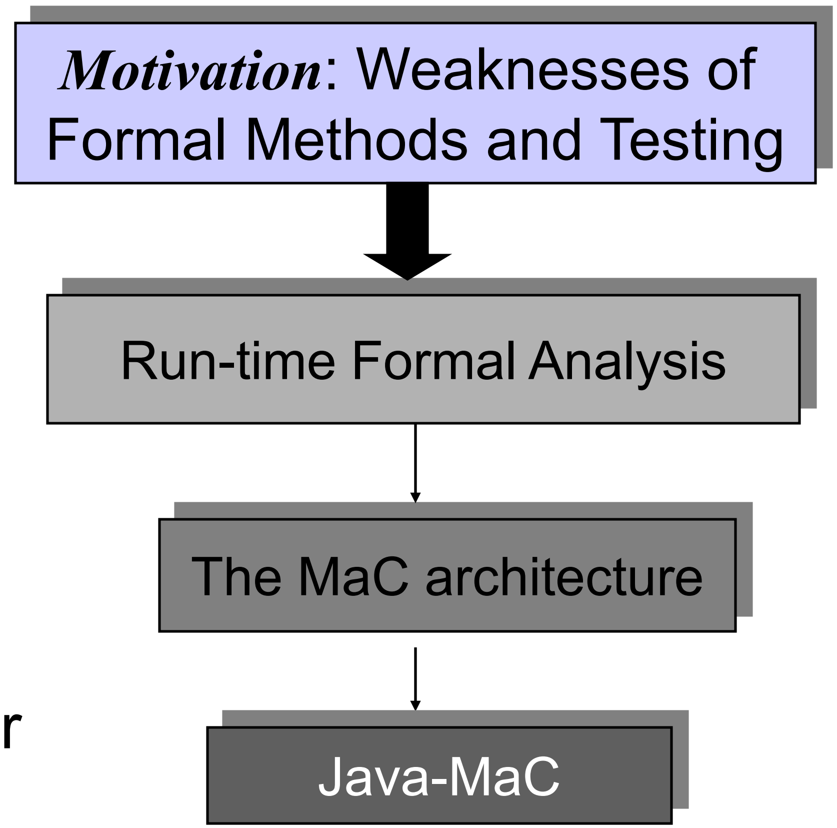
Information Extraction for Run-time Formal Analysis

Moonzoo Kim
KAIST

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)

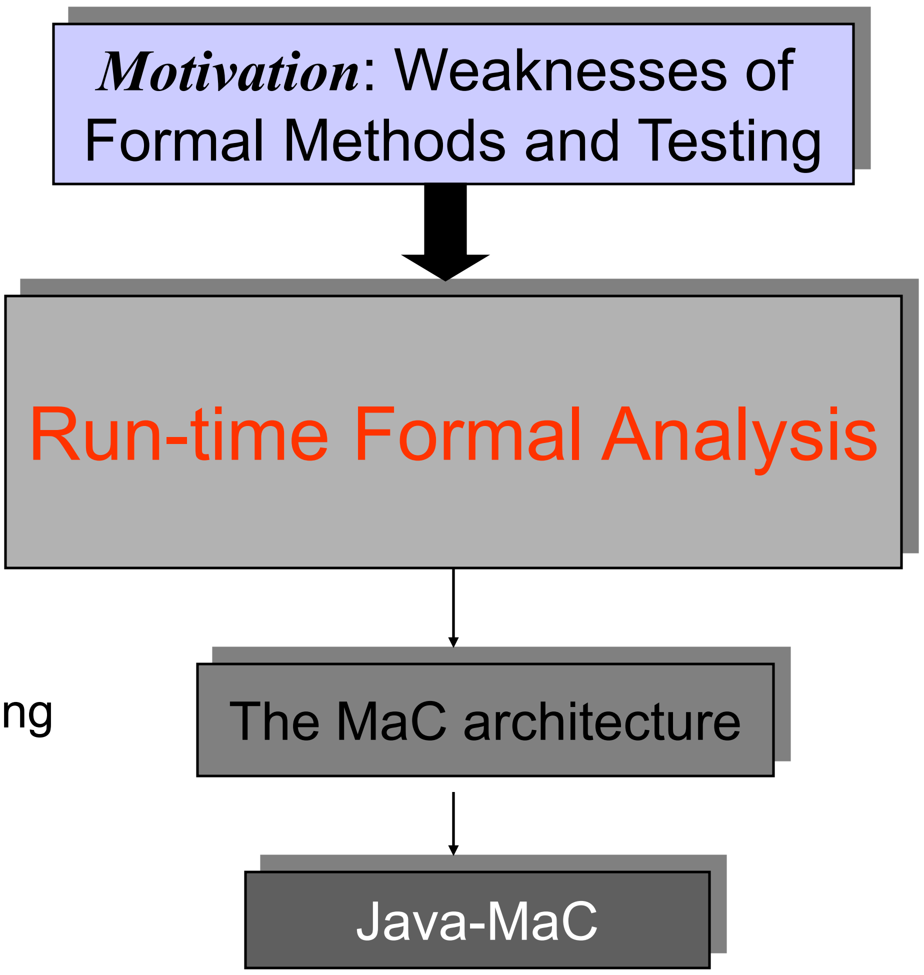


Motivation

- 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

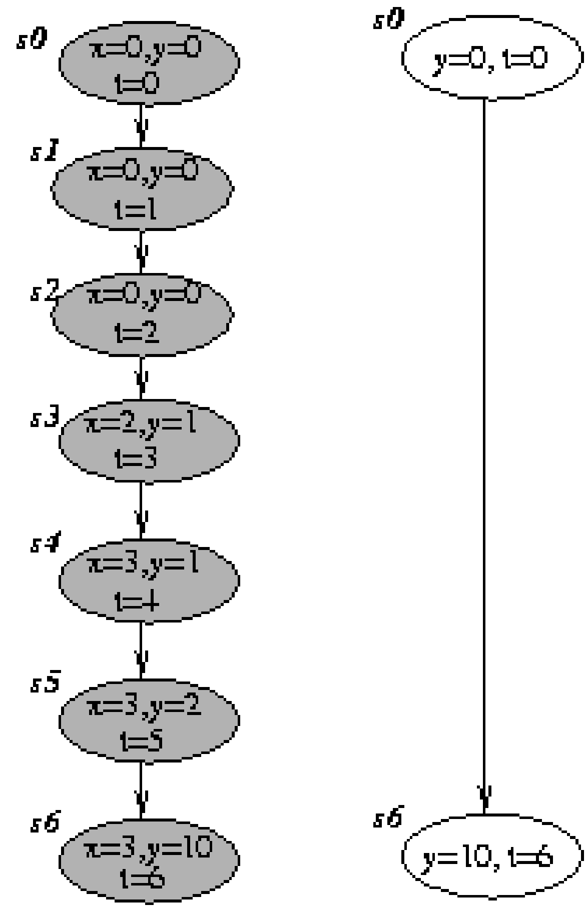


Run-time Formal Analysis

- 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.

