

Information Extraction for Run-time Formal Analysis

Moonjoo Kim

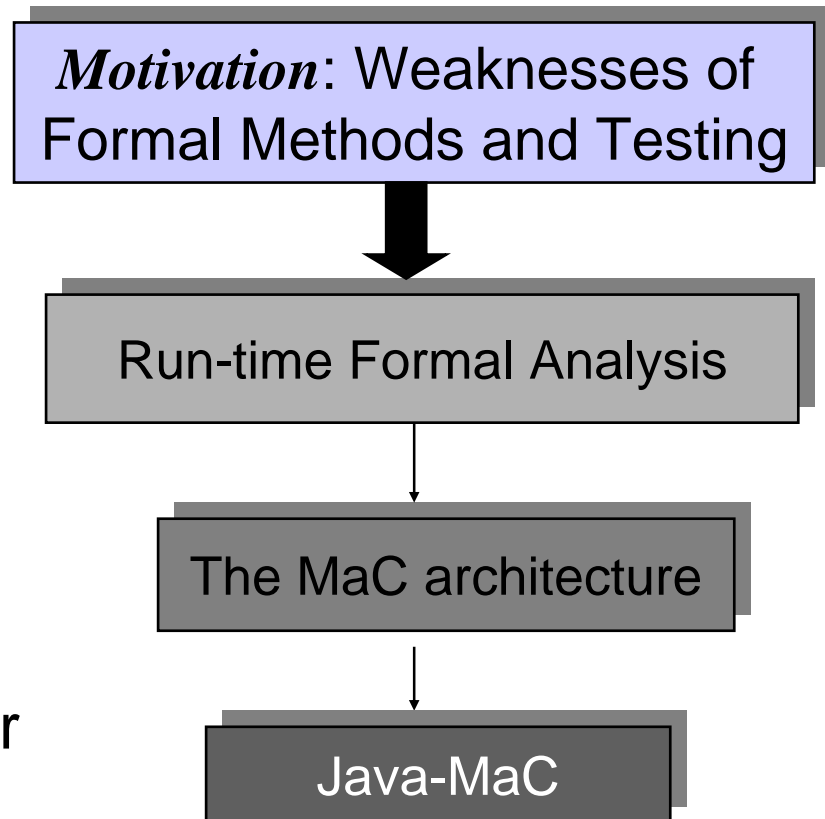
Advisors: Prof. Kannan and Prof. Lee

CIS Department

University of Pennsylvania

Outline

- WHY?
 - Motivation
- WHAT?
 - Run-time Formal Analysis
- HOW?
 - High-level: the Monitoring and Checking (MaC) Architecture
 - Low-level: a MaC Prototype for Java programs (Java-MaC)

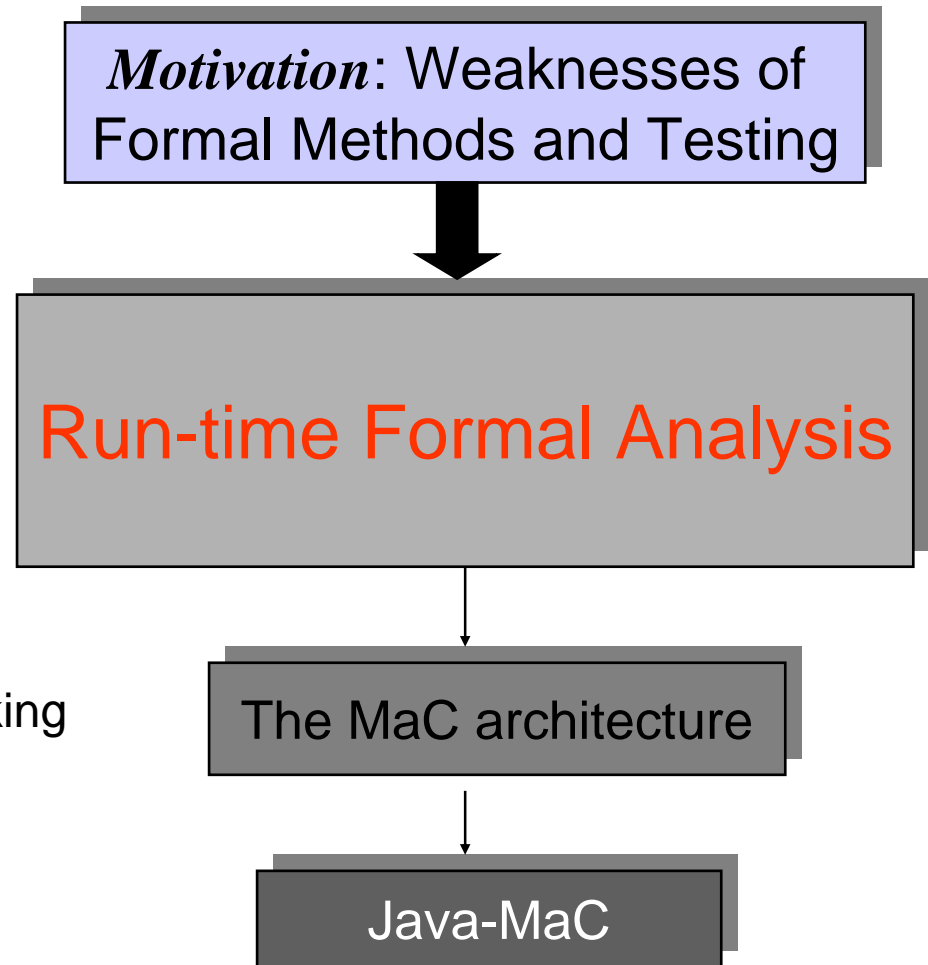


Motivation

- Weaknesses of formal verification and testing
 - formal verification:
 - gap between an abstract model and the implementation
 - lack of scalability
 - testing:
 - lack of complete guarantee

