



# Run-time verification: a MaC approach

Insup Lee    Usa Sammapun    Oleg Sokolsky

University of Pennsylvania

QEST Tutorial, Riverside, CA, September 11, 2006

Modified for CIS 480, Spring 2009

## Outline

- ▶ **Motivation and overview**
  - Why run-time verification
  - Formal methods and run-time verification
  - Property specification
  - Incremental property checking
- ▶ **MaC framework**
  - » Break
- ▶ **More on MaC framework**
- ▶ **Applications**



QEST 2006



2

## Motivation

- ▶ Run of a large system – real or simulated – produces lots of observations
- ▶ How do we make sense of a simulation run?
- ▶ Different aspects may be interesting:
  - Is it correct?
  - Does it have the necessary performance, reliability, etc.?
  - Are simulation parameters and input data suitable?
- ▶ Each of these questions is a property that needs to be checked



QEST 2006



3

## Properties of runs

- ▶ Behavioral
  - Sequencing of events
  - Correlation between values
    - Boolean
- ▶ Timing
  - Duration of interactions and computations
  - Timeliness
    - Boolean or quantitative
- ▶ Quality of service
  - Collection of statistics, aggregation of data
    - Mostly quantitative



QEST 2006



4

## Checking properties of runs

- ▶ By direct observation
  - No tools needed
  - Possible for simple and short traces
- ▶ By a custom checker
  - Checkers can be simple (e.g. PERL scripts)
  - Works fine if there are few fixed properties to check
- ▶ By a checker for a suitable property specification language
  - Flexible
  - Can be formal



QEST 2006



5

## Formal methods

- ▶ Specification
  - Precisely state what the system should be doing
    - Based on a language with mathematical semantics
- ▶ Verification
  - Prove that the system does the right thing
    - Use formal semantics to develop checking algorithms
- ▶ Satisfaction relation
 
$$M \models P$$
- ▶ Model checking
  - Algorithms for automatic checking of satisfaction



QEST 2006



6

## Temporal logic properties

- ▶ Describe evolving systems, that go through sequences of “worlds”
- ▶ Behavioral properties
  - Worlds are characterized by atomic propositions
  - Operators
    - Future: “eventually”, “globally”, “until”
    - Past: “previously”, “since”
- ▶ Quantitative properties
  - Worlds contain quantitative information
  - Operators
    - “eventually within interval”, “at least that much throughput”

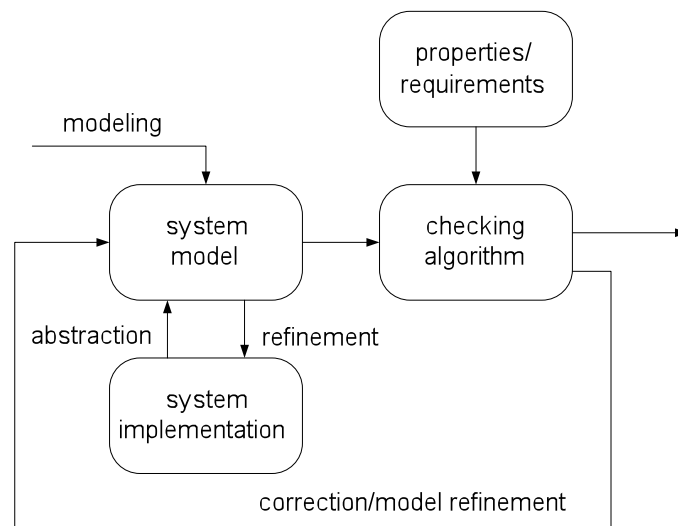


QEST 2006



7

## Model checking



QEST 2006

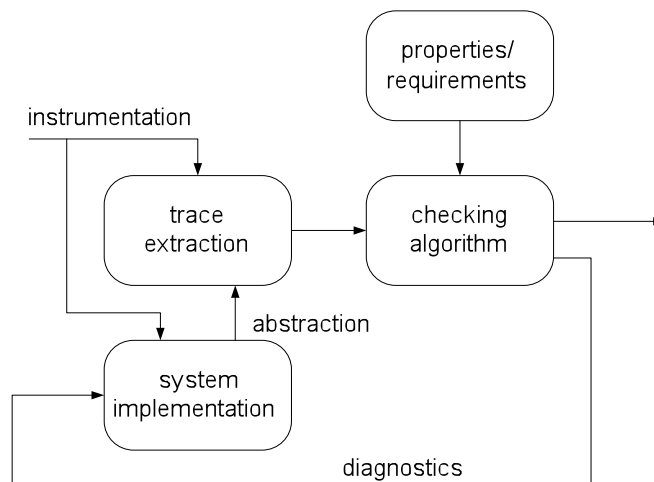


8

## Formal methods at run time

- ▶ Compared to model checking, there is no model
  - Execution trace is used as the model
- ▶ Trace extraction is easier than model extraction
  - No overapproximation involved
- ▶ Property checking on a trace is easier than over an arbitrary model
- ▶ Obviously, a weaker result is proved
  - Applies to current execution and not all executions
    - Can be generalized in some restricted cases

## Verification vs. runtime verification



## Monitoring behavioral properties

- ▶ Formulas in a temporal logic
- ▶ Always evaluated over a finite execution trace
- ▶ Safety properties
  - “something bad does not happen”
    - Raise alarm when the bad happens
- ▶ Liveness properties
  - Requires non-traditional interpretation
    - Check satisfaction at trace end, or
    - Check if finite trace can be extended to a compliant infinite trace
- ▶ We will consider safety properties only



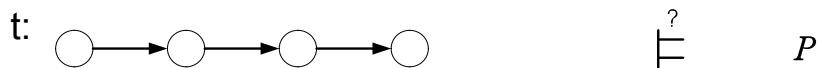
QEST 2006



11

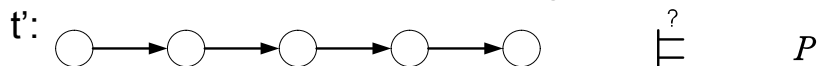
## Checking a property of a trace

- ▶ Satisfaction relation



- ▶ Simple algorithm, linear in the trace length

- ▶ At each step, trace becomes longer



- ▶ Furthermore, traces are too big to store

- ▶ **Need a different approach**



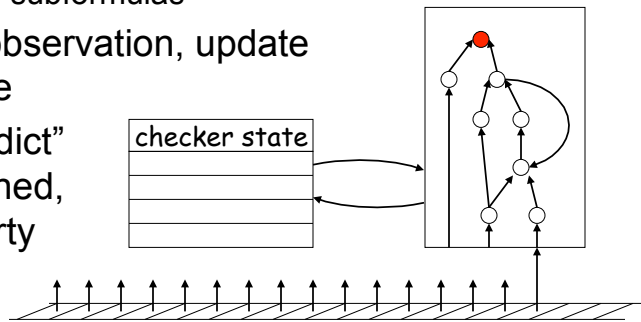
QEST 2006



12

## Incremental checking of a trace

- ▶ In fact, we do not need to check the whole trace over and over again
- ▶ Keep a checker state
  - values of all subformulas
- ▶ Upon each observation, update checker state
- ▶ When a “verdict” state is reached, report property value



## What about quantitative properties?

- ▶ Checker state need not be all boolean
- ▶ Auxiliary variables can store
  - Time instances and intervals
  - Event counts
  - Aggregate values
  - ...
- ▶ Predicates over auxiliary variables can be used as new atomic formulas
- ▶ “Verdict” states can also report values stored in auxiliary variables

## Requirements vs. observations

- ▶ Ultimately, properties determine what observations are relevant
  - Each atomic statement has to be matched to an observation
- ▶ System requirements are high-level and independent of an implementation
- ▶ Run-time observations are low-level and implementation-specific
  - Software: variable assignments, function calls, exceptions, etc.
  - Network: send, receive, route packets, update routing tables, etc.
- ▶ **Need an abstraction layer to match the two**



QEST 2006



15

## Trace extraction

- ▶ Too much information is just too much!
  - Trace is a sequence of observations
    - A temporal projection of execution
  - Observation is a projection of system state
    - Keep only relevant state components
- ▶ Too little information is a problem, too
  - Did you miss anything important?
  - Can you observe everything you need?
    - Not an issue with simulations, unless the model is a black box
  - Can you observe well enough?



