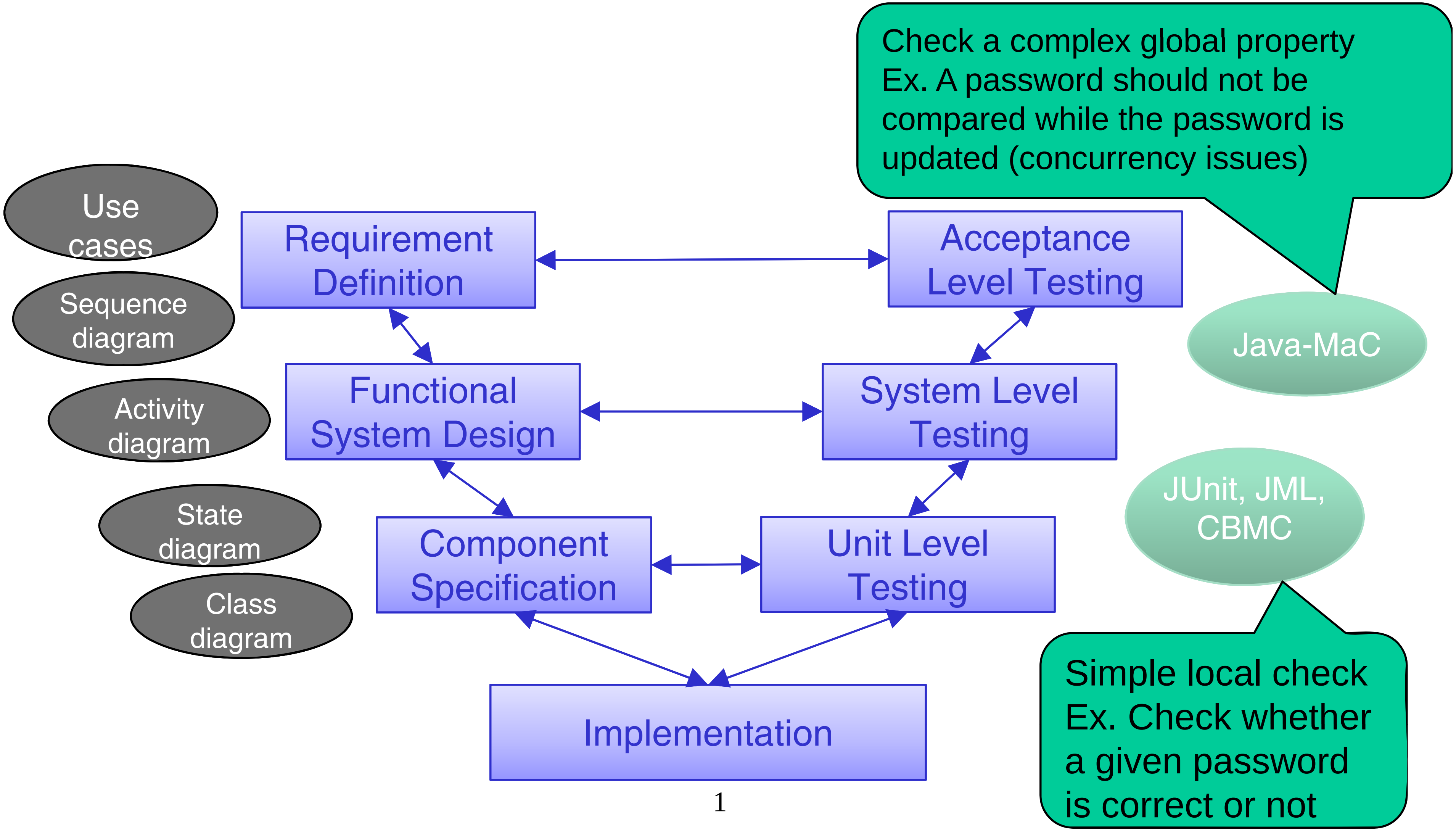
The V-Model of Software Development



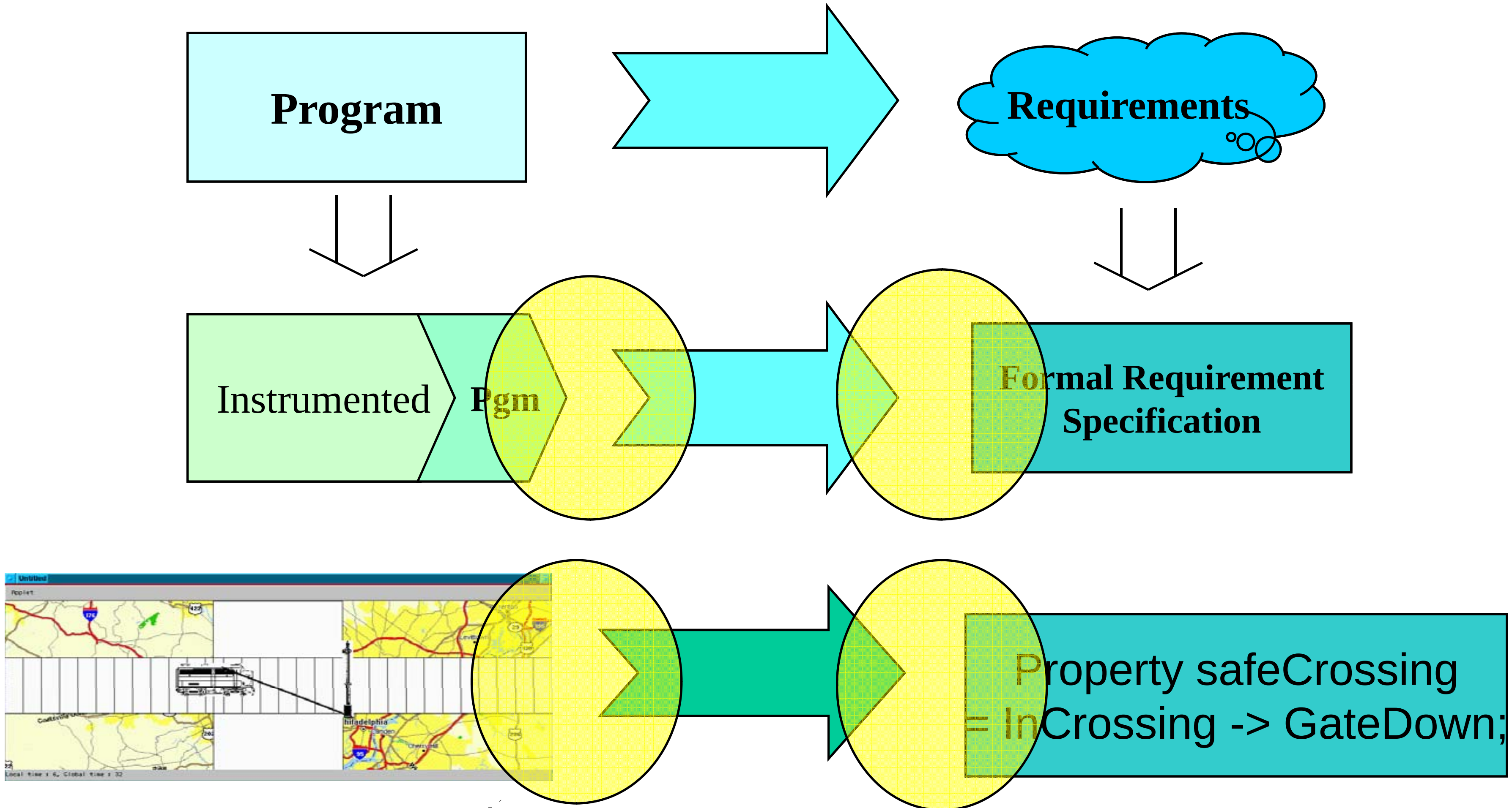
Java-MaC: a Run-time Assurance Tool for Java Programs

Moonzoo Kim
CS Division of EECS Dept.
KAIST
moonzoo@cs.kaist.ac.kr
http://pswlab.kaist.ac.kr/courses/CS350-07

Runtime Verification

- Motivation:
  - Run-time correctness is not guaranteed even after numerous testing
- The goal of run-time verification
  - to give confidence in the run-time compliance of an execution of a system w.r.t formal requirements
  - Monitoring an execution of system constantly with little overhead to detect symptom of (expected) failures
- The analysis validates properties on the *current* execution of application.
  - Similar to testing
- Run-time verification helps user to detect errors and prevent system crash.

