotPlusMinus::= ( ( "~" | "!" ) UnaryExpression )
                                | CastExpression
                                | PrimaryExpression
                CastLookahead::= "(" PrimitiveType
                                | "(" Name "[" "]"
                                | "(" Name ")" ( "~" | "!" | "(" |
                                   <IDENTIFIER> | Literal )
               CastExpression::= ( "(" Type ")" UnaryExpression | "(" Type
                                   ")" UnaryExpressionNotPlusMinus )
            PrimaryExpression::= PrimaryPrefix ( ( PrimarySuffix ) )*
                PrimaryPrefix::= Literal
                                | "(" Expression ")"
                                | ObjectName
                                | <IDENTIFIER>
                PrimarySuffix::= ( "[" Expression "]" | "." <IDENTIFIER> |
                                   Arguments )
```

```
Literal::= ( <INTEGER_LITERAL> )
        | ( <FLOATING_POINT_LITERAL> )
        | ( <CHARACTER_LITERAL> )
        | ( <STRING_LITERAL> )
        | ( "true" )
        | ( "false" )
        | ( "null" )
Arguments::= "(" ( ArgumentList )? ")"
ArgumentList::= Expression ( "," Expression )*
```

Appendix D

MEDL Grammar

BNF for medl.jj

NON-TERMINALS

```
                setRelOp::= java code
              setArithOp::= java code
         addToSymbolTable::= java code
      searchSymbolTable::= java code
             addVarToList::= java code
         addToVarBackPtrs::= java code
            error_skipto::= java code
       RequirementScript::= <REQSPEC> <IDENTIFIER> Statements Guards
                            <END>
               Statements::= ( ( Statement ) ( ";" )? )*
                Statement::= EventDeclaration
                           | ConditionDeclaration
                           | ActionDeclaration
                           | AuxilliaryVariableDeclaration
                           | EventDefinition
                           | ConditionDefinition
                           | SafetyPropertyDefinition
                           | AlarmDefinition
        EventDeclaration::= <IMPORT> <EVENT> IdentifierList
    ConditionDeclaration::= <IMPORT> <CONDITION> IdentifierList
       ActionDeclaration::= <IMPORT> <ACTION> IdentifierList
          IdentifierList::= <IDENTIFIER> ( "," <IDENTIFIER> )*
AuxilliaryVariableDeclaration::= <AUXVAR> PrimitiveType IdentifierList
         EventDefinition::= <EVENT> <IDENTIFIER> "=" EventExpression
     ConditionDefinition::= <CONDITION> <IDENTIFIER> "="
                            ConditionalExpression
SafetyPropertyDefinition::= <PROPERTY> <IDENTIFIER> "="
                            ConditionalExpression
```

```
      AlarmDefinition::= <ALARM> <IDENTIFIER> "=" EventExpression
      EventExpression::= ( SimpleEventExpression ( ( EventOp
                         EventExpression ) )? OptionalWhen )
         OptionalWhen::= ( <WHEN> ConditionalExpression )*
SimpleEventExpression::= ( <IDENTIFIER> | "(" EventExpression ")"
                         | <START> "(" ConditionalExpression ")" |
                         <END> "(" ConditionalExpression ")" )
              EventOp::= ( "&&" )
                       | ( "||" )
               Guards::= ( Guard )*
                Guard::= <IDENTIFIER> "->" "{" Updates "}"
              Updates::= Update ( ";" Update )* ";"
               Update::= ( ( <PRIMEDID> "=" Expression ) | (
                         <INVOKE> <IDENTIFIER> ) | ( <PRINT>
                         Expression ) )
                 Type::= ( ( PrimitiveType | Name ) ( "[" "]" )* )
        PrimitiveType::= ( "boolean" )
                       | ( "char" )
                       | ( "byte" )
                       | ( "short" )
                       | ( "int" )
                       | ( "long" )
                       | ( "float" )
                       | ( "double" )
           ResultType::= ( "void" )
                       | ( Type )
                 Name::= ( <IDENTIFIER> ( "." ( <IDENTIFIER> ) )*
                         )
           Expression::= ConditionalExpression
ConditionalExpression::= ( ConditionalOrExpression ( "?"