Program Execution

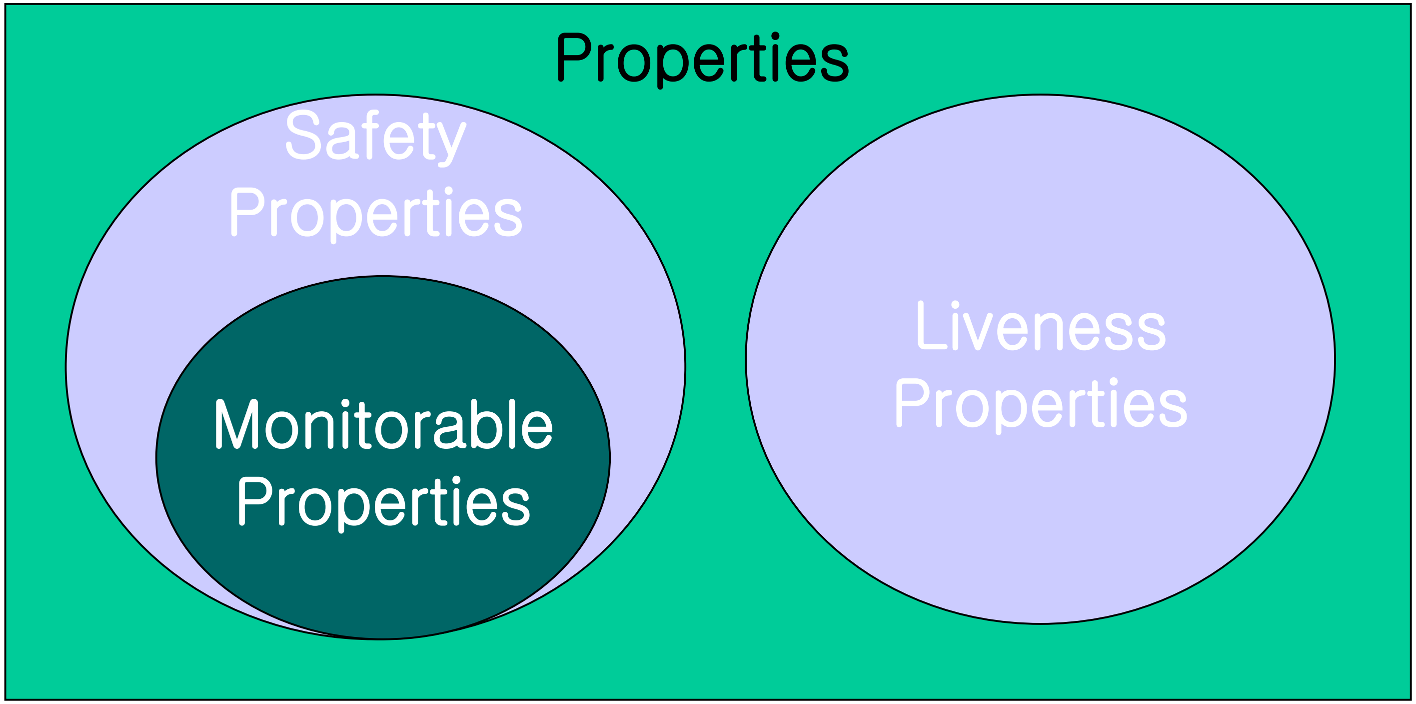
- A program execution σ is a sequence of states $s_0 s_1 \ldots$
  - A state s consists of
    - an environment $\rho_s: V \to R$
    - a timestamp t_s s.t. $t_{s_i} < t_{s_{i+1}}$
- We may abstract out state information unnecessary to detect requirements.




```
property p =

3 < y && y < 11
```

Monitorable Properties

- An execution of a program is an infinite sequence of program states S
- A property is a set of program executions