Relation Between Execution and Requirements

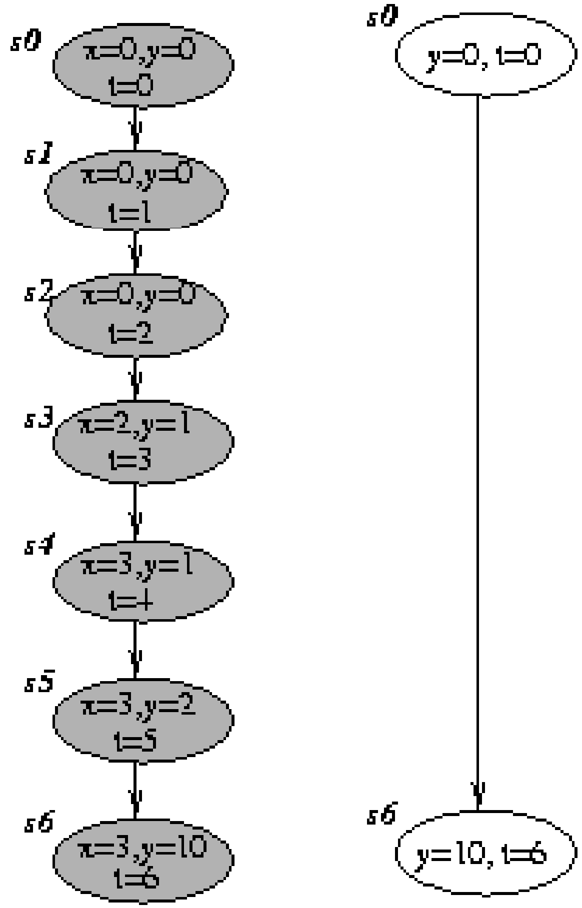



```
train_pos :20.5
crossing_pos:50
gate_angle:15
```