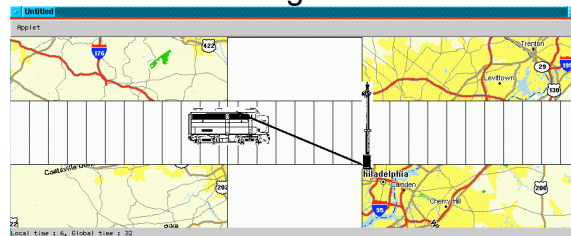
QEST 2006



16

## Running example

- ▶ Simulation of a railroad crossing
- ▶ Requirement: train in crossing => gate is down
- ▶ Observations:
  - gateUp, gateDown – changes in gate status
  - raiseGate, lowerGate – commands to move gate
  - position – coordinate of the train along the track



## Outline

- ▶ Motivation and overview
- ▶ **MaC framework**
  - Architecture
  - Specification languages
  - Implementation
    - » Break ...
  - Extensions
- ▶ Applications

## MaC: Monitoring and Checking

- ▶ Designed at U. Penn since 1998
- ▶ Components:
  - Architecture for run-time verification
  - Languages for monitoring properties and trace abstraction
  - Steering in response to alarms
- ▶ Prototype implementation
  - Implementation of checking algorithms
  - Recognition of high-level events
  - For Java programs: automatic instrumentation

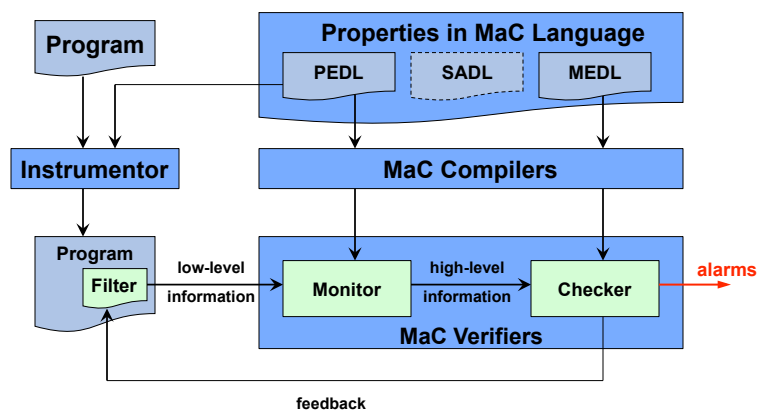


QEST 2006



19

## MaC architecture

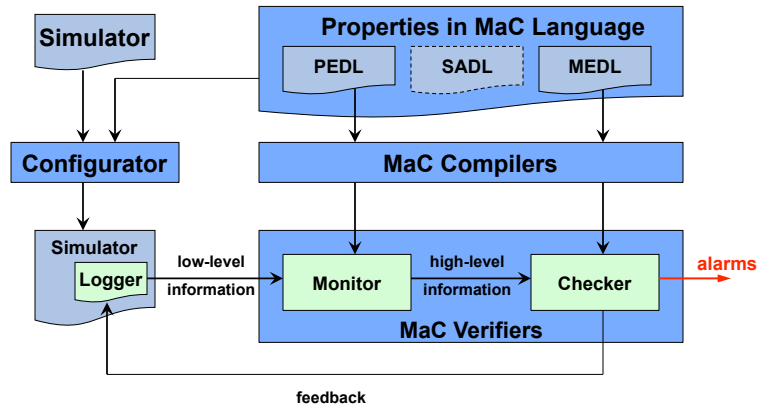


QEST 2006

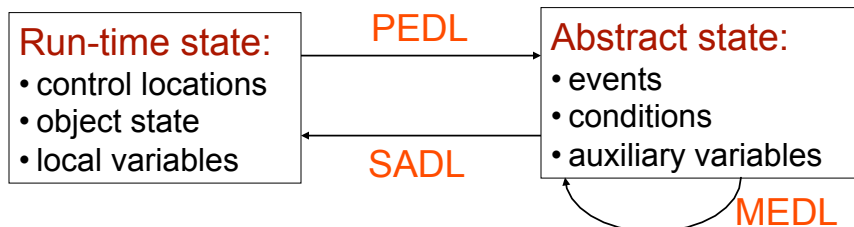


20

## MaC architecture - simulation



## MaC languages



- ▶ PEDL: Primitive Event Definition Language
  - abstraction
- ▶ MEDL: Meta Event Definition Language
  - abstract transformation
- ▶ SADL: Steering Action Definition Language
  - feedback

## Properties: **events** and **conditions**

- ▶ Natural distinction for monitoring properties:
  - instantaneous** vs. **durational**
    - Instantaneity depends on time granularity
- ▶ Motivations for the distinction:
  - Specification styles – state vs. event-based
  - Cannot monitor every time instance
- ▶ What is the value between trace states?
  - If you saw something in an observation, is it still there while you are not looking?
    - Yes – it is a **condition**
    - No – it is an **event**



QEST 2006



23

## Example: hundred years' war

- ▶ The war is a condition
- ▶ Battles are events
  - Battle durations notwithstanding
- ▶ Events change the state of conditions
  - $\text{end}(\text{War}) = \text{FallOfBordeaux}$
  - $\text{FinalDefeat} = [\text{FallOfParis}, \text{FallOfBordeaux}]$



QEST 2006

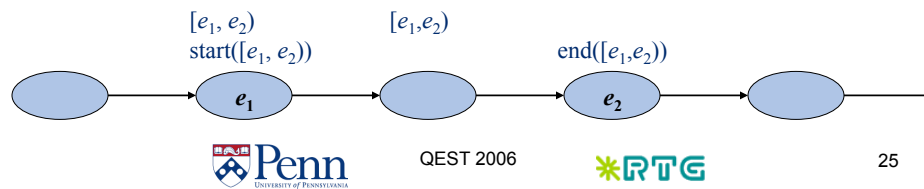


24

## Logical foundation

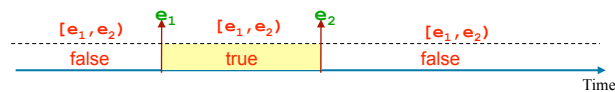
- ▶ *LEC*: 2-sorted logic: events and conditions
- ▶ Syntax:
 
$$E ::= e \mid \text{start}(C) \mid \text{end}(C) \mid E_1 \vee E_2 \mid E_1 \wedge E_2 \mid E \text{ when } C$$

$$C ::= c \mid [E_1, E_2) \mid \neg C \mid C_1 \vee C_2 \mid C_1 \wedge C_2$$
- ▶ Operator  $[., .)$  pairs events to define an interval
- ▶ Operators *start* and *end* define the events at the instant when conditions change their value

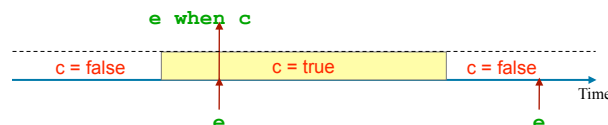


## Operators

- ▶ Interval operator:  $[e_1, e_2)$ 
  - Becomes true when  $e_1$  occurs
  - Becomes false when  $e_2$  occurs