                         Expression ":" ConditionalExpression )? )
ConditionalOrExpression::= ( ConditionalAndExpression ( "||"
                         ConditionalAndExpression )* )
ConditionalAndExpression::= ( InclusiveOrExpression ( "&&"
                         InclusiveOrExpression )* )
 InclusiveOrExpression::= ( ExclusiveOrExpression ( "|"
                         ExclusiveOrExpression )* )
 ExclusiveOrExpression::= ( AndExpression ( "^" AndExpression )* )
        AndExpression::= ( EqualityExpression ( "&"
                         EqualityExpression )* )
   EqualityExpression::= ( "[" EventExpression "," EventExpression
                         ")" )
                       | ( <DEFINED> "(" Expression ")" )
                       | InstanceOfExpression ( ( ( "==" | "!=" )
                         InstanceOfExpression ) )*
 InstanceOfExpression::= ( RelationalExpression ( "instanceof"
```

```
                              Type )? )
         RelationalExpression::= ShiftExpression ( ( ( "<" | ">" | "<=" |
                              ">=" ) ShiftExpression ) )*
              ShiftExpression::= AdditiveExpression ( ( ( "<<" | ">>" |
                              ">>>" ) AdditiveExpression ) )*
           AdditiveExpression::= MultiplicativeExpression ( ( ( "+" | "-"
                              ) MultiplicativeExpression ) )*
     MultiplicativeExpression::= UnaryExpression ( ( ( "*" | "/" | "%" )
                              UnaryExpression ) )*
              UnaryExpression::= ( ( "+" | "-" ) UnaryExpression )
                              | UnaryExpressionNotPlusMinus
                              | ( <TIME> "(" EventExpression ")" )
                              | ( <VALUE> "(" EventExpression ","
                              Expression ")" )
UnaryExpressionNotPlusMinus::= ( ( "~" | "!" ) UnaryExpression )
                              | CastExpression
                              | PrimaryExpression
               CastLookahead::= "(" PrimitiveType
                              | "(" Name "[" "]"
                              | "(" Name ")" ( "~" | "!" | "(" |
                              <IDENTIFIER> | Literal )
              CastExpression::= ( "(" Type ")" UnaryExpression | "(" Type
                              ")" UnaryExpressionNotPlusMinus )
           PrimaryExpression::= PrimaryPrefix ( PrimarySuffix )*
               PrimaryPrefix::= Literal
                              | "(" Expression ")"
                              | ResultType "." "class"
                              | Name
                              | PrimedName
                  PrimedName::= <PRIMEDID>
               PrimarySuffix::= "[" Expression "]"
                              | "." <IDENTIFIER>
                              | Arguments
                     Literal::= ( <INTEGER_LITERAL> )
                              | ( <FLOATING_POINT_LITERAL> )
                              | ( <CHARACTER_LITERAL> )
                              | ( <STRING_LITERAL> )
                              | ( "true" )
                              | ( "false" )
                              | ( "null" )
                   Arguments::= "(" ( ArgumentList )? ")"
                ArgumentList::= Expression ( "," Expression )*
```

Appendix E

MaCSware User Manual

Preface

This document presents the user interface of the monitoring and checking (MaC) toolset. It describes the process of putting together a monitoring application from a system to be monitored and a set of scripts that describe what should be monitored and how. The language of the scripts is beyond the scope of this manual. Consult the MaC language reference manual for syntax and semantics of the languages used in MaC. The rationale and the general philosophy for MaC, as well as a description of the components of MaC architecture can be found in [KKL⁺01, KVBA⁺99, KVBA⁺98]. The current version of MaC is targeted towards Java programs.

The manual is organized as follows: Section E.1 gives an overview of MaC architecture. It presents the components of the architecture and the steps necessary to produce a monitoring application. Then, Section E.3 describes the graphical user interface for MaC and its use to configure MaC for a sample application. Finally, Section E.4 gives the command-line interface to all MaC components.