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

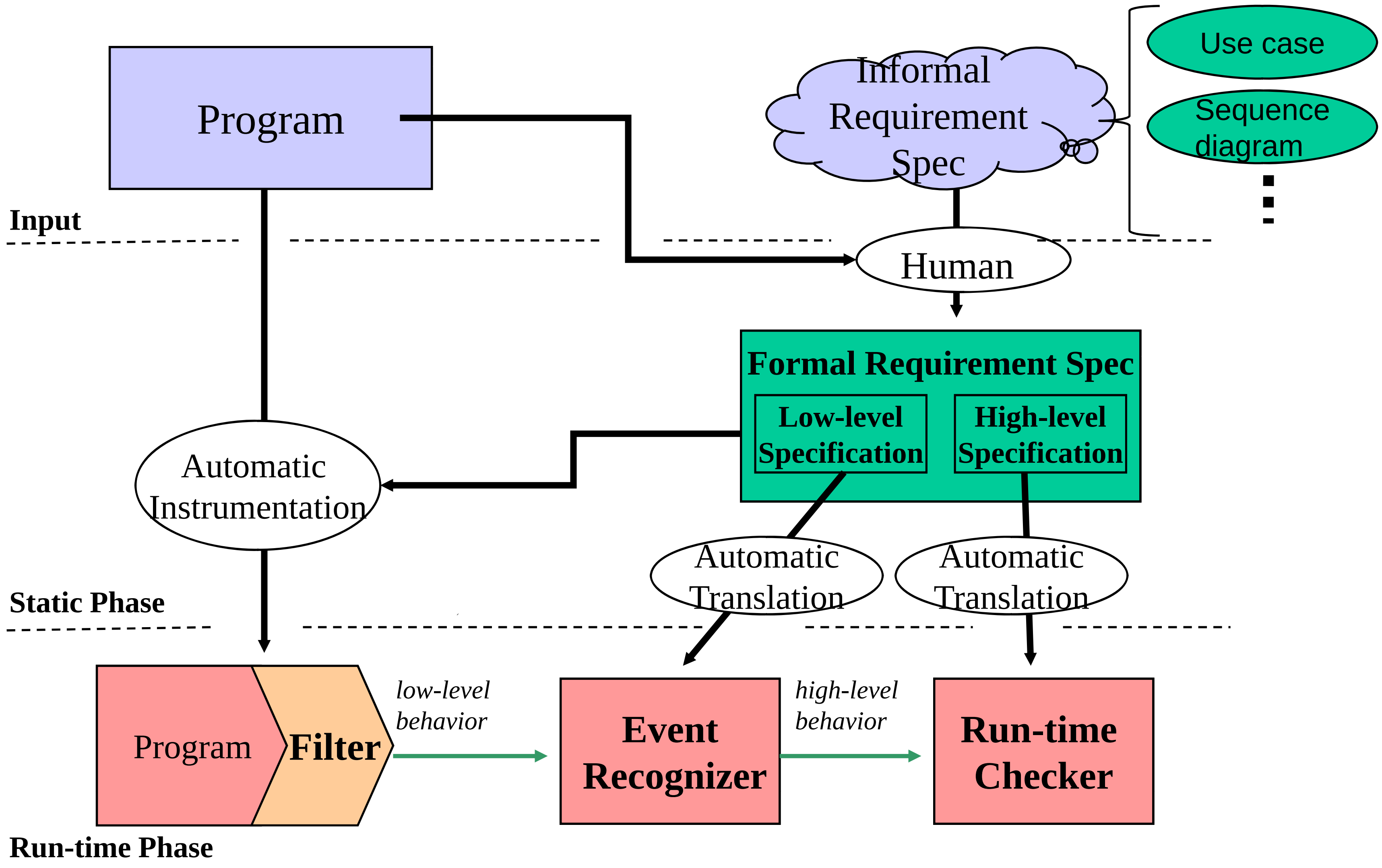
Program Execution

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

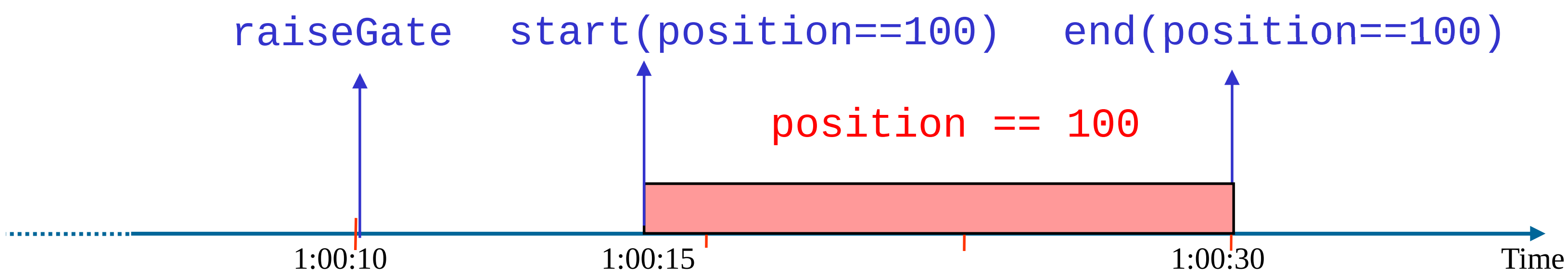



```
property p =
 3 < y && y < 11
```