Outline

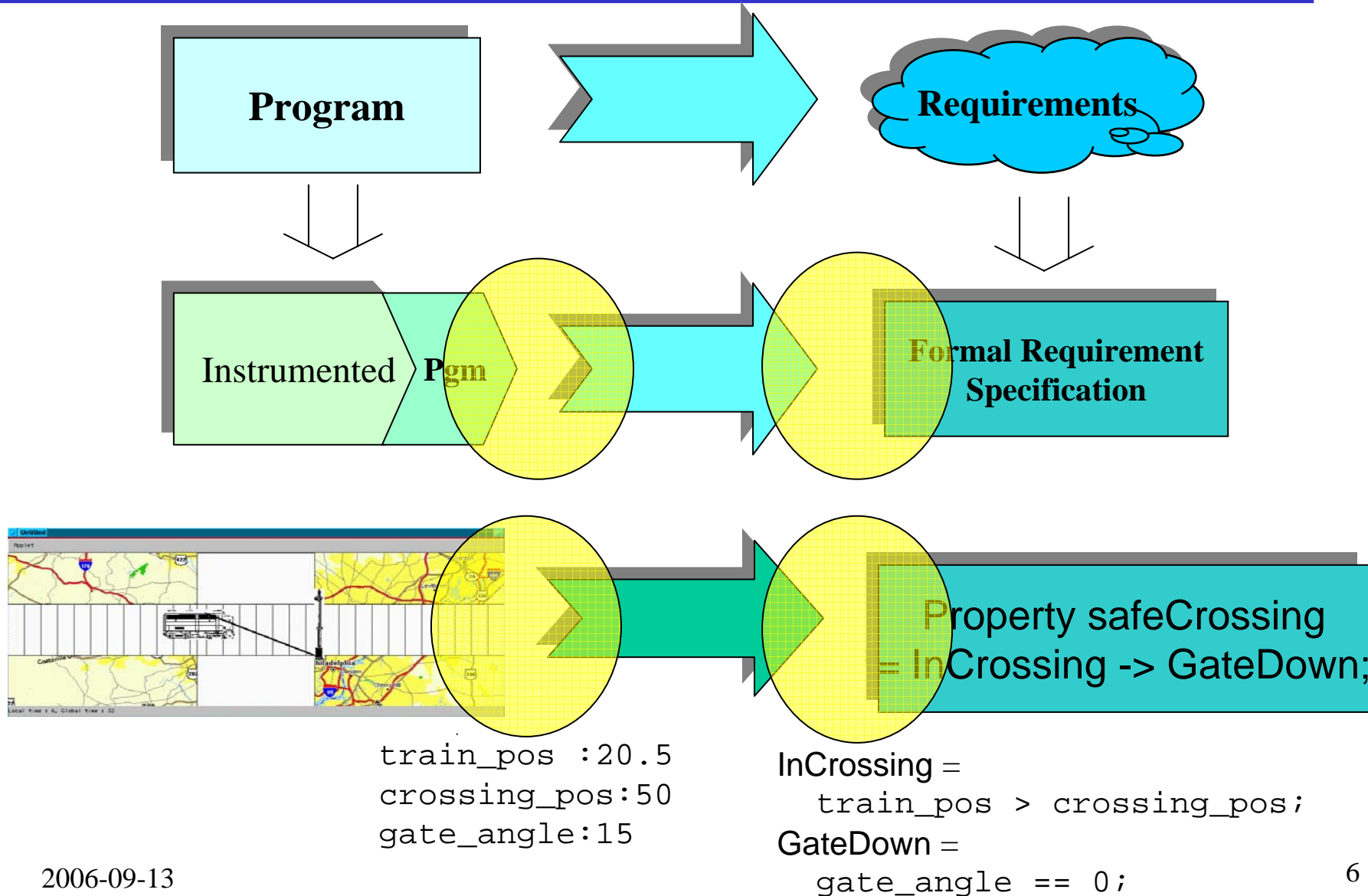
- WHY?
 - Motivation
- **WHAT?**
 - **Run-time Formal Analysis**
- HOW?
 - High-level: The Monitoring and Checking (MaC) Architecture
 - Low-level: a MaC Prototype for Java programs
- Summary



Run-time Formal Analysis

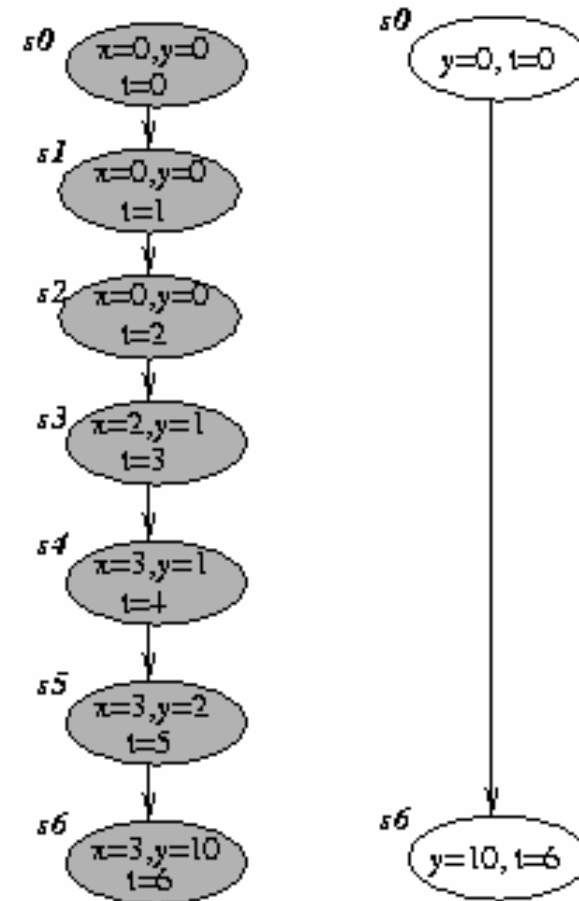
- Motivation:
 - Run-time correctness is not guaranteed
- The goal of run-time formal analysis
 - to give confidence in the run-time compliance of an execution of a system w.r.t formal requirements
- The analysis validates properties on the *current* execution of application.
- Run-time formal analysis helps user to detect errors and prevent system crash.