E.1 MaC overview

The structure of the MaC architecture is demonstrated in Figure E.1. The user specifies the requirements of the system in a formal language. Requirements are expressed in terms of high-level events and conditions. In addition, a *monitoring script* relates these events and conditions with low-level data manipulated by the system at run time. Based on the monitoring script, the system is *automatically* instrumented to deliver the monitored data to the *event recognizer*. The event recognizer, also generated from the monitoring script, transforms this low-level data into abstract events and delivers them to the run-time checker. The run-time checker is generated from the requirements specification. The run-time checker verifies the sequence of abstract events with respect to the requirements specification and detects violations of requirements.

The reason for keeping the monitoring script distinct from the requirements specification is to maintain a clean separation between the system itself, implemented in a certain way, and high-level system requirements, independent of a concrete implementation. Implementation-dependent event recognition insulates the requirement checker from
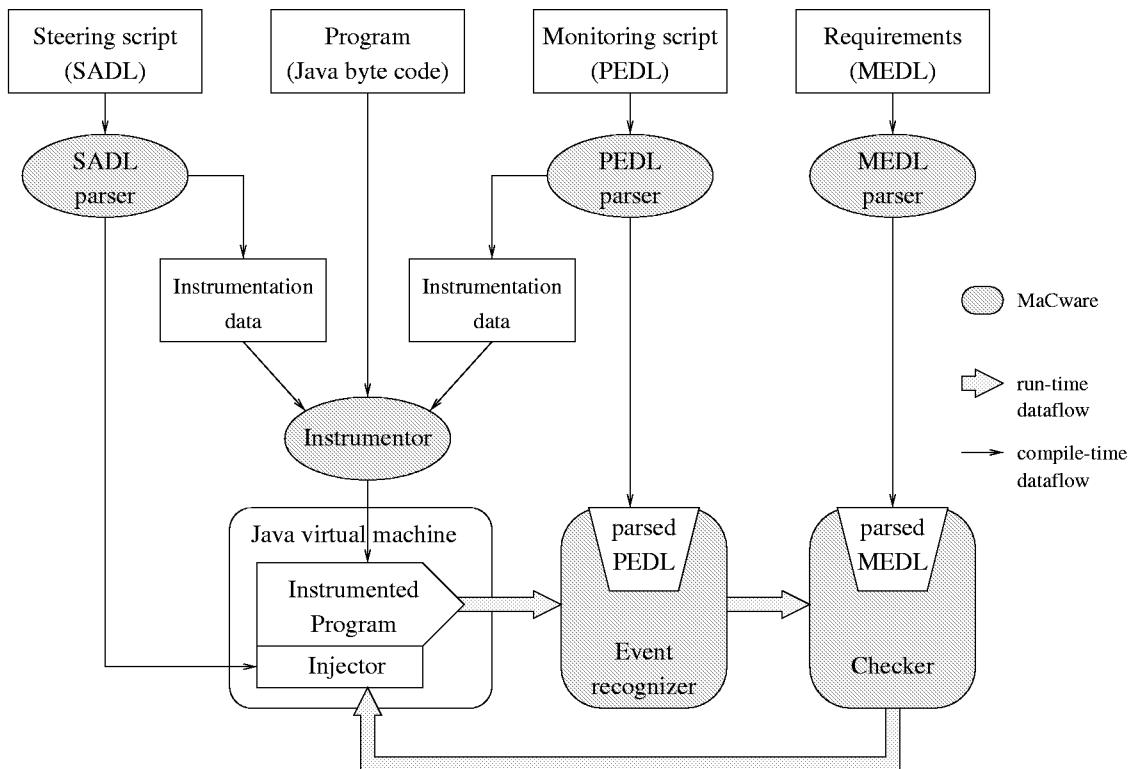
131

Figure E.1: MaC architecture

Diagram labels:

- Steering script (SADL)
- Program (Java byte code)
- Monitoring script (PEDL)
- Requirements (MEDL)
- SADL parser
- PEDL parser
- MEDL parser
- Instrumentation data
- Instrumentation data
- Instrumentor
- Java virtual machine
- Instrumented Program
- Injector
- parsed PEDL
- Event recognizer
- parsed MEDL
- Checker
- MaCware
- run-time dataflow
- compile-time dataflow

the low-level details of the system implementation. This separation also allows us to perform monitoring of heterogeneous distributed systems. A separate event recognizer may be supplied for each module in such system. Each event recognizer may process the low-level data in a way specific to the respective module. For example, an event recognizer that is associated with a software component will work very differently from the one that processes traffic on a bus. But all event recognizers deliver high-level events to the checker in a uniform fashion.