- ▶ When operator  $e \text{ when } c$



## Semantic function

- ▶ Formally, a model  $M = (S, \tau, L_C, L_E)$ 
  - $S = \{s_0, s_1, \dots\}$  is a sequence of states
  - $\tau$  is a mapping from  $S$  to a time domain
  - $L_C(s, c)$  is a function that assigns to each state  $s$  the truth value of primitive condition  $c$
  - $L_E(s, e)$  is a partial function defined for each event  $e$  that occurs at  $s$
- ▶  $M, t \models c$  means a condition  $c$  being true in a model  $M$  at time  $t$
- ▶  $M, t \models e$  means an event  $e$  occurring in a model  $M$  at time  $t$



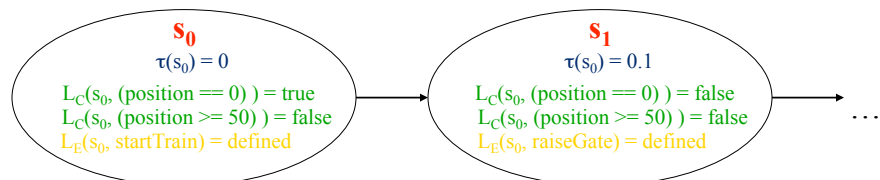
QEST 2006



27

## Traces as models

- ▶ An execution trace  $M = (S, \tau, L_C, L_E)$  is viewed as a sequence of worlds
- ▶ Each world has descriptions of:
  - Truth values of primitive conditions
  - Occurrences of primitive events



QEST 2006



28

## Semantic function

$M, t \models c$	iff	$\mathcal{D}_M^t(c) = true$ → next slide
$M, t \models e_k$ ( $e_k$ primitive)	iff	there exists state $s_i$ such that $\tau(s_i) = t$ and $L_E(s_i, e_k)$ is defined.
$M, t \models start(c)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models c$ and if $i > 0$ then $M, \tau(s_{i-1}) \not\models c$ ; i.e., $start(c)$ occurs when condition $c$ changes from false or undefined, to true; before state $s_0$ conditions are assumed to be undefined.
$M, t \models end(c)$	iff	$\exists s_i$ such that $\tau(s_i) = t$ and $M, \tau(s_i) \models !c$ and if $i > 0$ then $M, \tau(s_{i-1}) \not\models !c$ ; i.e., $end(c)$ occurs when condition $c$ changes from true or undefined, to false; before state $s_0$ conditions are assumed to be undefined.
$M, t \models e_1    e_2$	iff	$M, t \models e_1$ or $M, t \models e_2$ .
$M, t \models e_1 \&\& e_2$	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.



QUEST 2006



29

## Denotation for Conditions

[ $c_k$ 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]	$\mathcal{D}_M^t(c_1 \Rightarrow c_2) = \mathcal{D}_M^t(!c_1    c_2)$



QUEST 2006



30

## PEDL

- ▶ Primitive Event Definition Language
- ▶ Low-level specification
- ▶ Dependent on underlying applications
- ▶ Principles
  - Encapsulate all implementation-specific details of the monitoring process
  - Process of event recognition to be as simple as possible
- ▶ Reason only about the current state in the execution trace



QEST 2006



31

## PEDL constructs

- ▶ Declaration of monitored variables
- ▶ Definitions of primitive conditions
  - Predicates over monitored variables
- ▶ Definitions of primitive events
  - Update to a monitored variable  $x$ :  $\text{update}(x)$ 
    - New value is an attribute of the event
  - Other primitive events depend on the target system
    - For software: function/method calls and returns
    - For network models: send/receive
    - For automata models: transitions/mode switches

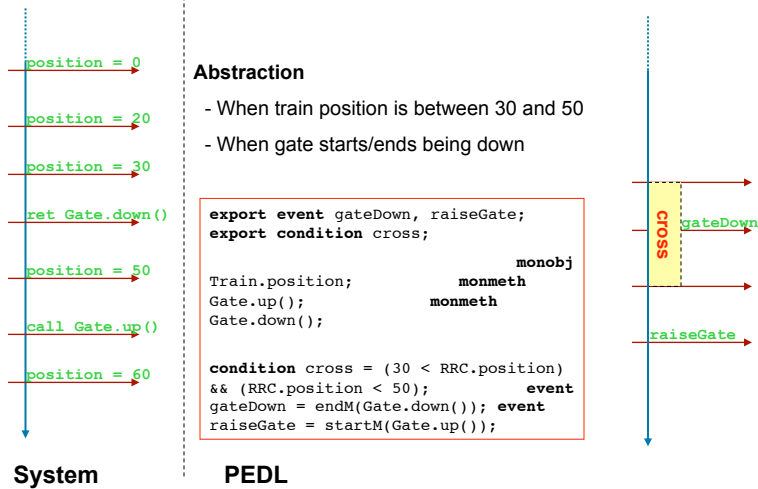


QEST 2006



32

## PEDL by example

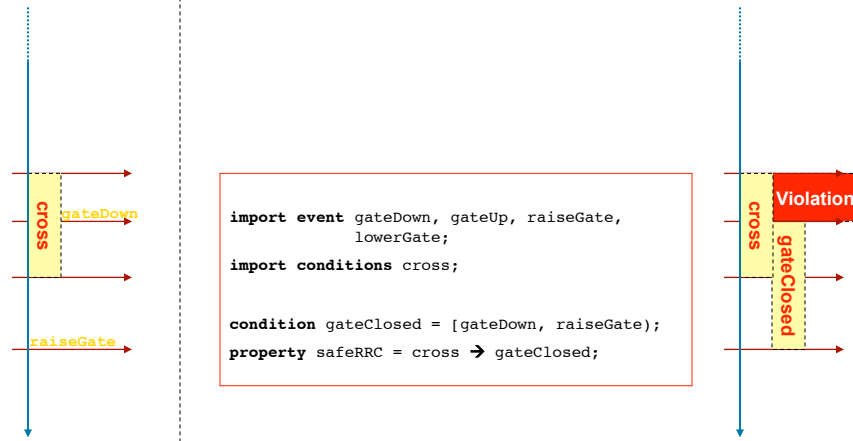


## MEDL

- ▶ Meta Event Definition Language
- ▶ Express requirements using the events and conditions, gathered from an execution
  - define events and conditions using incoming primitive events and checker state variables
- ▶ Describe the **safety requirements**
  - properties (conditions that must always be true)
  - alarms (events that must never be raised)
- ▶ **Independent** of the monitored system

## MEDL by example

**Railroad Crossing Property:** - If train is crossing, then gate must be closed



```
import event gateDown, gateUp, raiseGate,
lowerGate;
import conditions cross;

condition gateClosed = [gateDown, raiseGate];
property safeRRC = cross → gateClosed;
```

PEDL



MEDL  
QEST 2006



35

## Quantitative properties

- ▶ Timestamps of events
  - time of the last occurrence of event  $e$ :  $\text{time}(e)$
- ▶ Event attributes
  - Quantitative values from observations
- ▶ Auxiliary variables
  - Updated in response to events
  - Predicates over auxiliary variables define new events
  - Example: gate must be serviced every 1000 crossings

```
var int raiseCnt
gateUp → { raiseCnt' = raiseCnt + 1 }
alarm svcGate = start( raiseCnt > 1000 )
```