Relation Between Execution and Requirements



Program Execution

- A program execution σ is a sequence of states $s_0s_1\dots$
 - A state s consists of
 - an environment $\rho_s: V \rightarrow R$
 - a timestamp t_s s.t. $t_{s_j} < t_{s_{j+1}}$
- We may abstract out state information unnecessary to detect requirements.



property $p =$

$$3 < y \ \&\& \ y < 11 \quad 7$$

Outline

- WHY?
 - Motivation
- WHAT?
 - Run-time Formal Analysis
- HOW?
 - **High-level:**
 - **the Monitoring and Checking (MaC) Architecture**
 - Low-level: a MaC Prototype for Java programs
- Summary

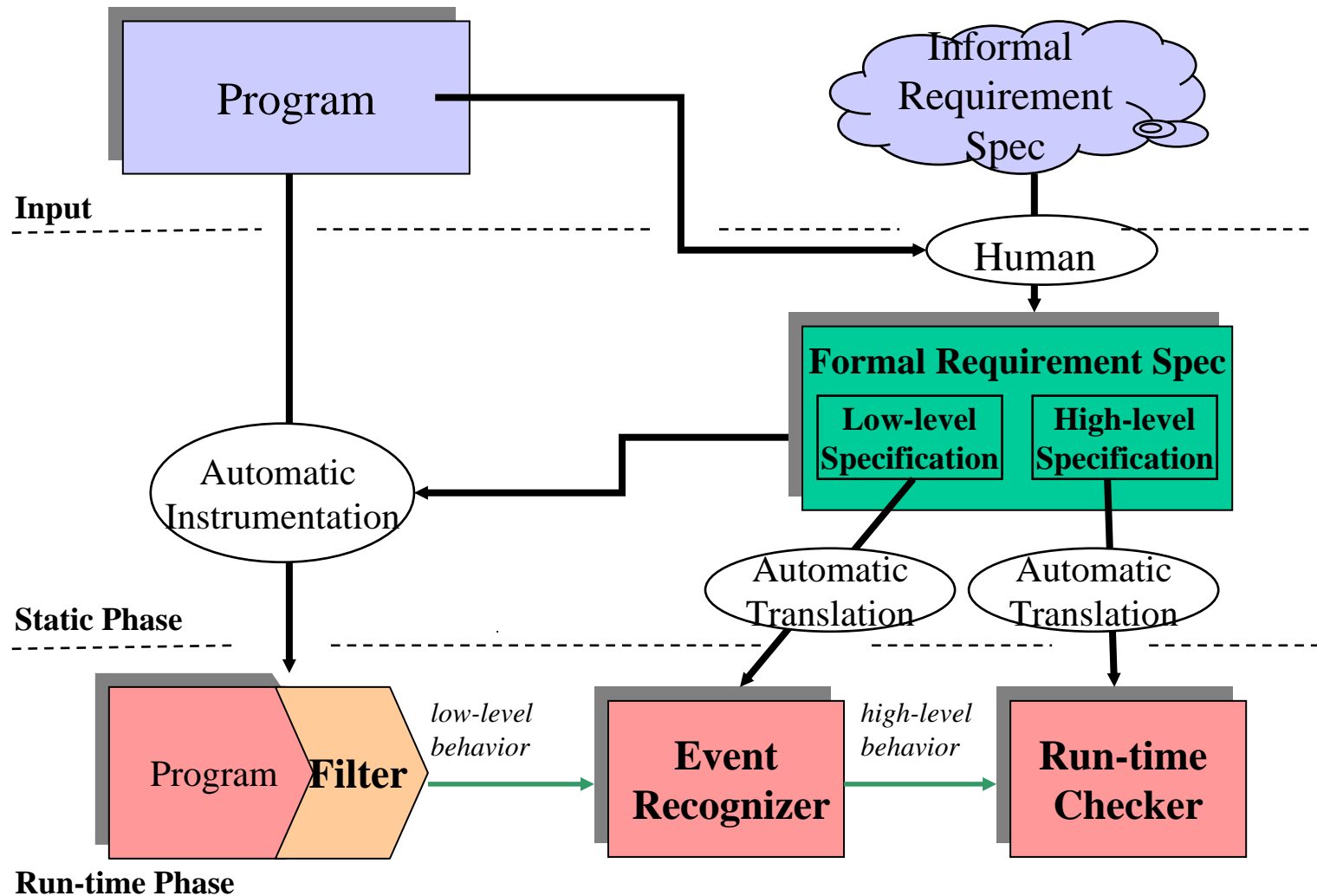
Motivation: Weaknesses of Formal Methods and Testing