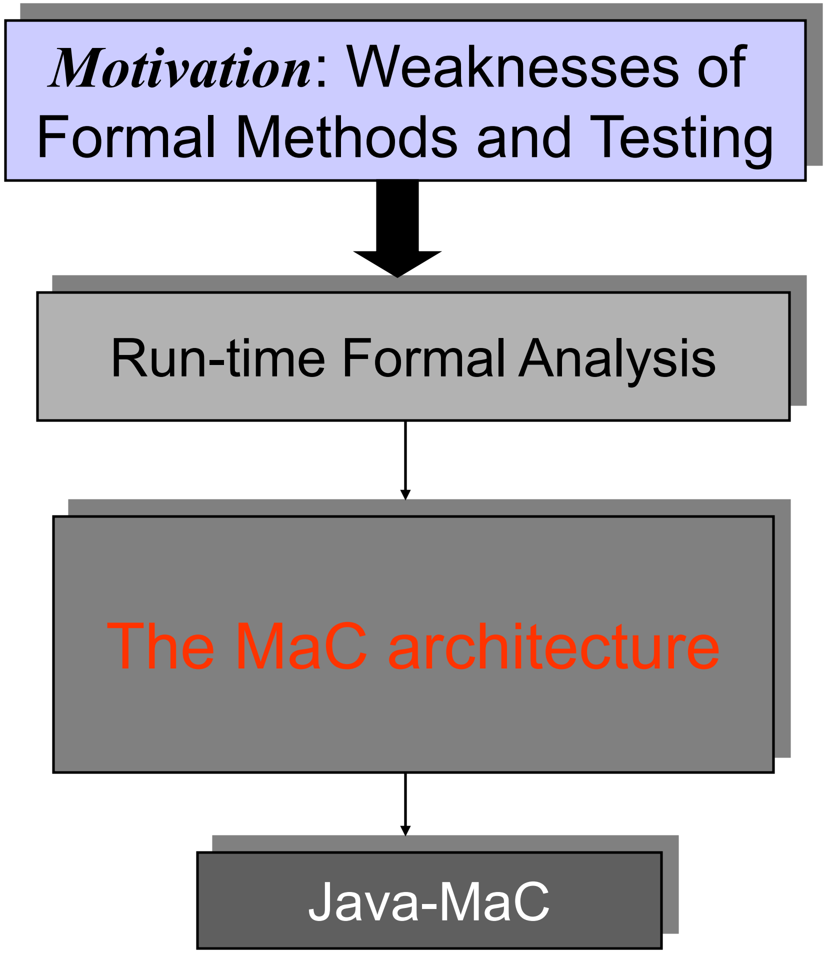


Evaluation of Properties

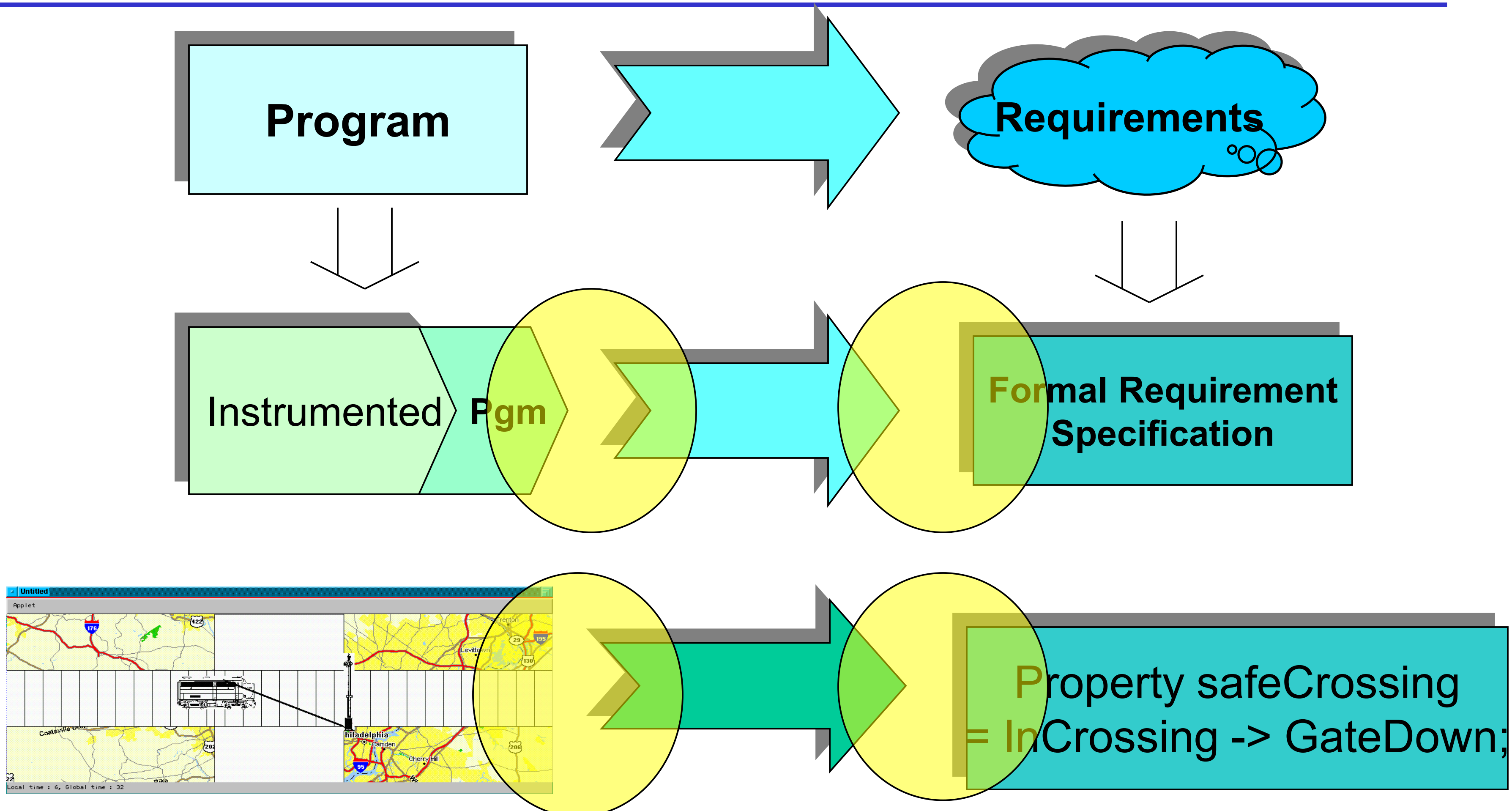
- Complexity of evaluating properties based on a given trace
  - Trace Validity Problem
    - Input: a process P and a string $s \in L^*$
    - Output: is s a valid trace of P
- When properties are given in non-deterministic way, this problem becomes NP-complete
  - reduction of 3SAT to the trace validity problem