In keeping with this design philosophy, two languages have been designed for use in the MaC architecture. The Meta-Event Definition Language (MEDL) is used to express requirements. It is based on an extension of a linear-time temporal logic. It allows to express a large subset of safety properties of systems, including real-time properties. Monitoring scripts are expressed in the Primitive Event Definition Language (PEDL). PEDL describes primitive high-level events and conditions in terms of system objects. PEDL, therefore, is tied to the implementation language of the monitored system in the use of object names and types. MEDL is independent of the monitored system.

In addition, when steering is used, the user provides a steering script in addition to the monitoring script. The steering script describes steering actions and conditions for their invocation using the Steering Action Definition Language (SADL). Steering actions are invoked in response to requirement violations detected by the checker. The steering script generates additional instrumentation data for the system and also a special run-time component called *injector* that accepts action invocations from the monitor and triggers their execution within the system.

Thus, in order to set up monitoring of a system, the user has to perform the following steps:

1. Specify the monitoring script (PEDL) and compile it to generate the event recognizer.

2. Specify the requirements specification (MEDL) and compile it to generate the run-time checker.

3. Optionally, specify the steering script (SADL) and compile it to generate the injector.

4. Perform instrumentation of the system.

5. Start the runtime checker.

6. Start the event recognizer.

7. Start the system.

Note that the order in which MaC components are started, is important: the checker should be run before the event recognizer and both of them must be running before the system is started.

E.2 Installation of MaCSware

MaCSware[1] is written in Java and runs on version 1.1 or later version of java platform. MaCSware uses JTREK library v1.1 for bytecode modification. Thus, user need to download JTREK v1.1[2] and bug patch of JTREK v1.1.[3] After downloading MaCSware into <mac_dir> and JTREKv1.1 into <jtrek_dir>, user need to set CLASSPATH. We assume that CLASSPATH variable already exists.

- For Unix user using tcsh, add

    ```
    setenv CLASSPATH $CLASSPATH":<mac_dir>:<jtrek_dir>:."
    ```

    at the last line of <home_dir>/.tcshrc. Then source <home_dir>/.tcshrc

- For Windows user, open Start button → Setting → Control Panel → System (→ Advanced) → Environment and add

    ```
    ;<mac_dir>;<jtrek_dir>;.
    ```

    into the value field of the variable CLASSPATH. Or, add

    ```
    set classpath=%classpath%;<mac_dir>;<jtrek_dir>;.
    ```

    at the last line of C:\autoexec.bat. Then restart a computer.

E.3 MaC GUI

The control panel for MaC is provided by the MacGUI application. It visually represents all MaC components that need to be set up in order to do monitoring, and guides the workflow by enforcing the dependencies between the components. For example, the checker cannot be set up before the user enters and compiles the requirements specification to be input to the checker.

Figure E.2 shows the view of the MaC control panel. It has been intentionally made to follow the layout of Figure E.1. Each component of the MaC architecture is represented as a box in the control panel. Components that are ready to be configured are enabled. When the user selects an enabled component with the mouse, a dialog comes up that allows the user to configure it. When the configuration of the component is completed, other components that depend on it become enabled.

There are three script boxes that allow the user to edit the monitoring and steering scripts and the requirements specification. When the user selects a script box, an editor window comes up, where the user can enter the script or load it from a file. Alternatively, a script is specified when the corresponding compiler box is enabled, through a file dialog.