Overview of the Monitoring and Checking (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.
  - Events and conditions are a natural division, which is also found in other formalisms such as SCR.
- 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.
- [., .) 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)
  - Specify *what to monitor* in the target program
    - Provides primitives to refer to values of variables and to certain points in the execution of the program.
  - Maps the *low-level state information* of the system to *high-level events*.
  - PEDL is designed so that events can be recognized in time linear to the size of the PEDL specification
  - 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, in terms of conditions that must always be true, and alarms (events) that must never be raised.
  - property safeRRC **= IC -> GD;**
  - alarm violation = start (!safeRRC);
- *Auxilliary 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
```

The MaC prototype for Java programs: Java-MaC

- PEDL for Java
- Monitoring objects
- Instrumentation process
- Structure of Java-MaC
- Run-time components

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

Instrumentation

- Java-MaC instruments Java executable code
- Java-MaC instrumentor detects instructions
  - variable updates
    - putstatic/putfield for global 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 invoking
  - sendObjMethod(Object parentAddress, <T> value, String varName)

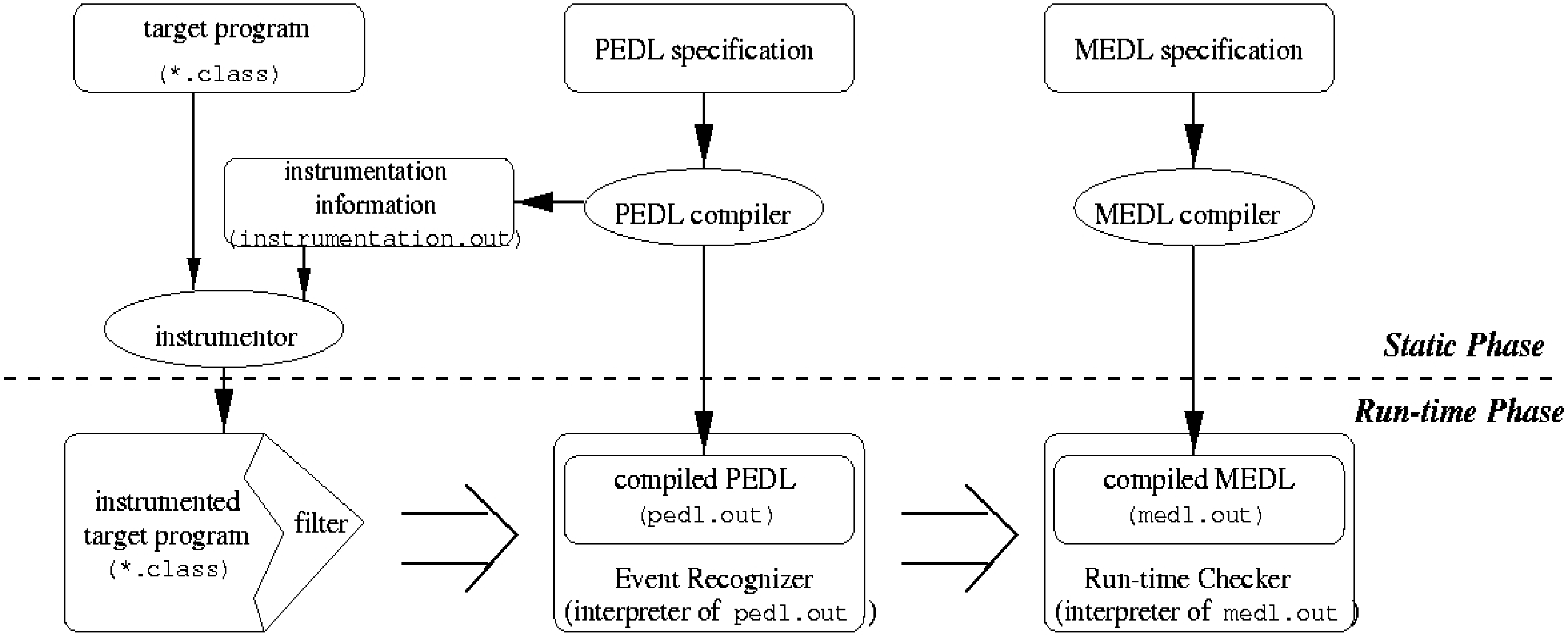
Sample Probe

- 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;Ijava/lang/String;)V
    putfield Var.val I
    getstatic mac.filter.Filter.lock Ljava.lang.Object;
    monitorexit
     ...
.end Method
```

Overview of Java-MaC



Run-time Components of Java-MaC

- 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
- Event recognizer
  - evaluates the abstract syntax tree generated from a PEDL specification 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