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



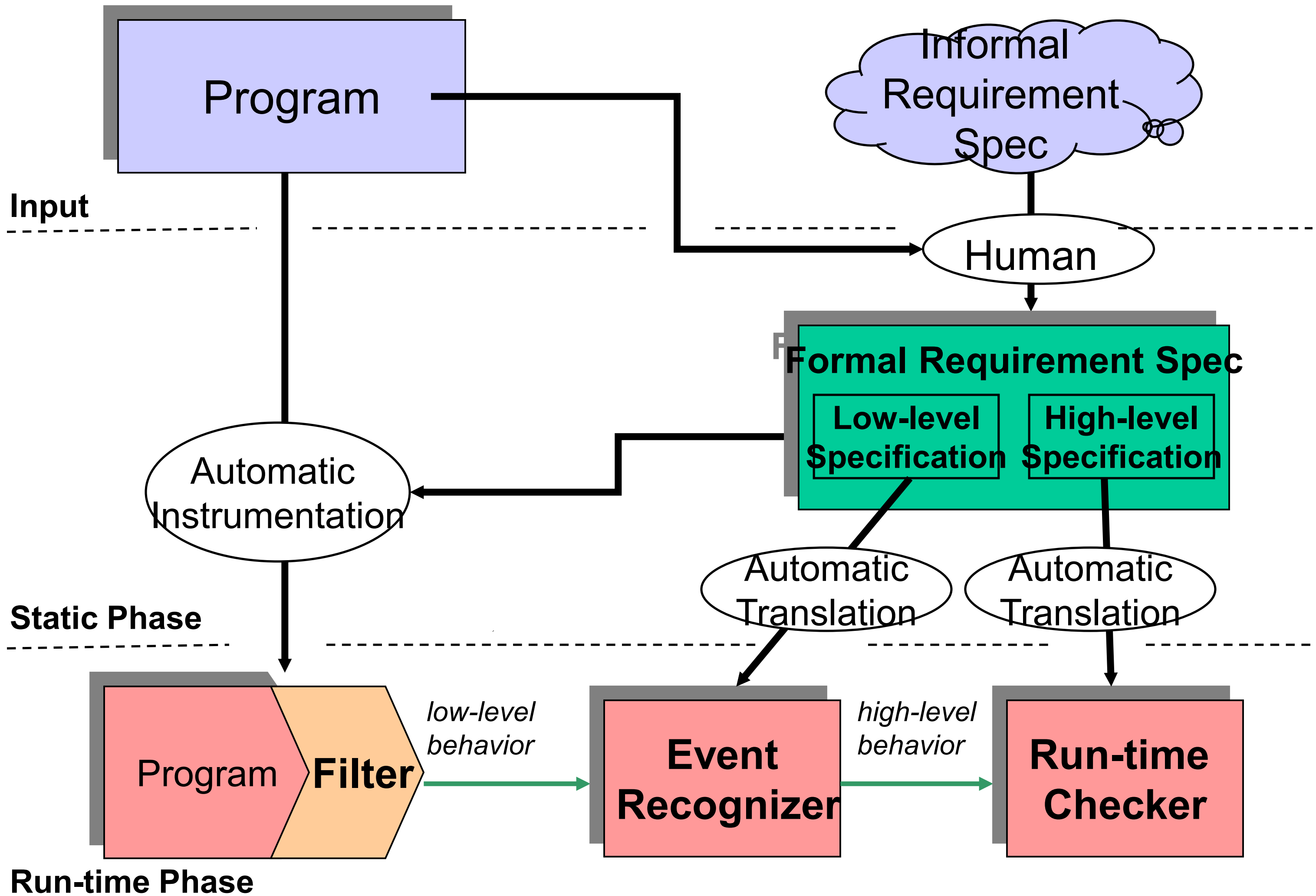
Relation Between Execution and Requirements



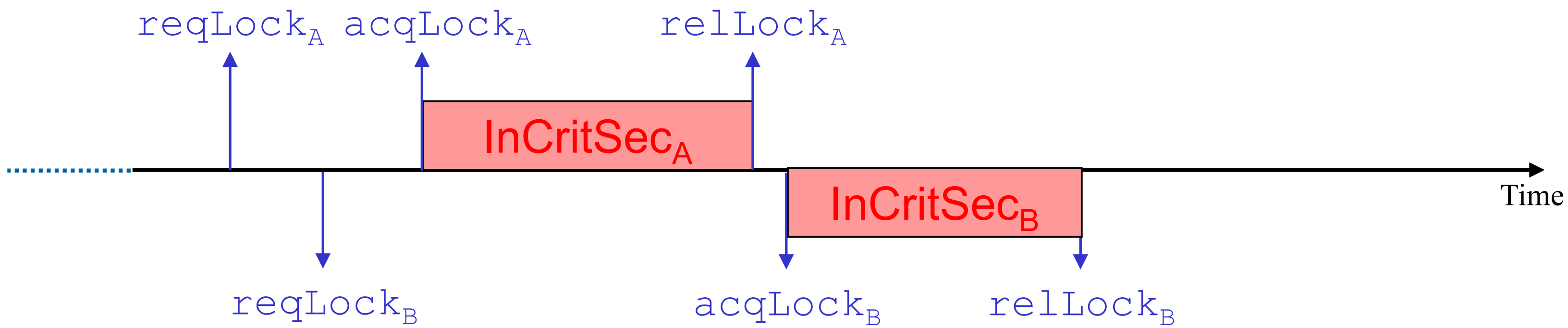
train_pos :20.5
crossing_pos:50
gate_angle:15

InCrossing =
  train_pos > crossing_pos;
GateDown =
  gate_angle == 0;

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

$$C ::= c \mid \text{defined}(C) \mid [E_1, E_2) \mid \neg C \mid C_1 \vee C_2 \mid C_1 \wedge C_2$$

$$E ::= e \mid \text{start}(C) \mid \text{end}(C) \mid E_1 \vee E_2 \mid E_1 \wedge E_2 \mid$$
$$E \text{ when } C$$

- 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)

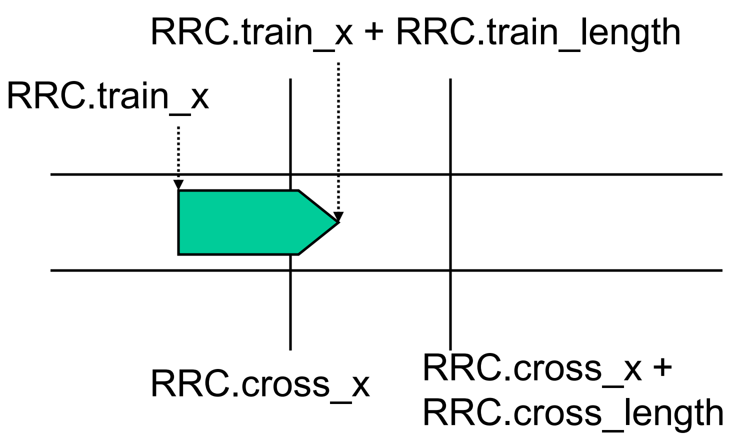
- Expresses requirements using the events and conditions
- Expresses the subset of safety languages.
- Describes the *safety requirements* of the system
  - property safeRRC **= IC -> GD;**
  - alarm violation = start (!safeRRC);
- *Auxiliary variables* may be used to store history.
  - endIC-> { num_train_pass'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