QEST 2006



36

## Event attributes

- ▶ Event attributes allow us to propagate quantitative values into MEDL
- ▶ Some events have implicit attributes
  - `update(x)` has the new value of `x` as attribute
- ▶ In general, we can associate any value in the event definition as an attribute

```
event newTrain(int tr, real w) =
  StartM(addTrain(t,weight)) {tr:=t,w:=weight}
```

- ▶ An event attribute can be used in expressions as

```
newTrain → { totalWght' = totalWght + newTrain.w }
```



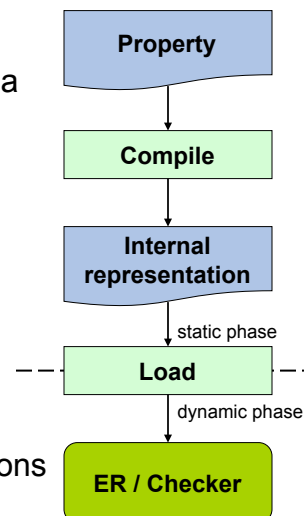
QEST 2006



37

## Implementation

- ▶ Static phase
  - PEDL and MEDL are compiled into a graph representation
- ▶ Dynamic phase
  - Event recognizer interprets PEDL graph on each observation sent by the filter
  - Checker interprets MEDL graph on each event/condition change sent by the event recognizer
  - Lazy evaluation driven by observations



QEST 2006



38

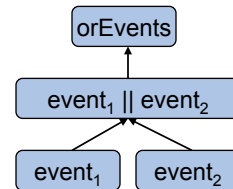
## Static phase: property graphs

- ▶ Each event and condition is a node
  - `event` stores time of the last occurrence
  - `condition` stores truth value
- ▶ Composition is represented by operator nodes
  - Each type of operator has its evaluation method, e.g. disjunction operator for events computes maximum of the values in its children
- ▶ Graphs cannot have algebraic loops

$e_1 = e_1 \parallel e_2$  a problem!

$e_1 = \text{start}([e_2, \text{end}[e_1 \text{ when } c, e_3]])$  OK

- Occurrence of  $e_1$  affects future occurrences



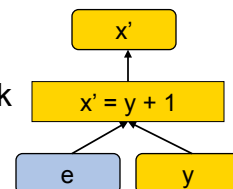
QEST 2006



39

## Static phase: quantitative properties

- ▶ Auxiliary variables and their updates are also represented as nodes
- ▶ Variable node stores the value of the variable
  - Update node does not have a value of its own
- $e \rightarrow \{ x' = y + 1 \}$
- ▶ If an event triggers update of a variable, it cannot be defined in terms of that variable
- ▶ Algebraic loops are disallowed
  - “New” values and “old” values can break loops and affect evaluation order



QEST 2006



40

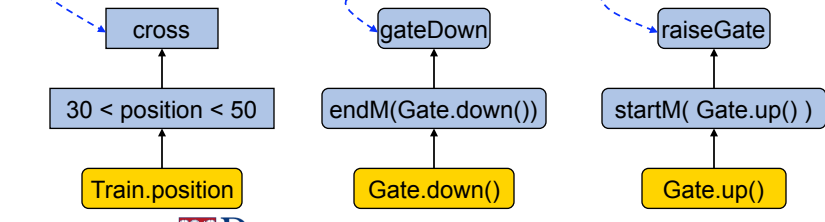
# PEDL Graph

```

export event gateDown, raiseGate;
export condition cross;

Train.position;          monobj
Gate.up();              monmeth
Gate.down();

condition cross = (30 < RRC.position)
&& (RRC.position < 50); event
gateDown = endM(Gate.down()); event
raiseGate = startM(Gate.up());
    
```



QEST 2006

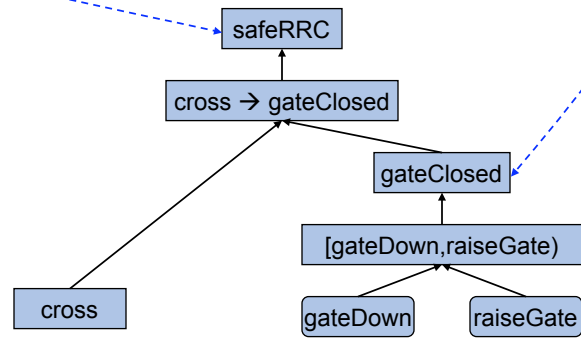


41

# MEDL Graph

```

import event gateDown, gateUp, raiseGate, lowerGate;
import conditions cross;
condition gateClosed = [gateDown, raiseGate];
property safeRRC = cross → gateClosed;
    
```



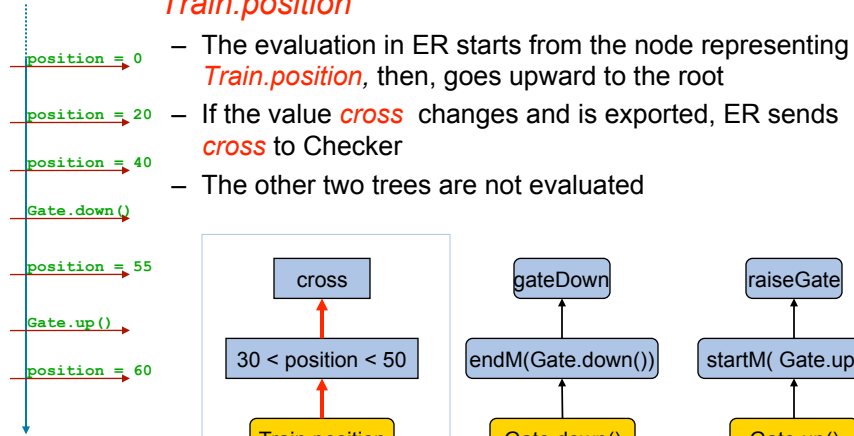
QEST 2006



42

# Dynamic Phase: Evaluation in ER

- Consider an update on a monitored variable *Train.position*



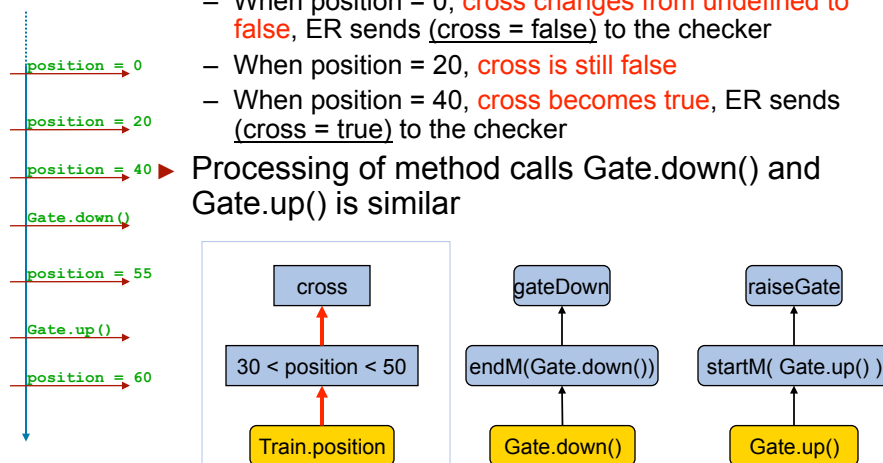
QEST 2006



43

# Evaluation in ER

- Here,
  - When position = 0, *cross* changes from undefined to false, ER sends (*cross = false*) to the checker
  - When position = 20, *cross* is still false
  - When position = 40, *cross* becomes true, ER sends (*cross = true*) to the checker
- Processing of method calls Gate.down() and Gate.up() is similar



