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
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

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
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

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
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”

Model checking
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

![Diagram](image-url)
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

Checking a property of a trace

► Satisfaction relation
  \[ t: \begin{array}{c}
  \text{\textbullet} \rightarrow \text{\textbullet} \rightarrow \text{\textbullet} \rightarrow \text{\textbullet} \\
  \end{array} \quad \vDash \quad P \]
► Simple algorithm, linear in the trace length
► At each step, trace becomes longer
  \[ t': \begin{array}{c}
  \text{\textbullet} \rightarrow \text{\textbullet} \rightarrow \text{\textbullet} \rightarrow \text{\textbullet} \rightarrow \text{\textbullet} \\
  \end{array} \quad \vDash \quad P \]
► Furthermore, traces are too big to store
► Need a different approach
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

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?
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

MaC architecture
MaC architecture - simulation

MaC languages

Run-time state:
- control locations
- object state
- local variables

Abstract state:
- events
- conditions
- auxiliary variables

► 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**

Example: hundred years’ war

- The war is a condition
- Battles are events
  - Battle durations notwithstanding
- Events change the state of conditions
  - end(War)=FallOfBordeaux
  - FinalDefeat = [FallOfParis,FallOfBordeaux)
Logical foundation

- **LEC**: 2-sorted logic: events and conditions

- Syntax:
  
  \[ E ::= e \mid \text{start}(C) \mid \text{end}(C) \mid E_1 \lor E_2 \mid E_1 \land E_2 \mid E \text{ when } C \]
  
  \[ C ::= c \mid [E_1, E_2] \mid \neg C \mid C_1 \lor C_2 \mid C_1 \land C_2 \]

- Operator \([. , .]\) pairs events to define an interval

- Operators \text{start} and \text{end} define the events at the instant when conditions change their value

![Diagram showing event intervals and conditions](image)

---

Operators

- Interval operator: \([e_1, e_2]\)
  
  - Becomes true when \(e_1\) occurs
  
  - Becomes false when \(e_2\) occurs

![Diagram showing interval operator](image)

- When operator \(e\) when \(c\)

![Diagram showing when operator](image)
Semantic function

Formally, a model $M = (S, \tau, L_C, L_E)$
- $S = \{s_0, s_1, \ldots\}$ 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 $t$

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
Semantic function

\[ M, t \models c \iff D^c_M(t) = \text{true} \]

\[ M, t \models e_k (e_k \text{ primitive}) \iff \exists s_t \text{ such that } \tau(s_t) = t \]

\[ M, t \models \text{start}(c) \iff \exists s_t \text{ such that } \tau(s_t) = t \text{ and } M, \tau(s_t) \models c \]

\[ M, t \models \text{end}(c) \iff \exists s_t \text{ such that } \tau(s_t) = t \text{ and } M, \tau(s_t) \not\models c \]

Denotation for Conditions

\[ [e_k \text{ primitive}] \quad D^c_M(e_k) = \mathcal{L}_c(s_t, e_k) \text{, where } \tau(s_t) \leq t \text{ and for all } s_j \quad (j > t) \quad \tau(s_j) > t \]

\[ [\text{defined}] \quad D^c_M(\text{defined}(c)) = \begin{cases} \text{true} & \text{if } D^c_M(c) \neq \Lambda \\ \text{false} & \text{otherwise} \end{cases} \]

\[ [\text{pair}] \quad D^c_M([e_1, e_2]) = \begin{cases} \text{true} & \text{if there exists } t_0 \leq t \text{ such that } M, t_0 \models e_1 \\ \text{false} & \text{otherwise} \end{cases} \]

\[ [\text{negation}] \quad D^c_M(\neg c) = \begin{cases} \Lambda & \text{if } D^c_M(c) = \Lambda \\ \text{false} & \text{if } D^c_M(c) = \text{true} \end{cases} \]