Monitoring Script for Railroad Crossing

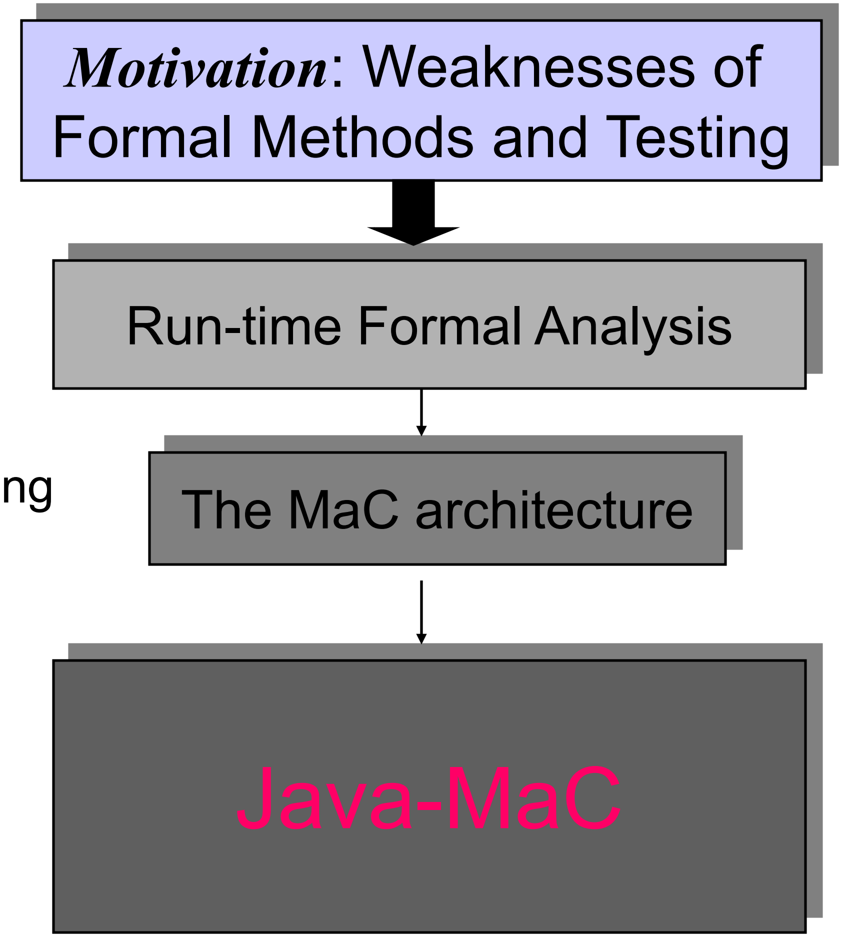
```
MonScr RailRoadCrossing
   export event startIC, endIC, gEndDown,gStartUp;

      monobj float RRC.train_x;
      monobj int    RRC.train_length;
      monobj int    RRC.cross_x;
      monobj int    RRC.cross_length;

      monmeth void Gate.gd(int);
      monmeth int Gate.gu();

      condition IC =
      RRC.train_x + RRC.train_length > RRC.cross_x &&
      RRC.train_x <= RRC.cross_x + RRC.cross_length;

      event startIC   = start(IC);
      event endIC     = end(IC);
      event gEndDown = endM(Gate.gd(int));
      event gStartUp  = startM(Gate.gu());
End
```

```
ReqSpec RailRoadCrossing
   import event startIC, endIC, gEndDown, gStartUp;

      condition IC = [startIC, endIC);
      condition GD = [gEndDown, gStartUp);

      property safeRRC = IC -> GD;

End
```