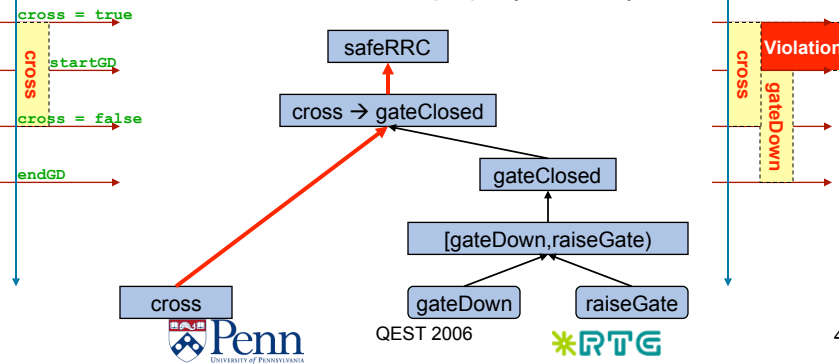
QEST 2006



44

## Evaluation in Checker

- ▶ On each event received from ER,
  - The evaluation starts from leaves (from the node corresponding to events received from ER)
  - Traverses upward to the root
- ▶ At roots, check for violations
  - If an occurred event is in the alarm list, notify users
  - If a false condition is in the property list, notify users



45

## Algorithm

- ▶ Assign a *height* to each node in the graph
- ▶ Maintain an evaluation list sorted by height
- ▶ For each new state
  - Add all occurred primitive events and changed conditions to the evaluation list at height 0
  - For each event/condition in the evaluation list,
    - Call `evaluate()` method
    - If changed, add its parent to the evaluation list (if not already in)
  - When the list is empty
    - In ER, maintain a list of exported events and conditions
      - Send occurred events/changed condition to checker
    - In Checker, maintain lists of alarms and properties
      - If an event in the alarm list occurs, notify users
      - If a condition in the property list is false, notify users
    - Copy new values of auxiliary variables into old values

46

## MaC extensions

- ▶ Steering
  - Feedback from property evaluation
- ▶ Support for dynamic properties
  - Dynamically created objects and indexed properties
- ▶ Support for timing properties
  - Time-driven evaluation
- ▶ Support for quality-of-service and probabilistic properties



QEST 2006



47

## Steering

- ▶ Steering provides feedback from the monitor to the system
- ▶ Steering actions triggered by events
  - `SEviolation → { invoke change2SC }`
  - Values can be calculated from observations
- ▶ SADL (Steering Action Definition Language)
  - Specifies actions to be taken
  - Describes conditions when it is safe to apply actions

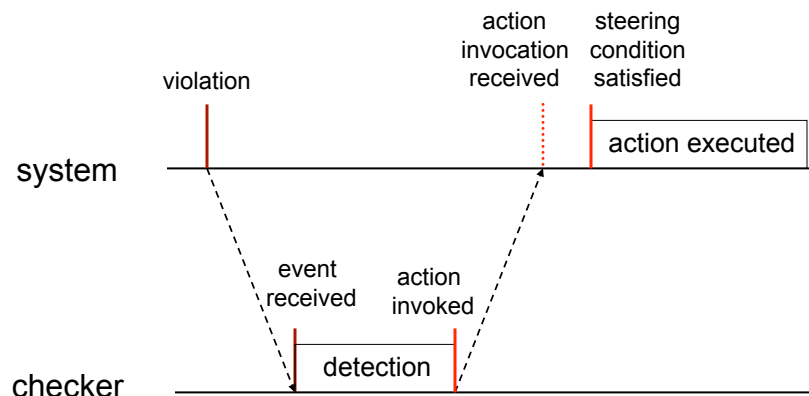


QEST 2006



48

## Steering process



## Steering and adaptation

- ▶ Steering is not a recovery/adaptation mechanism
  - It is a vehicle to invoke a built-in mechanism
- ▶ System should be ready to receive feedback
  - User specifies when it is safe to steer and what is the appropriate action
- ▶ When can a system be effectively steered?
  - the system is designed for run-time adjustments
    - e.g., Simplex architecture
  - the system naturally offers a degree of tolerance
    - e.g., routing protocols: flush buffers when performance decreases

## Steering as run manager

- ▶ Restart simulations
  - Logical criterion for “goodness” checked during the run
  - Runs that are not good are terminated as soon as the check fails
  - New runs can be restarted with simulation parameters computed from observations in the previous runs
    - Or determined statically
- ▶ If simulator supports, run-time adaptation of parameters can be done
- ▶ For interactive simulations, steering can supply new inputs, computed from past observations



QEST 2006



51

## What's in a name: indexing

- ▶ What if we have two tracks instead of one?
  - Track is safe if gate is closed when train is crossing on that track
    - `safeTrack1 = cross1 → gateClosed`
    - `safeTrack2 = cross2 → gateClosed`
  - For fixed number of objects, properties can be duplicated for each object
- ▶ Works for toy examples
  - Large number of objects – cumbersome and inefficient
  - Dynamically added objects – impossible



QEST 2006



52

## Dynamic MEDL

- ▶ We introduce indexed names and implicitly quantify over indices
- ▶ New data type `indexSet` supports adding and removing values
- ▶ Values are added by incoming events

- Suppose we can add tracks dynamically:

```

indexSet tracks
import event addTrack(tId t)
property trackSafe(tId t) = cross(t) → gateClosed
addTrack → { tracks.add( addTrack.t ) }

```



QEST 2006



53

## Beyond dynamic MEDL

- ▶ Explicit quantification and aggregation over index sets is possible
- ▶ First-order temporal logics are highly undecidable in general
- ▶ At run time, we work with concrete values and can efficiently evaluate “first-order MEDL”
  - Linear in the size of the trace
  - Exponential in the number of quantification nestings
- ▶ In practice, dynamic MEDL has been sufficient



QEST 2006



54

## RT-MaC: timing properties

- ▶ Requirement: trains should clear intersection fast enough

```
alarm slowTrain =
  time( end(cross) ) - time( start(cross) ) < 100
```

- ▶ Problems:
  - There is no alarm if train stops in the intersection
  - Alarm can be raised long time after violation occurs
  - Besides, syntax gets cumbersome for complex timing properties



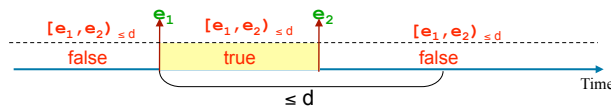
QUEST 2006



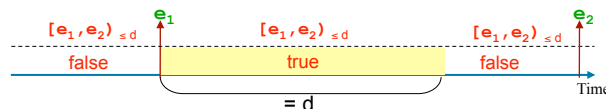
55

## Added syntax: timed interval

- ▶ Timed Interval:  $[e_1, e_2)_{\leq d}$ 
  - Becomes true when  $e_1$  occurs
  - Becomes false when  $e_2$  occurs within  $d$  time units



- Becomes false when  $d$  time units are up



- ▶  $[e_1, e_2) = [e_1, e_2)_{\leq \infty}$



QUEST 2006



56

## Added syntax: time-triggered event

- ▶ Time-triggered event:  $e + d$ 
  - An event  $e+d$  is raised at  $d$  time units after the occurrence of  $e$
- ▶ Requirement: trains should clear intersection fast enough

```
alarm slowTrain = start(cross)+100 when cross
```



- ▶ Solve the earlier problems:
  - If the train is slow, an alarm is raised at 100 time units after the train starts crossing
    - Even if the train stops in the intersection



QEST 2006



57

## Time-driven evaluation

- ▶ Semantics:
  - Evaluate time-triggered event right at  $d$  time units after the occurrence of  $e$
- ▶ How do we know it is time to evaluate?
  - Cannot set timer on checker clock!
- ▶ Real time
  - Option 1: set a timer in the filter to expire after  $d$
  - Option 2: heartbeat events
    - Bounded delay in evaluation
- ▶ Simulation time
  - Wait until simulator produces a larger timestamp