Run-time Formal Analysis

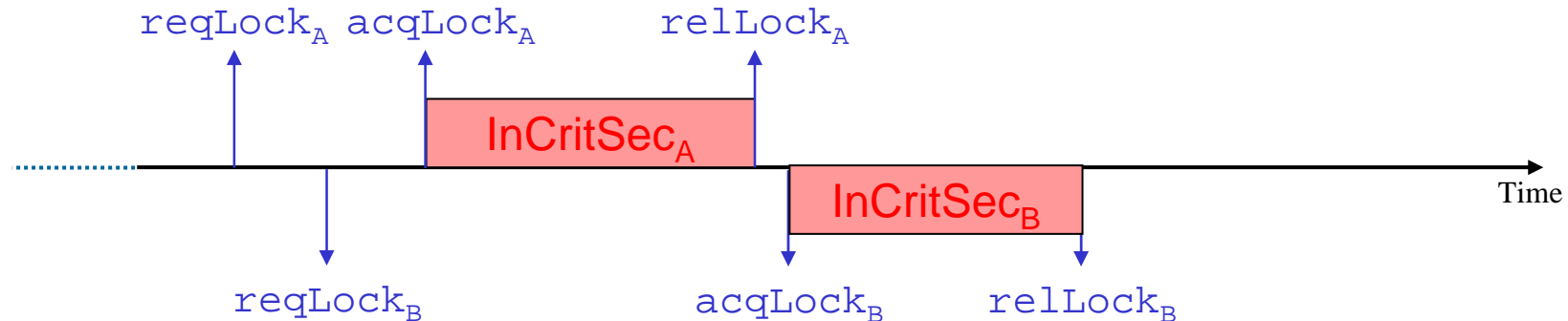
The MaC architecture

Java-MaC

Overview of the MaC Architecture



Design of the MaC Languages



- Must be able to reason about both **time instants** and information that holds for a **duration of time** in a program execution.
- Need temporal operators combining events and conditions in order to reason about traces.

Logical Foundation

$$C ::= c / \text{defined}(C) \mid [E_1, E_2) \mid \neg C \mid C_1 \vee C_2 \mid C_1 \wedge C_2$$
$$E ::= e \mid \text{start}(C) \mid \text{end}(C) \mid E_1 \vee E_2 \mid E_1 \wedge E_2 \mid$$
$$E \text{ when } C$$

- conditions interpreted over 3 values
 - true, false and undefined.
- $[\cdot, \cdot)$ pairs a couple of events to define an interval.
- start and end define the events corresponding to the instant when conditions change their value.

The MaC Languages

- Meta Event Definition Language(MEDL)
 - Describes the *safety requirements* of the system, in terms of **conditions** that must always be true, and *alarms* (**events**) that must never be raised.
 - Target program implementation independent.
- Primitive Event Definition Language (PEDL)
 - Defines primitive events/conditions in terms of program entities
 - Provides primitives to refer to values of variables and to certain points in the execution of the program.
 - Depends on target program implementation

Meta Event Definition Language (MEDL)

- Expresses requirements using the events and conditions
- Expresses the subset of safety languages.
- Describes the *safety requirements* of the system
 - property **safeRRC** = IC -> GD;
 - alarm **violation** = start (!safeRRC);
- *Auxiliary variables* may be used to store history.
 - endIC-> { num_train_pass' =
 num_train_pass + 1; }

```
ReqSpec <spec_name>

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

/*Auxiliary variable */
var int <aux_v>;

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

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

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

End
```

Outline

- WHY?
 - Motivation
- WHAT?
 - Run-time Formal Analysis
- HOW?
 - High-level: The Monitoring and Checking (MaC) Architecture