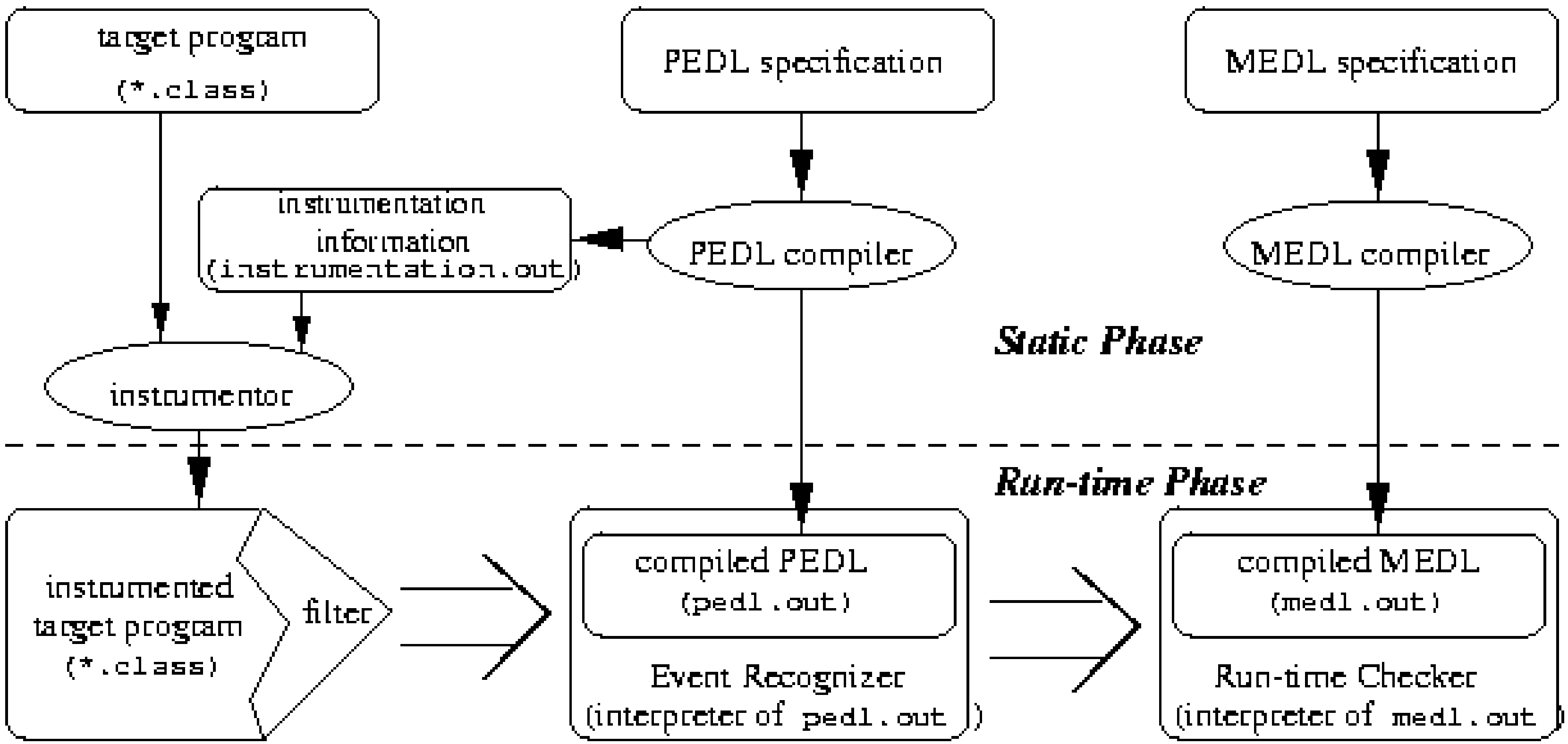


Outline

- 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



The MaC Prototype for Java Programs



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
- time(e) gives the time of the last occurrence of event e
  - used for expressing temporal properties
- value(e,i) gives the i th value in the tuple of values of e
  - value of update(var) : a tuple containing a current value of var
  - value of startM(f) : a tuple containing parameters of the method f
  - value of endM(f) : a tuple containing parameters and a return value of the method f

Sample Probe

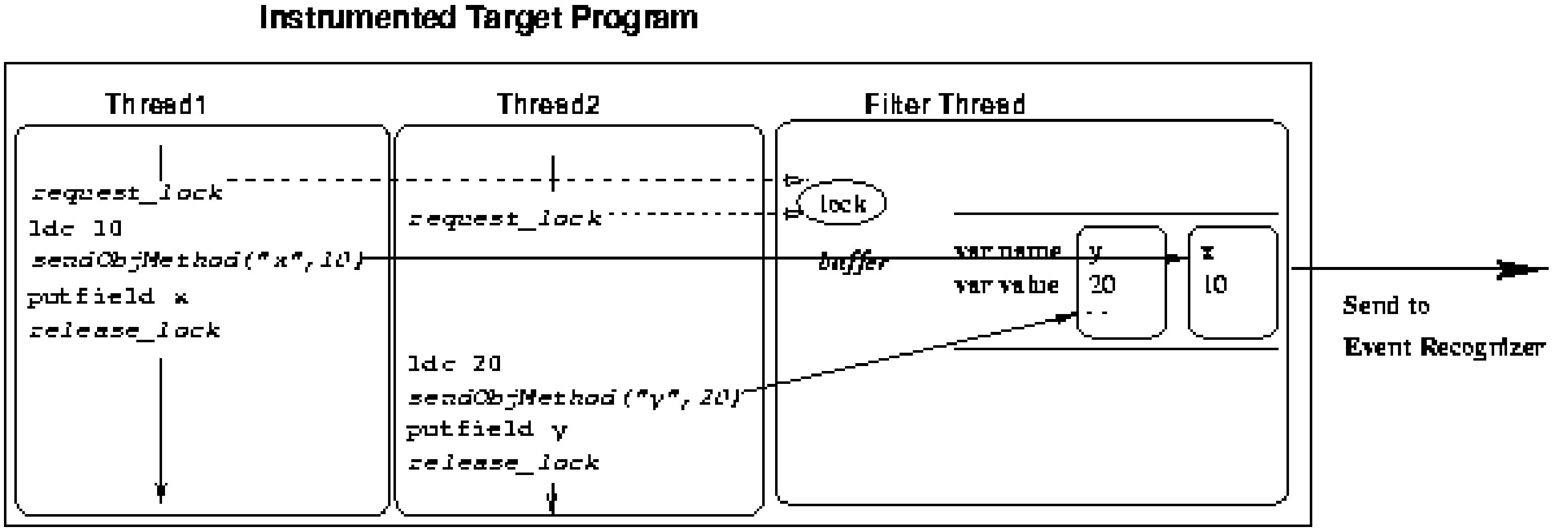
- Monitoring a field variable Var.val

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

```
; >> METHOD 8 <<
.method public run()V
    .limit stack 7
    .limit locals 2
    ...
    getfield DigitalVar.v I
    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
```