A dialog associated with the program box allows the user to specify a semicolon-separated list of class names that need to be instrumented. The instrumentation box

---

[1] http://www.cis.upenn.edu/~rtg/mac/macsware.zip

[2] http://www.digital.com/java/download/jtrek/index.html

[3] Patch is available at MaCSware distribution (http://www.cis.upenn.edu/~rtg/mac/patch.zip). User should unzip patch.zip in <jtrek_dir>/dec/trek/
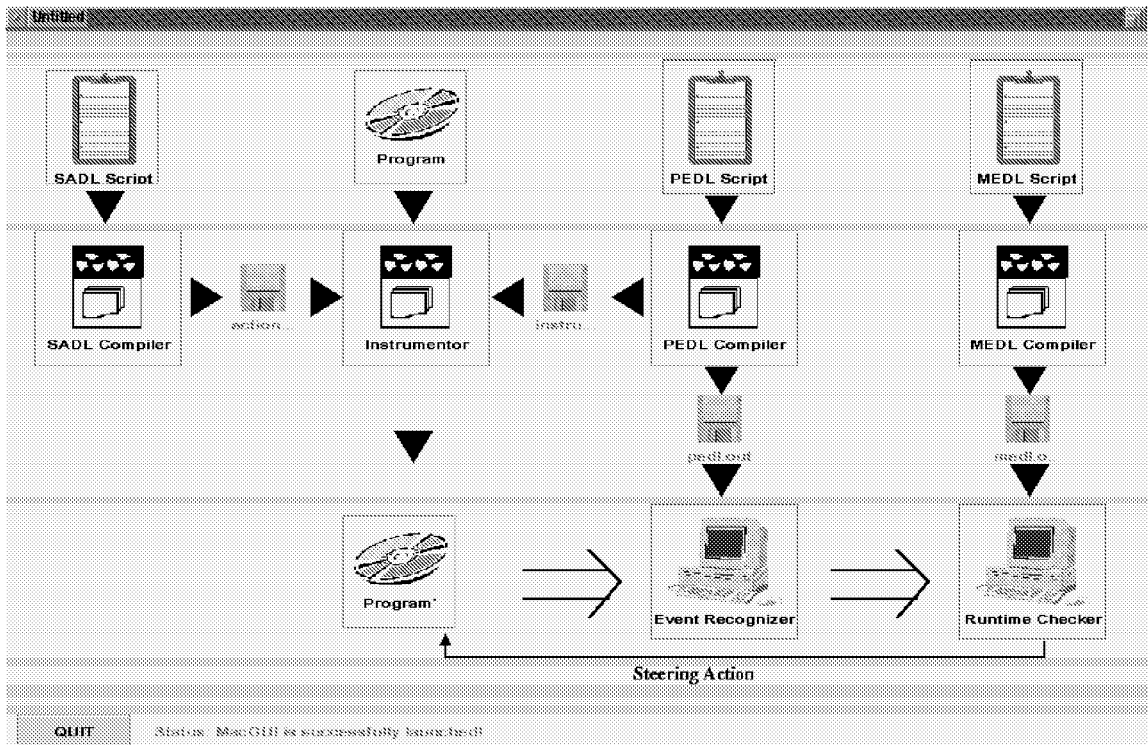
Figure E.2: MaC control panel

is activated after the class list has been given and the monitoring script has been compiled. The dialog of the instrumentation box allows the user to specify the media for communication with the event recognizer and the checker. Although MaC interaction, in general, can be done either through TCP/IP among different hosts or via FIFOs, the control panel allows only TCP/IP communication on the same host where the control panel runs. Therefore, the user has to specify the port address of the event recognizer and, if steering is used, the port number to listen for steering action invocations is also required. This data is inserted by the instrumentor into the initialization code of the instrumented program.

The bottom row of boxes starts the program and the monitoring components. The dialogs for the event recognizer lets the user specify the port number of the checker and the port number to receive data from the program. The checker dialog accepts the port number for messages from the event recognizer and, in the case of steering, the address of the injector. Be sure that the port number used by the event recognizer and the runtime checker must not be used by other programs.