–Low-level: a MaC
Prototype for Java
programs

- Summary

Motivation: Weaknesses of
Formal Methods and Testing

Run-time Formal Analysis

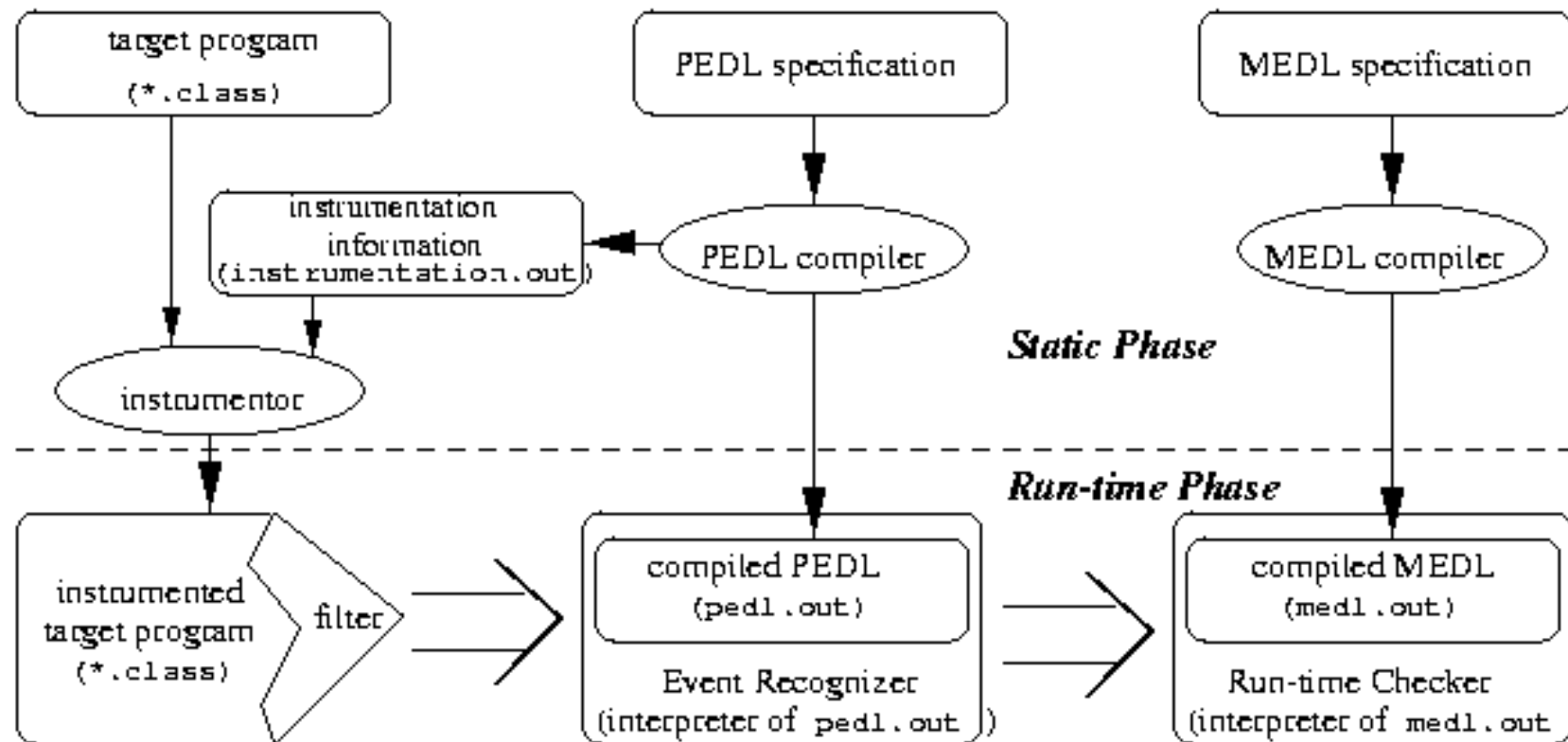
The MaC architecture

Java-MaC

Java-MaC

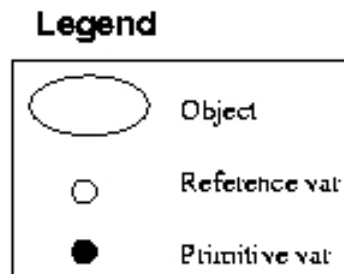
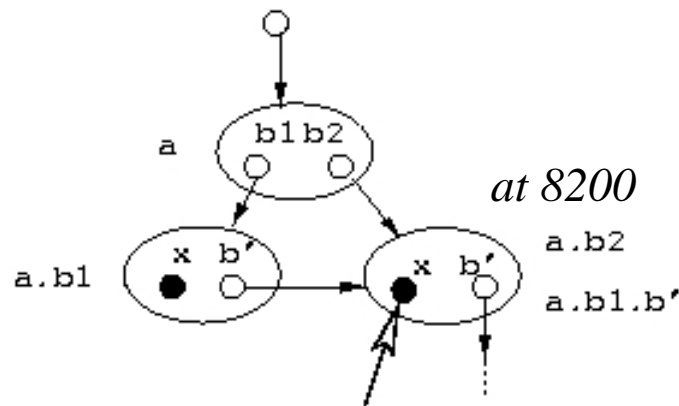
- Overview of Java-MaC
- Monitoring Java programs
 - Monitoring objects
 - PEDL for Java
- Static components
 - Instrumentor, PEDL/MEDL compilers
- Run-time components
 - Filter, event recognizer, run-time checker
- Overhead reduction
- Case study

The MaC Prototype for Java Programs



Monitoring Objects

- Specifying monitored objects
 - There can be several instances (objects) of the same class.
- Monitoring objects
 - A monitored object can be updated by several references.
- To test references, we need a globally accessible table (*address table*) containing pairs of *addresses* of monitored objects and monitored object *names*
 - Assumption: *no* primary reference to a monitored object is changed



Address Table

Address	Var Name
8200	a.b2

PEDL for Java

- Provides primitives to refer to
 - primitive variables
 - beginnings/endings of methods
- Primitive conditions are constructed from
 - boolean-valued expressions over the monitored variables
 - ex> condition IC =
(position == 100);
- Primitive events are constructed from
 - update(x)
 - startM(f)/endM(f)
 - ex>event raiseGate=
startM(Gate.gu());

```
MonScr <spec_name>
  /* Export section */
  export event <e>;
  export condition <c>;

  /* Monitored entities */
  monobj <var>;
  monmeth <meth>;

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

End
```

PEDL for Java (*cont.*)

- Events can have two attributes - time and value
- $\text{time}(e)$ gives the time of the last occurrence of event e
 - used for expressing temporal properties
- $\text{value}(e,i)$ gives the i th value in the tuple of values of e
 - value of $\text{update}(\text{var})$: a tuple containing a current value of var
 - value of $\text{startM}(f)$: a tuple containing parameters of the method f
 - value of $\text{endM}(f)$: a tuple containing parameters and a return value of the method f

Instrumentation

- Java-MaC instruments Java executable code
- Java-MaC instrumentor detects instructions
 - variable updates
 - putstatic/putfield for field variable updates
 - <T>store and iinc for local variable updates
 - execution points
 - instruction located at the beginning of method definition
 - return of method definition
- At the each detected instruction, Java-MaC instrumentor inserts a probe