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

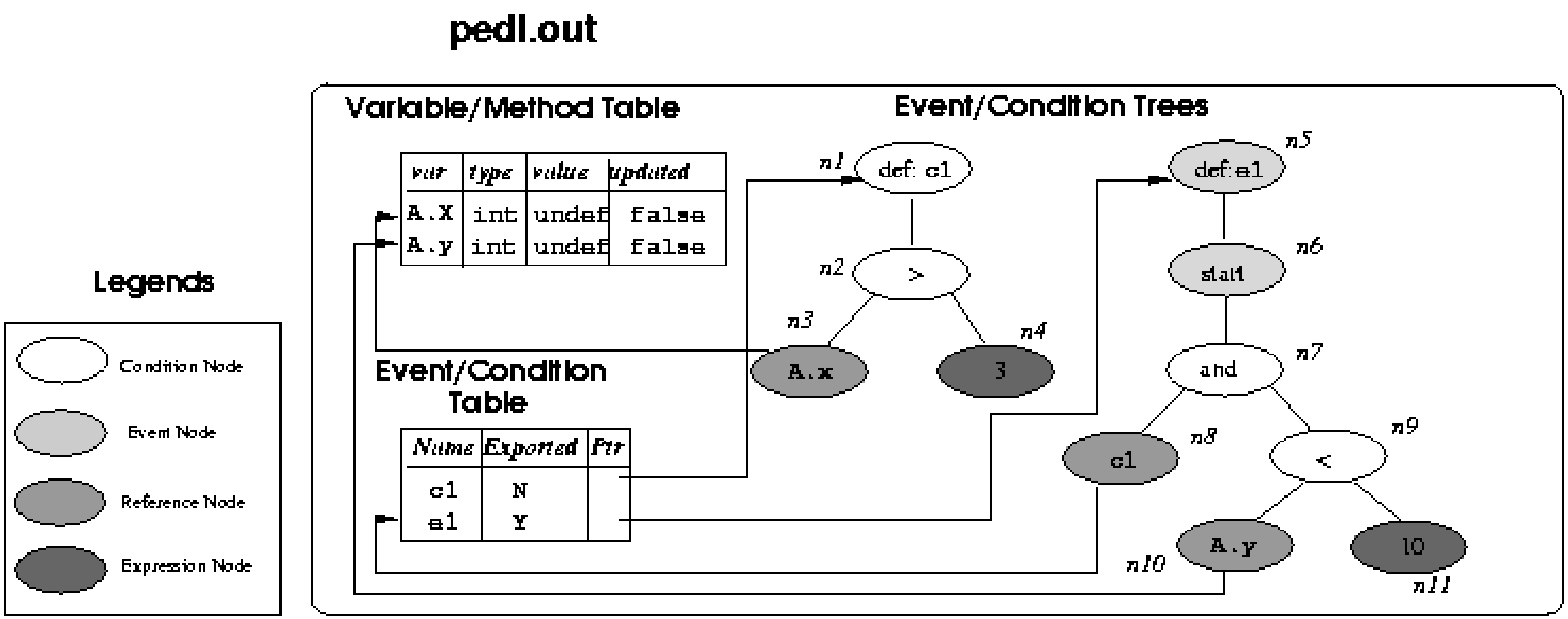


Event Recognizer/Run-time Checker

- Event recognizer
  - evaluates `pedl.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 `medl.out` whenever it receives events and conditions from the event recognizer.
  - detects a violation defined as alarm or property and raises a signal.

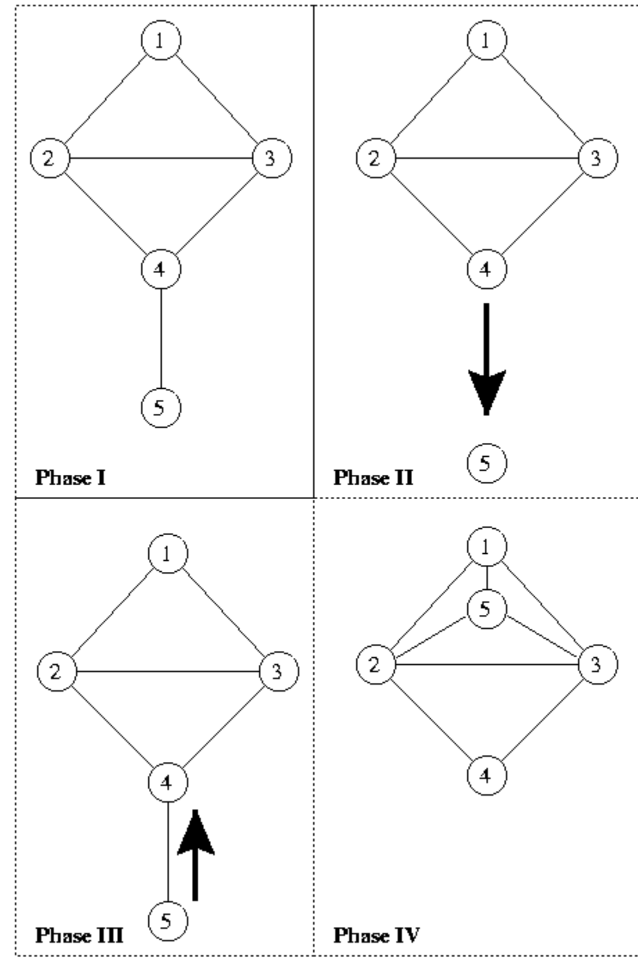
Event Recognizer/Run-time Checker (cont)

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



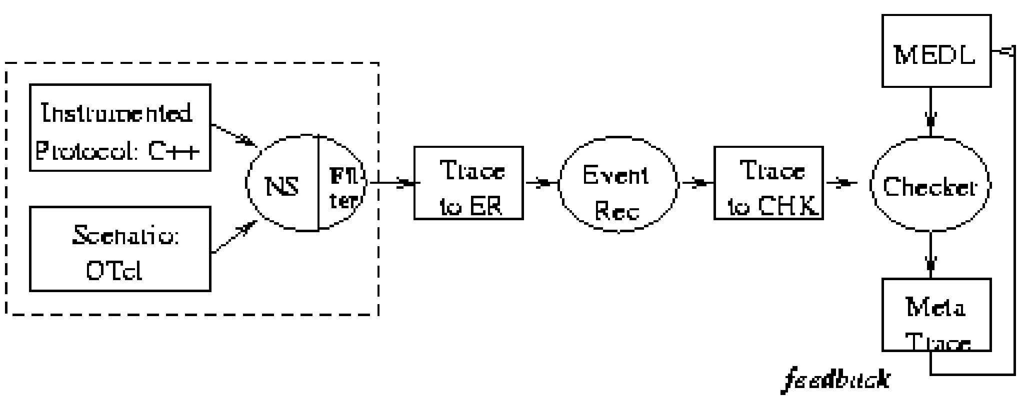
Case Study: Routing Protocol Validation

- 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 [IEEE TSE '02]