QEST 2006



58

## Probabilistic properties

- ▶ Probabilistic events
  - Given that an event  $e_1$  occurs, what is the probability that  $e_2$  will occur
- ▶ Examples
  - Given that train starts crossing, the probability that the train will not finish crossing within 100s is at most 0.2
  - When gate closes, the probability that the train will come within 20s is at least 0.8
- ▶ Collect statistics from the trace and use statistical analysis (hypothesis testing) to support checking



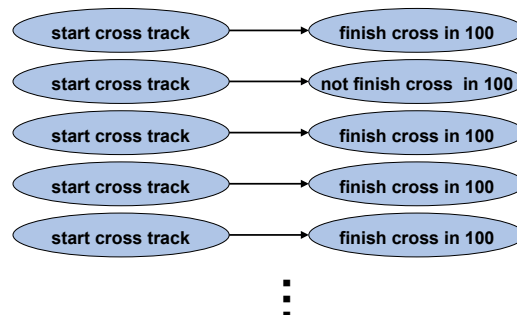
QEST 2006



59

## Conventional approach

- ▶ Multiple experiments
  1. Trigger  $e_1$  X times
  2. See how many times  $e_2$  has occurred



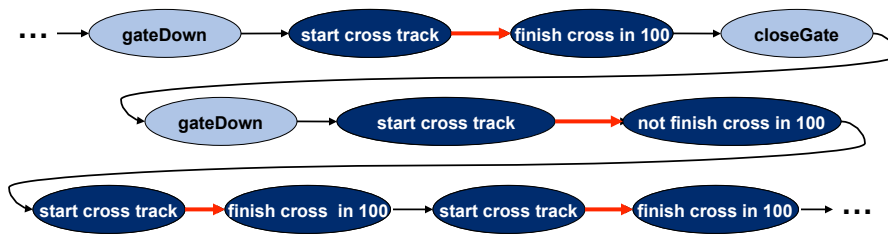
QEST 2006



60

## Collect samples during runtime

- ▶ Estimate a probability from only one execution path
  - The probabilistic properties that we can check must have repetitive behaviors
  - Count the number of occurrences of  $e_2$  against those of  $e_1$ 
    - Count  $e_2$  only when it occurs at the same time as  $e_1$  or after  $e_1$
    - I.e, count  $e_2$  only when  $e_2$  and  $(e_1 \text{ or } \text{end}([e_1, e_2]))$  occur at the same time



## Syntax

- ▶ Probabilistic event syntax
  - $e_2 \text{ prob}( > p, e_1 )$
  - $p$  is the probabilistic bound
- ▶ Example
  - Only allow trains to cross track slowly with probability at most 0.2

```
event slowTrain = start(cross)+100 when cross
alarm probSlowTrain = slowTrain prob(> 0.2, start(cross))
```

## Estimating probability

- ▶ Estimating probability  $p'$  from the actual program execution for  $e_2$

$$p' = \frac{\text{prob}( > p, e_1 )}{|\text{occurrences of } e_1|} = \frac{|\text{occurrences of } e_2 \ \&\& \ (e_1 \ || \ \text{end}([e_1, e_2]))|}{|\text{occurrences of } e_1|}$$

- ▶ Only allow trains to cross track slowly with probability at most 0.2

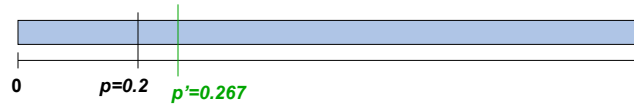
– **alarm probSlowTrain = slowTrain prob(> 0.2, start(cross))**

$$p' = \frac{|\text{slowTrain} \ \&\& \ (\text{start(cross)} \ || \ \text{end}([\text{start(cross), slowTrain}]))|}{|\text{start(cross)}|}$$

→  $|\text{slowTrain} \ \&\& \ (\text{start(cross)} \ || \ \text{end}([\text{start(cross), slowTrain}]))| = 12$

→  $|\text{start(cross)}| = 45$

→  $p' = 12 / 45 = 0.267$



QUEST 2006



63

## Estimating probability: z-score

- ▶ Binomial distribution (Success: slow train, Fail: fast train)
  - When sample is large enough, it is approximately normal distribution
- ▶ Use z-score to calculate how far apart  $p$  and  $p'$  are

$$z = \frac{p' - p}{\sqrt{\frac{p(1-p)}{n}}}$$

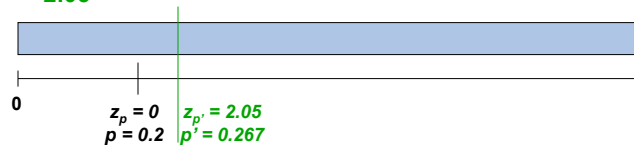
$n = |\text{occurrences of } e_1|$

- Sign of  $z$  says which direction
  - +  $z$  says  $p' > p$
  - $z$  says  $p' < p$
- Value of  $z$  says how far apart  $p'$  and  $p$

- ▶ Only allow trains to cross track slowly with probability at most 0.2

→  $p = 0.2 \quad p' = 0.267$

→  $z_{p'} = + 2.05$



QUEST 2006



64

## Compare using hypothesis testing

- ▶ Given
  - z-score of the estimated probability
    - In our example,  $z_{p'} = 2.05$
  - Set up hypotheses
    - $H_0: p' \leq 0.2$  (no alarm)
    - $H_A: p' > 0.2$  (raise alarm)
  - A critical value  $c$ : a threshold chosen by using significance level  $\alpha$ 
    - Significance level  $\alpha$  is the probability of mistakenly rejecting  $H_0$  (say raise alarm) when it is true (no violation)
    - If we choose  $c$  corresponding to  $\alpha = 0.05$ , then there is (only) a 5% probability that we will reject  $H_0$  if it is true.
    - Conventional significance level
      - $\alpha = 0.05$  (rejection is moderate evidence against  $H_0$ )
      - $\alpha = 0.01$  (rejection is strong evidence against  $H_0$ )
    - When  $\alpha = 0.05$ ,  $c = z_\alpha = 1.96$



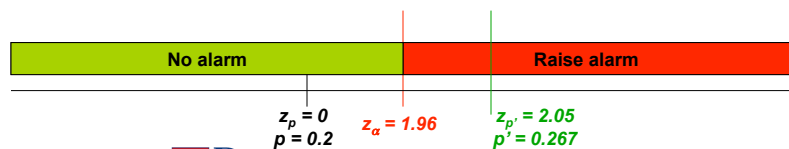
QEST 2006



65

## Compare using hypothesis testing

- ▶ Hypotheses
  - $H_0: p' \leq 0.2$  (no alarm)
  - $H_A: p' > 0.2$  (raise alarm)
- ▶ Decide: **alarm probSlowTrain = slowTrain prob(> 0.2, start(cross))**
  - Reject  $H_0$  (raise alarm):
    - $z_{p'} \geq z_\alpha$  [means  $p' > 0.2$  with only a 5% probability that we're wrong]
  - Accept  $H_0$  no alarm:
    - Otherwise
- ▶ Since  $2.05 \geq 1.96$ , we reject  $H_0$  and raise an alarm



QEST 2006



66

## Probabilistic Checking

- ▶ Can be mimicked using auxiliary variables in existing MaC
  - Can be more flexible
  
- ▶ New syntax in RT-MaC
  - Provide convenience
  - Ensure that statistical support is always used



QEST 2006



67

## Outline

- ▶ Motivation and overview
- ▶ MaC framework
- ▶ **Applications**
  - Network simulation case study
  - Control of MAV swarms
  - Simplex architecture case study



QEST 2006



68

## Case study: Verisim

- ▶ Verisim is an instantiation of the MaC architecture for network simulations
  - Large and very detailed traces make direct inspection of traces impractical
  - Logical properties in addition to performance properties help find subtle bugs



QEST 2006



69

## Ad Hoc Networks

- ▶ Routing for a wireless network without the aid of a central base station
- ▶ Connections are low-bandwidth, lossy, and highly transient
- ▶ Unique routing assumptions:
  - Most routes are seldom used
  - Bandwidth must be protected
- ▶ Ad-hoc On-demand Distance Vectors (AODV) protocol



QEST 2006





70

## Routing in Mobile Networks

The diagram illustrates a network of four nodes (blue circles) connected by lines. A red arrow labeled "Movement" points upwards from the top node. The nodes are surrounded by overlapping dashed circles representing their communication ranges. The word "Routing" is written in red in the center of the network.