Sample Probe

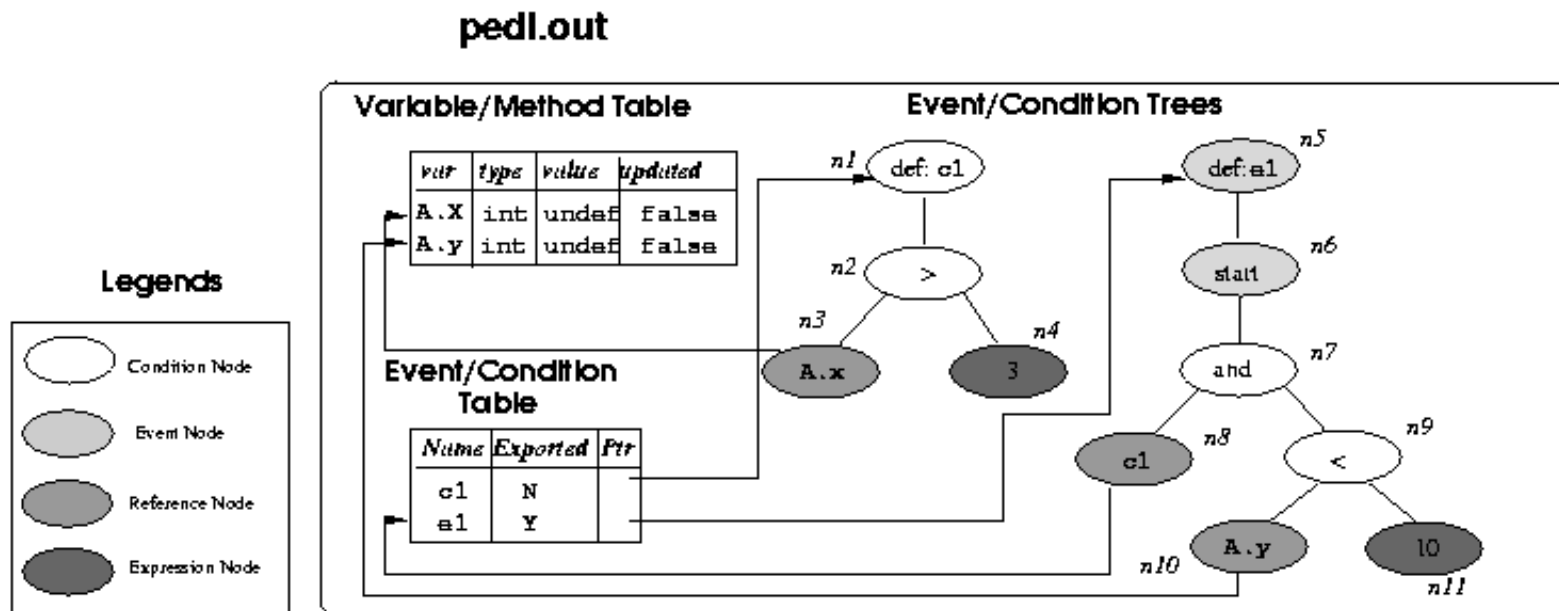
- Monitoring a field variable Var.val

```
; >> METHOD 8 <<      ; >> METHOD 8 <<
.method public run()V .method public run()V
  .limit stack 4      .limit stack 7
  .limit locals 2    .limit locals 2
  ...
  getfield DigitalVar.v I
  putfield Var.val I
  ...
.end Method

  ...
  getstatic mac.filter.Filter.lock Ljava.lang.Object;
  monitorenter
  dup2
  ldc "val"
  invokestatic mac.filter.SendMethods.sendObjMethod(
    Ljava/lang/Object;Ljava/lang/String;)V
  putfield Var.val I
  getstatic mac.filter.Filter.lock Ljava.lang.Object;
  monitorexit
  ...
.end Method
```

PEDL/MEDL Compilers

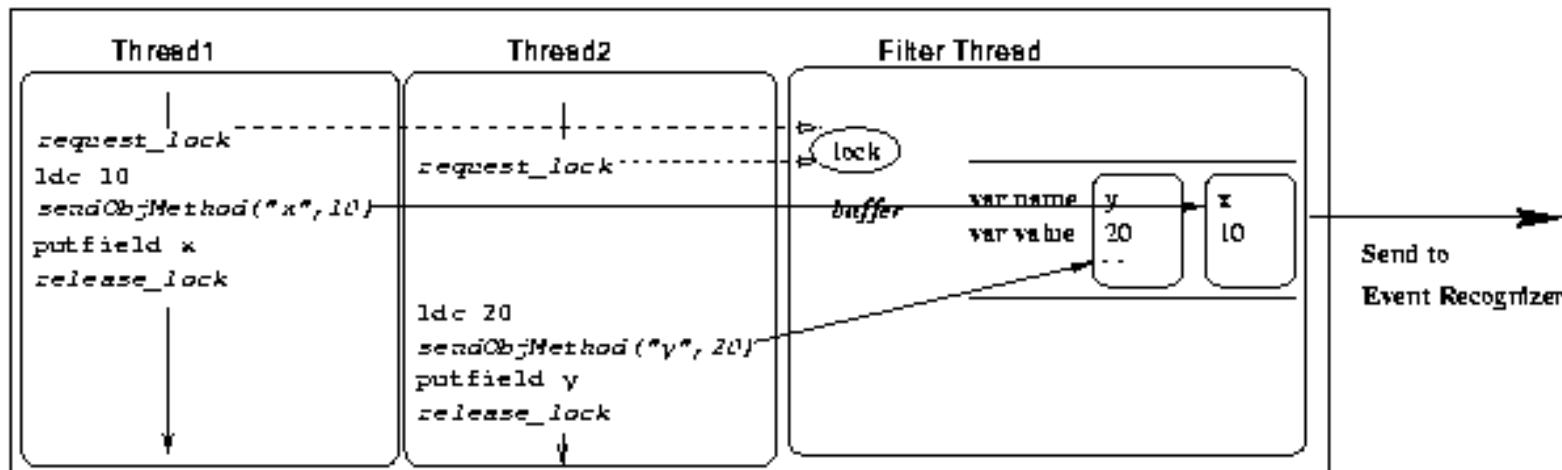
- Compiles PEDL/MEDL scripts into pedl.out/medl.out respectively
- Ex> condition c1 = A.x > 3;
event e1 = start(c1 && A.y < 10);



Filter

- A filter consists of
 - *a communication channel* to the event recognizer
 - *probes* inserted into the target system
 - *a filter thread* which flushes the content of communication buffers to the event recognizer
- Filter uses global lock for consistent snapshot ordering in spite of arbitrary preemption

Instrumented Target Program



Event Recognizer/Run-time Checker

- Event recognizer
 - evaluates `ped1.out` whenever it receives snapshots from the filter.
 - If an event or a condition changing its value is detected, the event recognizer sends the event or the condition to the run-time checker
- Run-time checker
 - evaluates `med1.out` whenever it receives events and conditions from the event recognizer.
 - detects a violation defined as alarm or property and raises a signal.