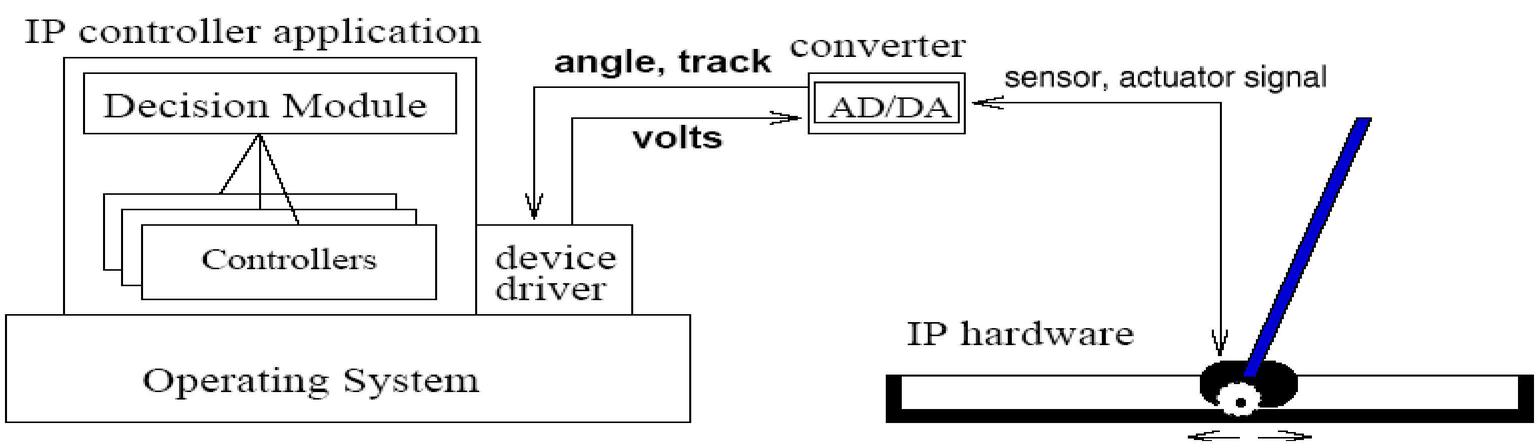
Case Study: Routing Protocol Validation (*cont.*)

- 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



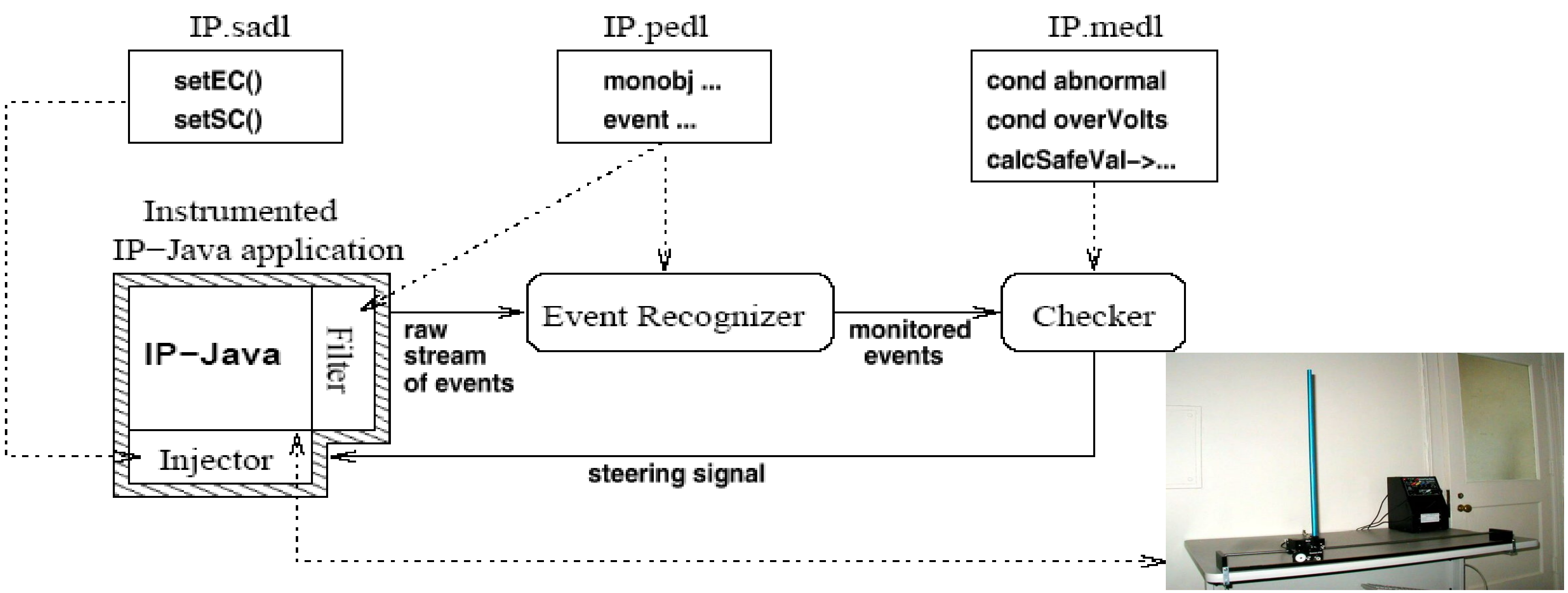
Case Study: Simplex Architecture

- Simplex Architecture [L.Sha '01]
  - dynamic reconfiguration of the system in order to improve performance w/o sacrificing safety
- Implement a fault-tolerance layer for an inverted pendulum (IP) controller using Java-MaC, called IP-Java [Run-time Verification '02]



Case Study: Simplex Architecture (*cont*)

- IP-Java
  - Raw values are extracted from device driver via JNI
  - Experimental controller (EC) registered at run-time
  - Checker computes safety conditions and when the safety conditions are violated, switch to safe controller (SC) through injector



Future Works

- Loosen the restriction on monitoring objects in Java-MaC
  - Combined approach of instrumenting classfiles and modified Java virtual machine
- Apply value abstraction in more general way to gain the benefit of abstraction broadly
- Language extension for easy property description
- Systematic steering activities
- Application areas

References

- ”Information extraction for run-time formal analysis”
  - *Ph.D Thesis M. Kim 2001*
- “Java-MaC: a Rigorous Run-time Assurance Tool for Java Programs”
  - *M. Kim, etc Kluwer Formal Methods in System Design, 2004 (vol 24, no 2)*
- “End-to-end Deployment of Formal Methodology – a case study on multiple reader/writer program”
  - *M. Kim and I. Kang, Letters of Software Engineering, Korea Information Processing Society, 2002 (vol 15, no 2)*
- “Monitoring, checking, and steering of real-time systems”
  - *M. Kim, etc. Runtime Verification, Copenhagen Denmark July 2002*
- “Computational Analysis of Run-time Monitoring – Fundamentals of Java-MaC”
  - *M. Kim, etc. Runtime Verification, Copenhagen Denmark July 2002*
- Verisim: Formal Analysis of Network Simulations”
  - *K. Bhargavan, etc IEEE Transaction on SW Engineering, 2002 (vol. 28, no. 2)*
- “Java-MaC: a Run-time Assurance Tool for Java Programs”
  - *M. Kim, etc, Runtime Verification Paris France July 2001*