Overview and Tutorial of Real-Time Maude

Peter C. Ölveczky
University of Oslo

Based on joint work with José Meseguer and others

Outline

• Overview:
  – High-level overview
  – Applications
  – Summary of experiences

• Tutorial:
  – Specification formalism and analysis commands
  – A very simple clock example
  – Objects and messages
    * simple thermostat example
  – Fragments of CASH scheduling algorithm

Overview

Real-Time Maude: a “light-weight” extension of Maude

• Specification:
  – like Maude (equations, rewrite rules) + tick rewrite rule(s) for advancing time
  – useful specification techniques for object-oriented real-time systems
Real-Time Maude Overview II

• Light-weight extension of Maude's analysis commands
  – execution strategies for tick rules for dense time
  – timed rewriting for simulation/prototyping
  – time bounded and unbounded explicit-state reachability analysis
    (search) and LTL model checking
    * can apply time bounded model checking to infinite-state systems
  
• Completeness of analysis?
  – analyze all behaviors for dense time?
  – Real-Time Maude much more expressive than timed automata, etc.
  – still: complete analysis for useful classes of OO systems

Real-Time Overview III

Inherits from Maude:

• Expressiveness
  – model many different kinds of systems
  – model large systems with different “features”
  
• Simple and intuitive formalism (equations and rules)
  – intuitive semantics
  
• Natural specification style for object-based real-time systems
  
• Range of analysis features (simulation, reachability analysis, model checking, . . .)
  
• High performance

Some Applications I: Comm. Protocols

1. AER/NCA active networks protocol suite for reliable, scalable, adaptive multicast
  – large protocol suite
  – found all “known” errors (not told to us)
  – found significant errors not found by traditional testing by the protocol developers

2. (Parts of) NORM multicast protocol developed by IETF

Some Applications II: Scheduling Algorithms

Variation of CASH scheduling algorithm (with M. Caccamo)

• CASH queue of unused execution budgets
  
• simulation: queue can grow beyond any bound
  – beyond the scope of finite-control formalisms and standard decision procedures for real-time systems
  
• specified “all possible task sets” for given set of servers
  
• search found that extension could not guarantee schedulability
  – no ingenuity in def of initial states
  – “Monte Carlo simulation” showed that it was “unlikely” that overflow could be found by simulation
Some Applications III: OGDC algorithm for WSNs

OGDC coverage algorithm for wireless sensor networks (WSNs)
- developed by Jennifer Hou and Honghai Zhang
  - simulated in ns-2 with wireless extension by Hou & Zhang
- WSN challenging domain (in OGDC):
  - new forms of communication (radio: broadcast to nodes within transmission range; with delay)
  - model areas, angles, ...
  - probabilistic behaviors
  - analyze performance (and correctness)
- need expressive formalism

Some Applications III: OGDC algorithms for WSNs (cont.)

Formally specified, simulated, and model checked OGDC
- communication model w/ delay
- simulation using timed rewriting:
  - 400 nodes in one round
  - 75 nodes in 50 rounds
- measured Zhang & Hou performance metrics during simulation
  - number of active nodes and % coverage at end of first round
  - % coverage + total power throughout network lifetime
- slightly different results from Z&H
  - could be because of transmission delays
- took Univ. Oslo student 4 months total

Time-Sensitive Cryptographic Protocols

- Model checking
  - only up to 6 nodes
- TTBOMK first formal model and analysis of “advanced” WSN algorithm
- Initial efforts using Real-Time Maude for WSN at Air Force Labs and UC Irvine/SRI

- Non-Zeno intruder
  - time-bounded analysis terminates
  - strategy where time bound and intruder capability increased gradually after “unsuccessful” search/model checking
- Wide-mouthed frog and Kerberos
  - found easily known flaws in WMF (search)
  - and almost all in Kerberos
Summary I

+ Expressiveness/Generality
  - diverse class of (large) state-of-the-art systems
  - different communication forms (links, area broadcast) at different levels of abstraction
+ Intuitive formalism for OO RTS (AER/NCA)
+ Spec techniques + high level of abstraction:
  - models quickly developed
+ Simulation useful for large systems
+ Expressive “queries”
  - LTL formulas with parameterized atomic propositions
+ Inherits Maude’s high performance

Summary II

- Explicit-state search and model checking
  - memory (and time) easily exhausted
  - often model check only small systems/states/durations
But found “overflow” in CASH in minutes using search
  - Only 6 nodes for OGDC
  - but no other formal tool can even specify OGDC (?)
But found WMF attack faster than only other model checking effort I know
  - Problems often found for small states (NSPK, RBP, ...)
  - Difficult for large and highly nondeterministic systems
  - RTS sometimes more deterministic than untimed systems