Reduce Overheads

- Less snapshot, less overhead
- Not every snapshot affects requirement properties
 - Evaluates *simple* expressions to check whether current snapshot *may* affect requirements

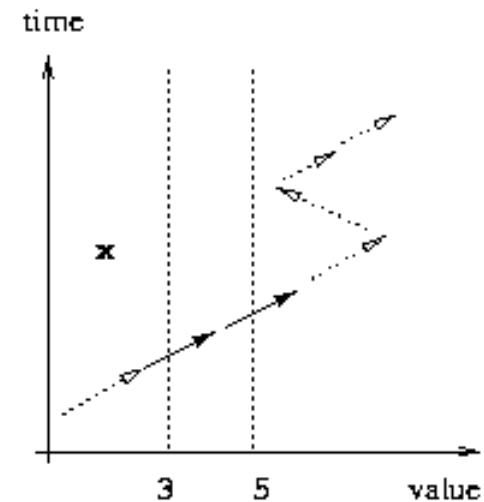
•Ex>

condition c1 =

$(3 < x \ \&\& \ x < 5) \ || \ y > 10;$

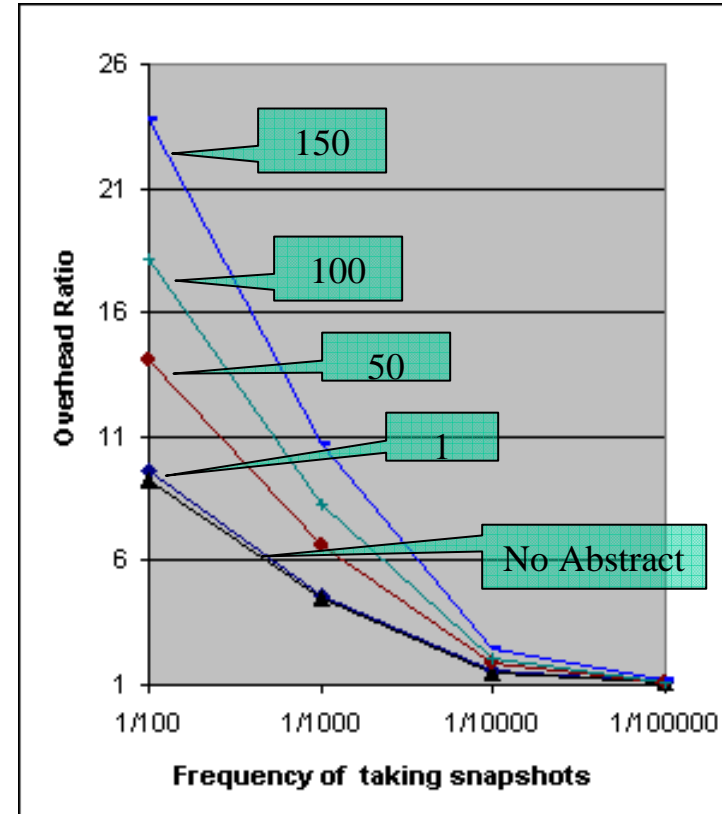
condition c2 = $w > z;$

property req = $c1 \rightarrow c2;$



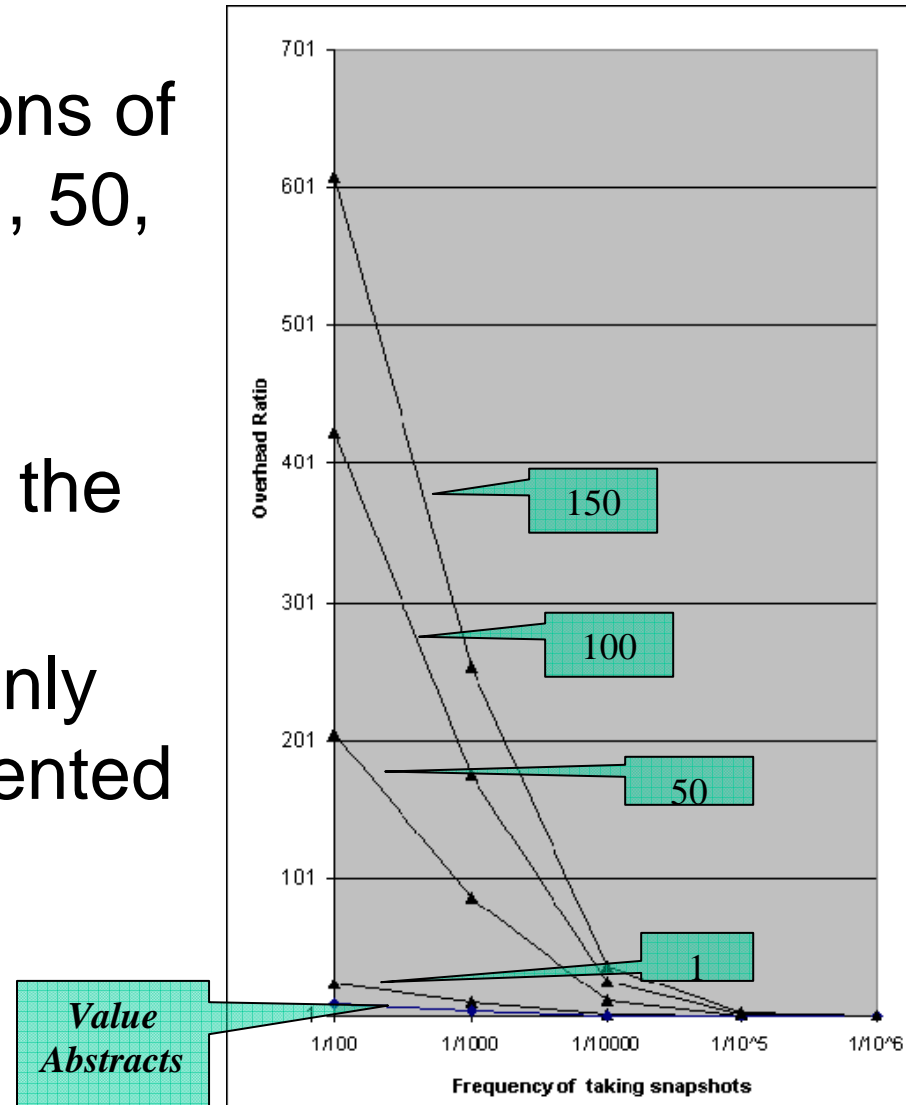
Probe Overhead

- Measure overhead over various frequency of updating a monitored integer variable by the target program
- Value abstraction with 1, 50, 150, 200 *simple expressions* to check



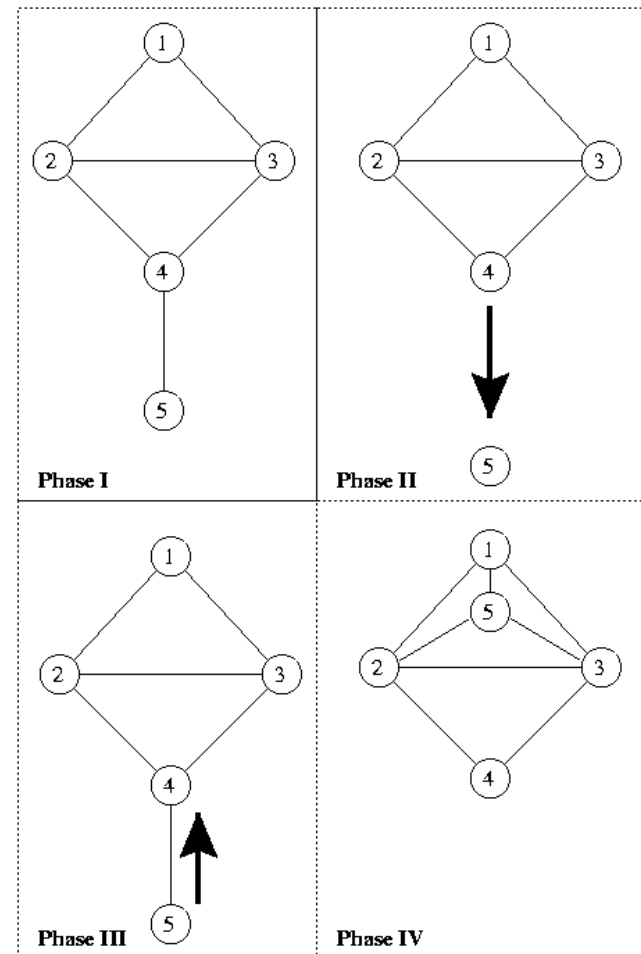
Overall Overhead

- Evaluating expressions of 4 different lengths (1, 50, 100, 150)
- Value abstraction significantly reduces the overhead
- The overhead is mainly due to the object-oriented implementation of `pedl.out`



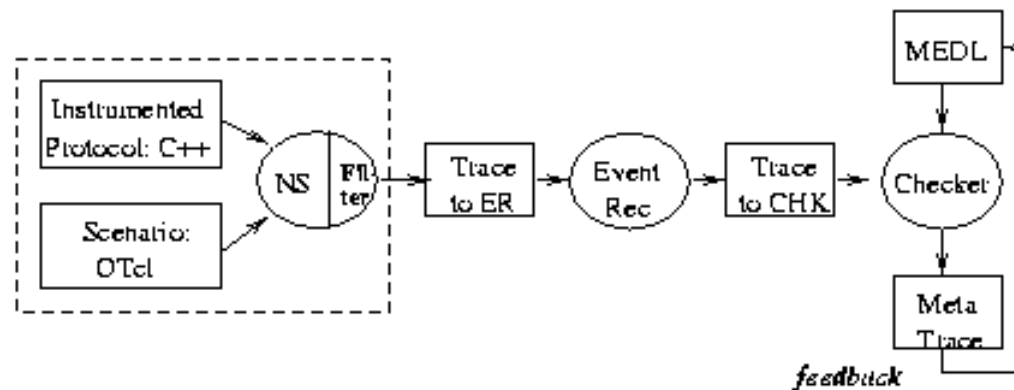
Case Study: Routing Protocol Validation

- Ad-hoc On Demand Vector (AODV) routing protocol used in packet radio networks consisting of mobile nodes
- Detect violations of properties such as loop invariant in AODV routing protocol implemented using NS2 simulator [Bhargavan,etc]



Case Study: Routing Protocol Validation (*cont.*)

- NS2 simulator is used instead of target Java program
- Execution trace containing packets delivered among nodes is analyzed repeatedly with different property descriptions without running the simulation again



Contributions

- Main contribution
 - Confirming the idea that run-time formal analysis can assure a user of the correctness of program execution in a *practical* manner through the implementation of the MaC architecture.
- Technical contributions
 - Rigorous analysis
 - Flexibility
 - Automation
 - Easy of use

Future Works

- Loosen the restriction on monitoring objects
 - Combined approach of instrumenting classfiles and modified Java virtual machine
- Apply value abstraction in more general way to gain the benefit of abstraction broadly
- Real-time extension of Java-MaC
- Application areas