Movement

Routing



QUEST 2006

71

## Routing in Mobile Networks

The diagram shows the same network of four nodes as in slide 71. A red arrow points upwards from the top node, which has moved out of its previous position. The text "New Routing" is written in red, indicating a change in the network's connectivity.

New Routing

QUEST 2006

72

## AODV Protocol

### ► Rules

- If a node  $S$  needs a route to a destination  $D$  and does not have one, it floods a *route-request (RREQ)* packet through the network
- Each recipient  $R$  of this RREQ keeps a return pointer
- $R$  broadcasts the request to its neighbors if it is not  $D$  and does not have a route to  $D$
- If  $R$  is  $D$ , or has a route to  $D$ , it responds with a *route-reply (RREP)* packet using the return pointers for  $S$

### ► Can be stated as a state machine and model checked

- We want to check protocol code!

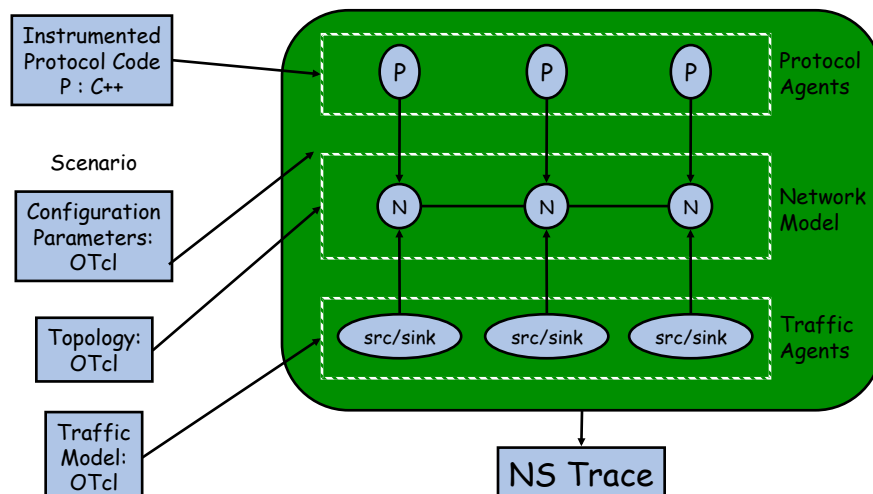


QEST 2006



73

## NS Network Simulator



QEST 2006



74

## Analysis by simulation

- ▶ Conventional analysis:
  - Manually inspect the trace – too much!
  - Calculate performance of a run
- ▶ Drawbacks:
  - Flaws may not be detected if no expected performance can be used for comparison
  - When flaws are suspected, finer means of analysis are useful
  - Some flaws do not manifest themselves as performance problems (e.g. security)



QEST 2006



75

## AODV properties

Monotone Seq No.	Node's own sequence number never decreases
Destination stops	When a packer reaches destination, it should not be forwarded
Correct forward	A packet is forwarded along best unexpired route
Destination reply	Reply to route request should have hops field set to 0
Node reply	A route is sent along the best unexpired route
RREQ Seq No.	Route request for d should have seq. no. either 0 or the last seq. no. recorded for d
Detect Route Err.	If broken route is detected, RREP increases seq. no.
Forward Route Err.	Broken route RREP is forwarded with the same seq. no.
Loop Invariant	Along every route to node d, $(-seq\_no_d, hops_d)$ strictly decreases lexicographically



QEST 2006



76

## Outline of Experiment

- ▶ Run a scenario of modest complexity
- ▶ Analyze it in Verisim using the list of 9 properties of AODV expressed in MEDL
- ▶ We instrumented simulation code for AODVv0 supplied by the CMU Monarch Project



QEST 2006



77

## Sample MEDL Alarm

```

alarm LoopInv[at][nxt][dst] =
  sendroute[at][dst] when
    ((at≠nxt) ∧ (at≠dst) ∧ (nxt≠dst) ∧
     (nexthop[at][dst] == nxt) ∧
     ((seqno[at][dst] > seqno[nxt][dst]) ∨
      ((seqno[at][dst] == seqno[nxt][dst]) ∧
       (hopcnt[at][dst] <= hopcnt[nxt][dst])))
  
```



QEST 2006



78

## Verisim experiences

- ▶ Bugs found
  - Destination reply error
    - hop count not initialized to 0
  - Forward route error
    - sequence number not incremented
  - Node reply error
    - conditionals for sending RREP are buggy
- ▶ Simulator is more efficient than checker on complex properties
  - Robust tool vs. early prototype



QEST 2006



79

## Verisim modes

- ▶ On-line mode
  - Checker runs concurrently with the simulator
  - Works for simple properties and relatively short simulations
- ▶ Off-line mode
  - Trace is produced and stored, then fed into the checker
  - Trace can be re-analyzed multiple times



QEST 2006



80

## Debugging strategies

- ▶ Once a bug is encountered, checker generates lots of alarms
- ▶ With off-line checking, two strategies are possible
- ▶ “Repair first bug”
  - Fix the problem and re-run simulation
  - Many simulation runs are needed
- ▶ “Tune” the property
  - Adjust checker state to mask the problem
  - Re-run checking on the same trace



QEST 2006



81

## Case study: control of MAV swarms

- ▶ Collaboration with NRL
- ▶ Construction of global patterns via local rules
  - $F = G m_1 m_2 / r$
  - $F$  repulsive if  $r < R$ ; else attractive
  - Pattern forms in close proximity
- ▶ Vulnerable to turbulence
  - Size < 6”, weight 50 – 70 gr
- ▶ Need external impulse to reform

Unmanned air vehicle (UAV)