Summary III

Real-Time Maude does not “verify” a system
  - analysis from single initial states and up to a given duration
  - could use inductive techniques (theorem proving) via Maude’s ITP
  - narrowing: reachability analysis from (potentially) infinite number of initial states
  - ...
  - tool’s reflective capabilities allows it to be extended with analysis strategies/techniques

Tutorial
Specification Formalism

- Specification formalism:
  - algebraic equational specification for data types/functional aspects
  - rewrite rules define instantaneous transitions
  - time elapse modeled by tick rewrite rules
    \[
    \{t\} \Rightarrow \{t'\} \text{ in time } \tau \text{ if } \text{cond}
    \]
- Specification techniques for object-based specification
  - system state: multiset of objects
  - rewrite rules on subconfigurations

Example: A Retrograde Clock

Clock which shows the time:
- global state \{clock(r)\}
- dense time domain
- clock can stop at any time due to battery failure
- retrograde clock: clock(24) should be reset to clock(0)

Example: Retrograde Clock

```plaintext
(tmod DENSE-CLOCK is pr POSRAT-TIME-DOMAIN .
  ops clock stopped-clock : Time -> System .
  vars R R' : Time .

  crl [tickWhenRunning] :
    \{clock(R)\} \Rightarrow \{clock(R + R')\} \text{ in time } R'
    \text{ if } R' \leq 24 - R \text{ [nonexec]} .
  rl [tickWhenStopped] :
    \{stopped-clock(R)\} \Rightarrow \{stopped-clock(R)\}
    \text{ in time } R' \text{ [nonexec]} .
  rl [reset] : clock(24) \Rightarrow clock(0) .
  rl [batteryDies] :
    clock(R) \Rightarrow stopped-clock(R) .

endtm)
```

Time Sampling Strategies

- Tick rules "cover" the dense time domain
- ... but are not executable
- Choice of time sampling strategies to treat such rules, including
  - advance time by time \(\Delta\)
  - advance time as much as possible
- Time sampling often sufficient in practice
  - usually discrete time domain in communication, scheduling, etc.
  - things happen because timers expire, messages are read, etc.
- Maximal time sampling sound and complete for useful class of systems
Example: Setting a Time Sampling Strategy

Set time to advance by 1 in each tick rule application:

Maude> (set tick def 1 .)

• all subsequent Real-Time Maude analysis performed w.r.t. this strategy

Formal Analysis I: Symbolic Simulation

Timed rewriting simulates one possible behavior:

Maude> (trew {clock(0)} in time <= 100 .)

Result ClockedSystem :
{stopped-clock(24)} in time 100

• can trace the rewrite steps

Formal Analysis II: Search and Model Checking

• To analyze all behaviors, relative to the chosen time sampling strategy, from initial state
  – search for reachable states matching a given pattern
  – linear temporal logic model checking

Reachable state space often infinite in larger systems:
  – search/model check up to a given duration
  – reachable state space then becomes finite (most often)

• When reachable state space finite: unbounded search/model checking

• No guarantee that all “interesting” behaviors covered
  – false positives possible
  – a counterexample is a valid counterexample

Example: Search

• Can \{clock(25)\} be reached from \{clock(0)\} within time 100?
  \( (tsearch [1] \{clock(0)\} =>* \{clock(25)\} \text{ in time } <= 100 .) \)

• Finite reachable state space: unbounded search:
  \( (utsearch [1] \{clock(0)\} =>* \{clock(25)\} .) \)

• State \{clock(1/2)\} not found because of time sampling:
  \( (utsearch [1] \{clock(0)\} =>* \{clock(1/2)\} .) \)

• Deadlock possible?
  \( (utsearch [1] \{clock(0)\} =>! \{S: System\} .) \)
Temporal Logic Model Checking

- Propositional linear temporal logic
- Not metric temporal logic
  - propositions may involve elapsed time ("clocked properties")
- Time bound ⇒ termination of model checking
- Provides counterexamples
- Uses Maude's high-performance model checker

Formula:
- user-defined atomic propositions
- temporal logic operators such as \( \neg, \lor, [], <>, U \)

Example: Defining Propositions

- clock-dead: holds if clock has stopped
  - \( \text{op clock-dead} : \rightarrow \text{Prop [ctor]} \).
  - \( \text{vars} R R' : \text{Time} \).
  - \( \text{eq} \{ \text{stopped-clock(R)} \} \models \text{clock-dead} = \text{true} \).
- clock-is(r): running clock shows \( r \)
  - \( \text{op clock-is} : \text{Time} \rightarrow \text{Prop [ctor]} \).
  - \( \text{eq} \{ \text{clock(R)} \} \models \text{clock-is(R')} = (R == R') \).
- clockEqualsTime: running clock shows the current time elapse
  - \( \text{op clockEqualsTime} : \rightarrow \text{Prop [ctor]} \).
  - \( \text{eq} \{ \text{clock(R)} \} \text{ in time } R' \models \text{clockEqualsTime} = (R == R') \).