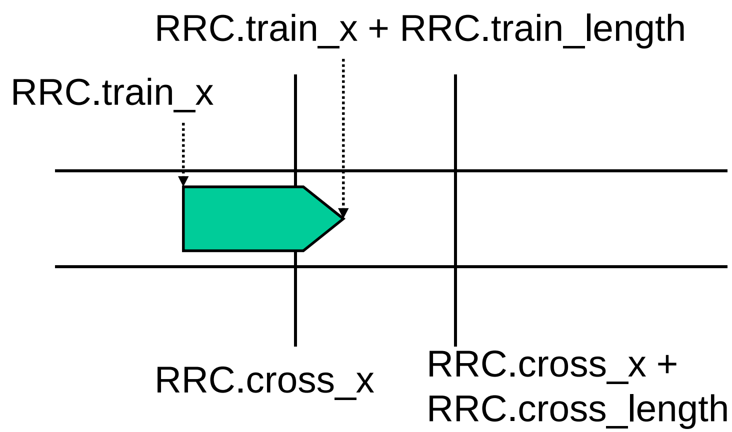
Run-time Components of Java-MaC (cont.)

- Run-time checker
  - evaluates the abstract syntax tree generated from a MEDL specification whenever it receives events and conditions from the event recognizer.
  - Detects a violation defined as alarm or property and raises a signal.
- Connection among run-time components
  - TCP socket connection
  - FIFO file connection
  - User implemented connection using InputStream and OutputStream obtained by Java-MaC API

Monitoring Script for Railroad Crossing

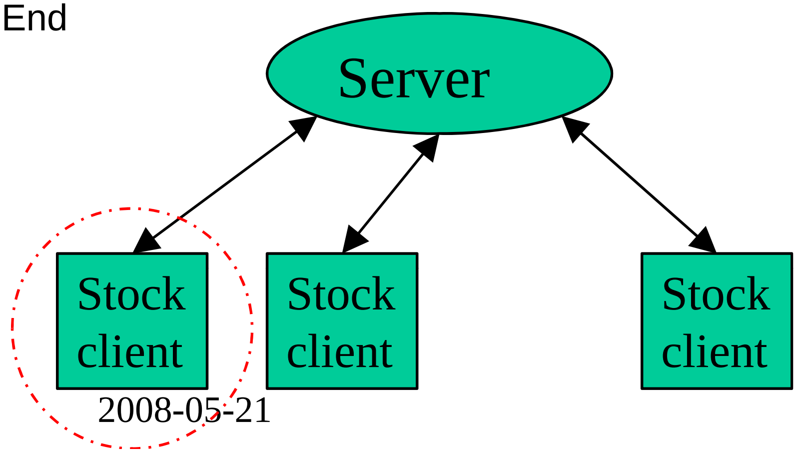
```
MonScr RailRoadCrossing
   export event startIC, endIC, gEndDown,gStartUp;

      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
```



Specifications for Stock Clients

```
MonScr StockClient
   export event startPgm, periodStart, conFail,
                queryResend, oldDataUsed;

   monmeth void Client.main(String[]);
   monmeth void Client.run();
   monmeth void Client.failConnection(ConnectTry);
   monmeth Object Client.retryGetData(int);
   monmeth Object Client.processOldData();

   event startPgm = startM(Client.main(String[]));
   event periodStart = startM(Client.run());
   event conFail = startM(Client.failConnection(ConnectTry));
   event queryResend = startM(Client.retryGetData(int));
   event oldDataUsed = startM(Client.processOldData());
End
```




```
ReqSpec StockClient
   import event startPgm, periodStart, conFail,
                queryResend, oldDataUsed;

   var long periodTime;
   var long lastPeriodStart;
   var int numRetried;
   var int numConFail;

   alarm violatedPeriod = end((perioidTime' >= 900)
           && (periodTime' <= 1100));
   alarm wrongFT = oldDataUsed when (
           (numRetries' < 4)|| (numConFail' < 3));

   startPgm -> {periodTime' = 1000;
         lastPeriodStart' = time(startPgm) -1000;
         numRetries' = 0;
         numConFail' = 0;}
   periodStart ->{ numREtries' = 0;
         numConFail' = 0;
         periodTime' =time(periodStart)-lastPeriodStart;
         lastPeriodStart' = time(periodStart);}
...
End
```

Conclusion and Future Work

- The MaC architecture provides a lightweight formal methodology for assuring of the correct execution of a target program at run-time
  - Rigorous analysis
  - Flexibility
  - Automation
  - Easy of use
- Systematic extension of the MaC architecture to platforms other than Java
- http://www.cis.upenn.edu/~rtg/mac