Starting the program, the user specifies the class file where the execution starts and gives the run-time parameters of the program. All monitoring-related information has already been specified during instrumentation.

Finally, the bottom line contains the quit button and shows the status of MaC activity.

135

Starting the control panel. The MaC control panel is a Java application that is started through class `mac.gui.guiLauncher.MacGUI` with no parameter.[4]

E.4 Command-line MaC interface

All MaC components can be started from command line. Command-line interface provides larger flexibility than the control panel. Below, we give the executable class name of every component and parameters that the component can accept, as well as the output produced by each component. All MaC components are Java appplications.

MEDL compiler. The executable class is
> `mac.runtimeChecker.medlParser.MedlParser`

and can take a single argument, the file containing a script to compile. If no argument is given, the parser reads from standard input. The compiled script is stored in the file `medl.out`.

PEDL compiler. The executable class is
> `mac.eventRecognizer.pedlParser.PedlParser`

and can take a single argument, the file containing a script to compile. If no argument is given, the parser reads from standard input. The compiled script is stored in the file `pedl.out`, which is used by the event recognizer. In addition, instrumentation data, used by the instrumentor, are stored in the file `instrumentation.out`.

SADL compiler. The executable class is
> `mac.steering.parser.SadlParser`

and can take a single argument, the file containing a script to compile. If no argument is given, the parser reads from standard input. The compiled script is stored in the file `actions.out`. In addition, the compiler produces and stores in the current directory a Java source file named `foo_Injector.java`, where `foo` is the name of the steering script. The file must be compiled into byte code, which will be loaded into the JVM together with the monitored program.

Instrumentor. The instrumentor application is used when there is no steering involved.[5]
> `mac.filter.instrumentor.Instrumentor`

and the following arguments are required in the order shown:

- the list of monitored objects, produced by the PEDL parser as `instrumentation.out`.

- a semicolon-separated list of class files to be instrumented. Class files are assumed to be in the current directory.

---

[4]MacGUI requires Swing 1.1

[5]The instrumentor generates a policy file `mac_policy` which allows communication from a target program to the event recognizer. See more detail in `http://java.sun.com/products/jdk/1.3/docs/guide/security/PolicyFiles.html`

- The address of the event recognizer or the name of a file or FIFO where the monitored data will be output.

If steering is to be used, a different instrumentation tool is used, located in mac.steering.Steer Steering instrumentor allows to specify its arguments in any order with the following keys:

- -monitor denotes that the following argument is the monitoring instrumentation data, that is, the list of monitored objects.

- -steer denotes that the following argument is the steering instrumentation data, steering actions and positioning information.

- -ports denotes that the following two arguments are the address of the event recognizer and the port number for action invocations, in that order. When steering is used, the MaC run-time components have to communicate through TCP/IP.

- -classfiles denotes that the following argument is the list of classfiles to be instrumented.

Run-time checker. The executable class is
<div align="center">mac.runtimeChecker.interpreter.CheckerMain.</div>
The first argument specifies the input stream for the checker. It is a port number on which to listen for connection requests from the event recognizer or, if preceded by the -f key, the file or pipe name from which event recognizer messages are read. The second, optional argument, denoted by -s key, denotes that steering will be used in this run and specifies the address of the system to be steered in the form hostname:port. The hostname can be omitted, in which case the system is assumed to be executed on the same machine as the checker. The checker then establishes a connection to the injector during start-up. The last argument is the compiled MEDL script produced by the MEDL compiler.

Event recognizer. The executable class is
<div align="center">mac.eventRecognizer.interpreter.EventRecognizerMain.</div>
The following arguments are required in the order shown:

- the port number for the monitored data sent by the instrumented program, or a file name preceded by the -f key.

- the address of the checker (hostname may be omitted), or a file name preceded by the -f key.

- the compiled monitoring script (pedl.out).

- the list of monitored objects (instrumentation.out).

Instrumented program. The program is started in the same way as it would without monitoring. All data necessary for communication with MaC components have been given to the program during instrumentation.