Using Artificial Physics, MAVs form a hexagonal lattice sensing grid

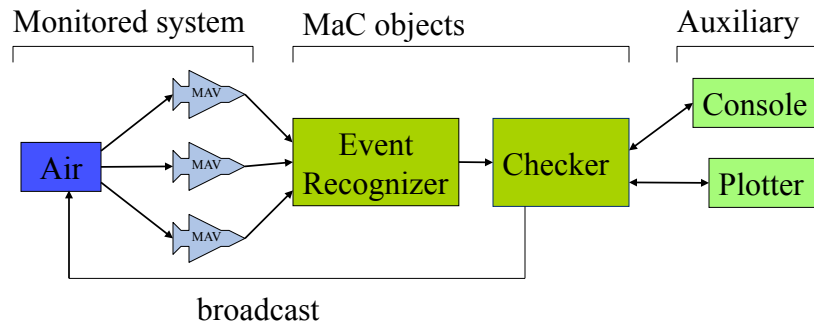


QEST 2006



82

## Monitoring setup

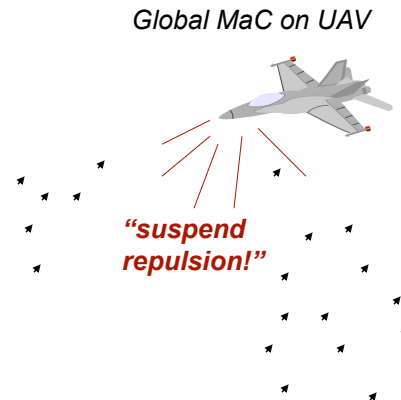


## Pattern formation monitoring

- ▶ **Pattern formation:**
  - monitored entity: distance to a neighbor
  - imported event:  $MAV_{alert} = 0.25 \cdot R \leq \text{distance} \leq 0.75 \cdot R$
- ▶ **Pattern alarm: alerts count increases sharply**
  - Count alerts within a window
  - Average over three consecutive windows
  - An increase of over 15% triggers alarm

## Steering (repair)

- ▶ Monitor cannot address individual MAVs
  - commands are broadcast
- ▶ Two steering actions are used:
  - After a pattern alarm, repulsion between close MAVs is suspended. MAVs are drawn together
  - After a fixed interval, repulsion is restored, restarting the formation process



## Monitored requirements (MEDL)

```
ReqSpec HexPattern

import event MAValert, startPgm;

var long currInterval;
var int count0, count1, count2, prevAvg, currAvg;

event startPeriod = start(time(MAValert) - currInterval > 10000);

property NoPattern = (currAvg <= prevAvg*1.15 + 100) || (prevAvg == -1);

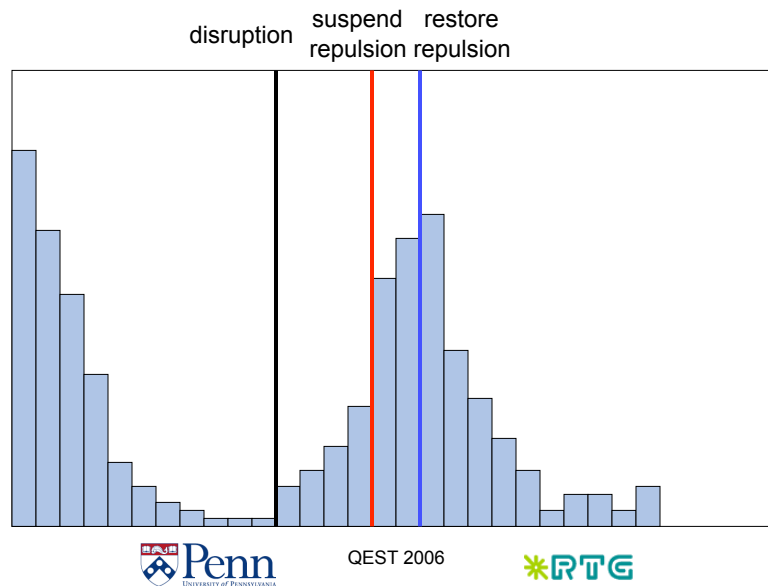
startPgm -> {
  currInterval = time(startPgm);
  count0 = 0; prevAvg = -1; currAvg = -1; }

startPeriod -> {
  currInterval = currInterval + 10000;
  prevAvg = currAvg; currAvg = (curr0+curr1+curr2)/3;
  count2 = count1; count1 = count0; count0 = 0; }

MAValert -> { count0 = count0 + 1; }

End
```

## Simulation run and control events

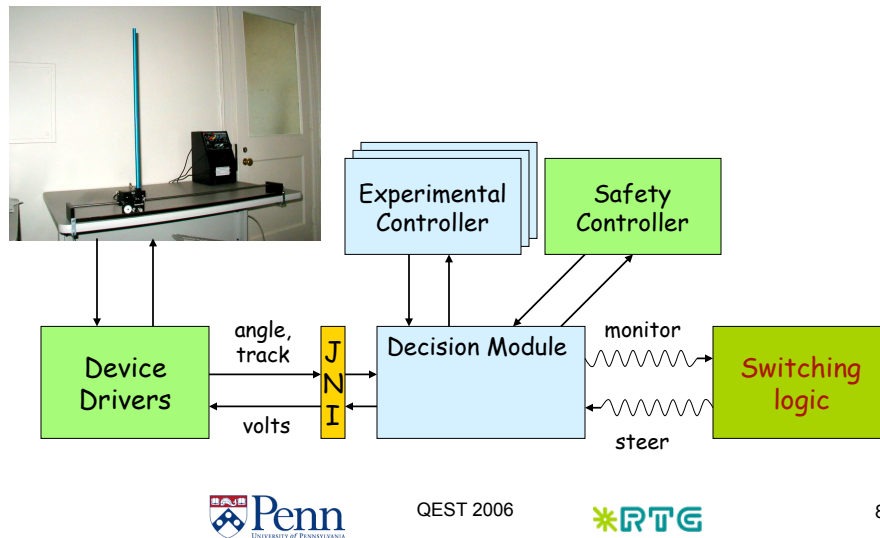


87

## Simplex architecture case study

- ▶ Simplex architecture for control systems provides hot-swapping of experimental controllers
  - Enhanced performance, uncertain stability
  - If stability is compromised, switch to a safety controller
  - Stability is checked by computing the safety envelope of the system
- ▶ Case study
  - Control system is an inverted pendulum
  - Use checker to compute safety envelope
  - Use steering to switch between controllers

## Inverted Pendulum in MaC



## Summary

- ▶ Analysis of logical and quality of service properties of simulation runs help in evaluating the model
- ▶ Specifying properties in a language with formal semantics, and checking them with a checker for the language add precision and flexibility
- ▶ MaC is an architecture for monitoring and checking of properties of executions
  - Includes languages for property specification
- ▶ Several case studies testify to MaC's utility

## Acknowledgements

- ▶ **MaC team**
  - Insup Lee
  - Sampath Kannan
  - Usa Sammapun
  - Arvind Easwaran
- ▶ **Former members**
  - Mahesh Viswanathan (UIUC)
  - Moonjoo Kim (Postech U., Korea)
- ▶ **Generous support from**
  - ONR, NSF, ARO



QEST 2006



91

## MaC bibliography

- ▶ [KKL+04] Java-MaC: a rigorous run-time assurance tool for Java programs, M. Kim, S. Kannan, I. Lee, O. Sokolsky, M. Viswanathan, *Formal Methods in Systems Design*, Vol. 24, No. 2, March 2004
  - Comprehensive introduction to basic MaC
- ▶ [SLS05] RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties, U. Sammapun, I. Lee and O. Sokolsky, *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA '05)*, August 2005
  - MaC extensions for real-time and probability
- ▶ [SSL05] Run-time checking of dynamic properties, O. Sokolsky, U. Sammapun, I. Lee, and J. Kim, *Proceedings of the Fifth Workshop on Runtime Verification (RV '05)*, July 2005
  - MaC extensions for dynamic properties
- ▶ [BGK02] Verisim: Formal analysis of network simulations, K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan, *IEEE Transactions on Software Engineering*, Vol. 28, No. 2, February 2002, pp. 129-145.
  - Case study of AODV protocol simulation
- ▶ [GSS99] Distributed spatial control, global monitoring and steering of mobile physical agents, D. Gordon, W. Spears, O. Sokolsky, and I. Lee. *Proceedings of the IEEE International Conference on Information, Intelligence, and Systems*, pp. 681-688, November 1999.
  - Case study of MAV simulation



QEST 2006



92