\[ [\text{disjunction}] \quad D^c_M(c_1 || c_2) = \begin{cases} \text{true} & \text{if } D^c_M(c_1) = \text{false} \quad \text{or } D^c_M(c_2) = \text{false} \\ \text{false} & \text{if } D^c_M(c_1) = \text{true} \quad \text{and } D^c_M(c_2) = \text{false} \\ \Lambda & \text{otherwise} \end{cases} \]

\[ [\text{conjunction}] \quad D^c_M(c_1 \& c_2) = D^c_M([l_1(c_1) \& l_2(c_2)]) \]

\[ [\text{implication}] \quad D^c_M(c_1 \Rightarrow c_2) = D^c_M(l_1(c_1) | l_2(c_2)) \]
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

PEDL constructs

- Declaration of monitored variables
- Definitions of primitive conditions
  - Predicates over monitored variables
- Definitions of primitive events
  - Update to a monitored variable \( x \): \texttt{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
**PEDL by example**

### Abstraction
- When train position is between 30 and 50
- When gate starts/ends being down

```pedl
position = 0
position = 20
position = 30
```

- Call `Gate.down()` when position = 30
- Call `Gate.up()` when position = 50

```pedl
export event gateDown, raiseGate;
export condition cross;
monobj
Train.position;
monmeth Gate.up();
monmeth Gate.down();
condition cross = (30 < RRC.position) && (RRC.position < 50);
```

```pedl
event gateDown = endM(Gate.down());
event raiseGate = startM(Gate.up());
```

---

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

```plaintext
import event gateDown, gateUp, raiseGate, lowerGate;
import conditions cross;
condition gateClosed = [gateDown, raiseGate);
property safeRRC = cross ⟽ gateClosed;
```

**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

```plaintext
var int raiseCnt
gateUp ⟽ { raiseCnt' = raiseCnt + 1 }
alarm svcGate = start( raiseCnt > 1000 )
```
Event attributes

- Event attributes allow us to propagate quantitative values into MEDL
- Some events have implicit attributes
  - \texttt{update}(x) has the new value of \( x \) as attribute
- In general, we can associate any value in the event definition as an attribute
  
  \[
  \text{event newTrain(int tr, real w) =} \\
  \quad \text{StartM(addTrain(t,weight)) \{ tr:=t, w:=weight \}}
  \]
- An event attribute can be used in expressions as
  
  \[
  \text{newTrain} \rightarrow \{ \text{totalWght}' = \text{totalWght} + \text{newTrain.w} \}
  \]

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
Static phase: property graphs

- Each event and condition is a node
  - event node stores time of the last occurrence
  - condition node 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
  \[ e_1 = e_1 \parallel e_2 \text{ a problem!} \]
  \[ e_1 = \text{start}([e_2, \text{end}[e_1 \text{ when } c, e_3]]) \text{ OK} \]
  - Occurrence of \( e_1 \) affects future occurrences

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
**PEDL Graph**

```plaintext
export event gateDown, raiseGate;
export condition cross;

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

condition cross = (30 < RRC.position) && (RRC.position < 50);

event
gateDown = endM(Gate.down());
event
raiseGate = startM(Gate.up());
```

**MEDL Graph**

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

condition gateClosed = (gateDown, raiseGate);