Example: Temporal Logic Model Checking

- The clock is never 25:
  - \( \text{(mc } \{ \text{clock(0)} \} \models \text{u } [ ] \sim \text{clock-is(25)} \). \)
- Clock shows elapsed time until time 24 or until it dies:
  - \( \text{(mc } \{ \text{clock(0)} \} \models \text{t clockEqualsTime U}
    \text{(clock-is(24) \lor clock-dead)}
    \text{ in time } <= 100 \). \)
- Counter-example: clock value 4 not always reached:
  - \( \text{(mc } \{ \text{clock(0)} \} \models \text{u } [ ] < \text{clock-is(4)} \). \)

Classes and Objects

Class declaration:

- \( \text{class } C | \text{att}_1 : s_1, ..., \text{att}_n : s_n \).

A object instance \( O \) of class \( C \) is a term

- \( \text{< } O : C | \text{att}_1 : \text{val}_1, ..., \text{att}_n : \text{val}_n \text{ >} \)

Example:

- \( \text{class Thermostat | currTemp : PosRat, heater : OnOff,}
  \text{desiredTemp : Nat} \).

An object might be

- \( \text{< "Kitchen" : Thermostat | currTemp : 74, heater : on,}
  \text{desiredTemp : 72 >} \)
An *instantaneous* rewrite rule could be

```
 var O : Oid .
 vars M N : Nat .

crl [turnOffHeater] :
  < O : ThermoStat | currTemp : M, heater : on, desiredTemp : N >
  =>
  < O : ThermoStat | heater : off >
  if M >= N + 6 .
```

A message can be seen as a term of sort `Msg`:

```
```

A concrete message is then

```
setDesTemp("LivingRoom", 66)
```

For transmission delays:

```
dly(setDesTemp("LivingRoom", 66), 5)
```

will be “ready” in time 5

The whole state: a *multiset* of objects and (delayed and ripe) messages:

```
{< "Kitchen" : ThermoStat | currTemp : 74, heater : on, desiredTemp : 72 >
 < "BedRoom" : ThermoStat | currTemp : 68, heater : off, desiredTemp : 66 >
 < "Bathroom" : ThermoStat | currTemp : 73, heater : off, desiredTemp : 78 >

dly(setDesTemp("BedRoom", 64), 2)
setDesTemp("Bathroom", 80))
```

- Multiset (AC) rewriting supported directly in Maude
- Rewrite subconfigurations

Rule:

```
rl [readNewDesTemp] :
  setDesTemp(O, N)
  < O : ThermoStat | >
  =>
  < O : ThermoStat | desiredTemp : N > .
```

Must also have rule where some *User* generates messages
Usually only one tick rule

\[
\text{var } C : \text{Configuration} . \ \text{var } T : \text{Time} .
\]

\[
\text{crl } \{C\} \Rightarrow \{\text{delta}(C, T)\} \text{ in time } T \text{ if } T \leq \text{mte}(C).
\]

- nondeterministic time advance
- \text{delta} defines time advance on a configuration
- \text{mte}: how much time can advance before something must happen?
- \text{delta} and \text{mte} distribute over elements and must be defined for single objects:

\[
\text{eq } \text{delta}(< O : \text{ThermoStat} | \text{currTemp} : N, \text{heater} : \text{on} >, T) = \\
< O : \text{ThermoStat} | \text{currTemp} : N + 2 * T > .
\]

\[
\text{eq } \text{delta}(< O : \text{ThermoStat} | \text{currTemp} : N, \text{heater} : \text{off} >, T) = \\
< O : \text{ThermoStat} | \text{currTemp} : N - T > .
\]

\[
\text{eq } \text{mte}(< O : \text{ThermoStat} | \text{currTemp} : M, \text{heater} : \text{on}, \text{desiredTemp} : N >) = \\
((N + 6) \text{ monus } M) / 2 .
\]

\[
\text{eq } \text{mte}(< O : \text{ThermoStat} | \text{currTemp} : M, \text{heater} : \text{off}, \text{desiredTemp} : N >) = \\
M \text{ monus } (N - 6) .
\]