property safeRRC = cross | gateClosed;
```
Consider an update on a monitored variable $\text{Train.position}$

- The evaluation in ER starts from the node representing $\text{Train.position}$, then, goes upward to the root
- If the value $\text{cross}$ changes and is exported, ER sends $\text{cross}$ to Checker
- The other two trees are not evaluated

Here,
- When position = 0, $\text{cross}$ changes from undefined to false, ER sends (cross = false) to the checker
- When position = 20, $\text{cross}$ is still false
- When position = 40, $\text{cross}$ becomes true, ER sends (cross = true) to the checker

Processing of method calls Gate.down() and Gate.up() is similar
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

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
## 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**

## Steering

- Steering provides feedback from the monitor to the system
- Steering actions triggered by events
  
  \[ \text{SEviolation} \rightarrow \{ \text{invoke change2SC} \} \]
  
  - 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
Steering process

- violation
- action invocation received
- steering condition satisfied
- action executed
- event received
- action invoked
- detection
- action invocation received

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

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
    safeTrack_1 = cross_1 → gateClosed
    safeTrack_2 = cross_2 → 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
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:

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

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
RT-MaC: timing properties

► Requirement: trains should clear intersection fast enough

\[
\text{alarm slowTrain} = \\
\text{time( end(cross) )} - \text{time( start(cross) )} < 100
\]

► 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

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

\[\begin{align*}
\text{false} & \quad \text{true} & \quad \text{false} \\
\[e_1, e_2]\ & \quad \leq d & \quad \leq d
\end{align*}\]

  – Becomes false when \(d\) time units are up

\[\begin{align*}
\text{false} & \quad \text{true} & \quad \text{false} \\
\[e_1, e_2]\ & \quad = d & \quad \leq d
\end{align*}\]

► \([e_1, e_2] = [e_1, e_2] \leq \infty\)
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
  
  $\text{alarm slowTrain} = \text{start(cross)}+100 \text{ when cross}$

- 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

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
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

Conventional approach

► Multiple experiments
  1. Trigger $e_1$ X times
  2. See how many times $e_2$ has occurred
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 \((e_1\) or end\((e_1, e_2))\) occur at the same time

Syntax

- Probabilistic event syntax
  - \(e_2\) prob\((> p, e_1)\)
  - \(p\) is the probabilistic bound

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

\[
\text{event slowTrain = start(cross)+100 when cross} \\
\text{alarm probSlowTrain = slowTrain prob\( (> 0.2, \text{start(cross)})\)}
\]
Estimating probability

- Estimating probability $p'$ from the actual program execution for $e_2$
  \[ p' = \frac{\text{occurrences of } e_2 \&\& (e_2|\text{end}(e_2))}{\text{occurrences of } e_1} \]

- Only allow trains to cross track slowly with probability at most 0.2
  \[ \text{alarm probSlowTrain} = \text{slowTrain prob}(0.2, \text{start(cross))} \]
  \[ p' = \frac{\text{slowTrain \&\& (start(cross))|| end((start(cross), slowTrain))}}{\text{start(cross)}} \]
  \[ \Rightarrow | \text{slowTrain\&\& (start(cross))|| end((start(cross), slowTrain))} | = 12 \]
  \[ \Rightarrow | \text{start(cross)} | = 45 \]
  \[ \Rightarrow p' = \frac{12}{45} = 0.267 \]

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
  \[ z = \frac{p' - p}{\sqrt{\frac{p(1-p)}{n}}} \]
  \[ n = \text{occurrences of } e_1 \]

- Only allow trains to cross track slowly with probability at most 0.2
  \[ p = 0.2 \quad p' = 0.267 \]
  \[ z_{p'} = +2.05 \]
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 $\alpha = 0.05$, $c = z_\alpha = 1.96$

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
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

Outline

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

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
Routing in Mobile Networks

Movement

Routing

Routing in Mobile Networks

New Routing
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!

NS Network Simulator

Diagram showing the interaction between protocol agents, network model, traffic agents, and OTcl configurations.
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)

AODV properties

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

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])))])
```
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

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
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

Unmanned air vehicle (UAV)

Case study: control of MAV swarms

► Collaboration with NRL
► Construction of global patterns via local rules
  \[ F = G \frac{m_1 m_2}{r} \]
  – \( F \) 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

Using Artificial Physics, MAVs form a hexagonal lattice sensing grid
Monitoring setup

Pattern formation monitoring

► Pattern formation:
  – monitored entity: distance to a neighbor
  – imported event: MAValert = 0.25*R ≤ distance ≤ 0.75*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

Monitor requirements (MEDL)

```plaintext
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

- Disruption
- Suspend
- Restore
- Repulsion
- Repulsion

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
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

MaC bibliography

  – Comprehensive introduction to basic MaC

  – 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

  – Case study of AODV protocol simulation

  – Case study of MAV simulation