---

**Modeling CASH in Real-Time Maude (I)**

Two models of CASH:

- modeling all possible task sets
  - a job can arrive at any time, and can execute for any time ($> 0$)
  - can analyze all possible behaviors for a given set of servers
- “randomly” generated jobs
  - for “Monte-Carlo” simulations

---

**Modeling CASH in Real-Time Maude (II)**

- Data types for the CASH queue, etc
- Object-based specification; the task server class Server:

\[
\text{class Server} | \\
\text{maxBudget} : \text{ NzTime}, \\
\text{period} : \text{ NzTime}, \\
\text{state} : \text{ ServerState}, \\
\text{usedOfBudget} : \text{ Time}, \\
\text{timeToDeadline} : \text{ Time}, \\
\text{timeExecuted} : \text{ Time} .
\]

- Dynamic behavior modeled by 10 rewrite rules
Modeling CASH in Real-Time Maude (III)

Dynamic behavior: job arrives at server $S_i$ while server $S_j$ is executing:

```plaintext
rl [idleToActive]:
  < $i$ : Server | period : $T_i$, state : idle,
  timeToDeadline : $T$ >
  < $j$ : Server | state : executing, timeToDeadline : $T'$ > ->
  if $(T + T_i) < T'$ then --- preempt $S_j$
    (< $i$ : Server | state : executing, timeExecuted : 0,
      timeToDeadline : $T + T_i$ >
    < $j$ : Server | state : waiting >)
  else --- wait for processor
    (< $i$ : Server | state : waiting, timeExecuted : 0,
      timeToDeadline : $T + T_i$ >
    < $j$ : Server | >)
fi .
```

Sept 25–29, 2006 UIUC’06 37

Modeling CASH in Real-Time Maude (IV)

Remaining budget larger than relative deadline:

```plaintext
op DEADLINE-MISS : -> Configuration [ctor] .
crl [deadlineMiss]:
  < $i$ : Server | state : STATE, usedOfBudget : $T$,
  timeToDeadline : $T'$, maxBudget : $Q_i$, >
  => DEADLINE-MISS
  if $(Q_i - T) > T'$
  \ STATE -- waiting or STATE -- executing .
```

Sept 25–29, 2006 UIUC’06 38

Analyzing CASH in Real-Time Maude (I)

Initial state (which should not lead to missed deadline):

```plaintext
op init2 : -> GlobalSystem .
eq init2 =
  {< s1 : Server | maxBudget : 2, period : 5,
    timeExecuted : 0, usedOfBudget : 0,
    state : idle, timeToDeadline : 0 >
  < s2 : Server | maxBudget : 4, period : 7, ... >
  [CASH: emptyQueue]
  AVAILABLE-PROCESSOR} .
```

Sept 25–29, 2006 UIUC’06 39

Analyzing CASH in Real-Time Maude (III)

Time-bounded reachability analysis

- search for state containing DEADLINE-MISS:

Maude> (tsearch [1] init2 =>*
  {DEADLINE-MISS C:Configuration} in time <= 12 .)

Solution 1
C:Configuration <= ... ; TIME_ELAPSED:Time <= 12

- hard tasks cannot be guaranteed!
- search took 140 sec.
- no ingenuity in choice of initial state
Example: Ventilation Machine

- Short email from Lui Sha
  - many assumptions need clarification

- Problem:
  - ventilation machine must stop when Xray taken
  - but not too often or too long
  - components may break down...
  - ... but patient should survive!

- Nondeterministic message delay
- “all clocks synchronized within 10 msec” (?)

Formal Model

- Modulo my current understanding ...
- Large time scale (600 000) and nondeterministic delays
  - new efficient communication model (nondet message delay + maximal time sampling)
- OO solution
  - slight overkill
  - intuitive
  - specification techniques useful
  - does not add more states
- Failure not yet added!

Controller

```scala
class Controller | clock : Time, lastPauseTime : Time .

r1 [readPushButtonMsg] :
  pushButton
  < C : Controller | clock : T, lastPauseTime : T' >
  =>
  if not tooEarly(T, T') then --- take X-ray
    < C : Controller | lastPauseTime : T + 3000 >
    dly(to VentMachine pause 2sec in 1sec, MIN-DELAY, 50)
    dly(to X-ray takeXray in 2sec, MIN-DELAY, 50)
  else --- too early!
    < C : Controller | >
    dly(userWaitUntil (T' + 600000 + 10), 0, 50)
  fi .
```
**Controller: Timed Behavior**

\[ \text{eq} \ \delta(< C : \text{Controller} | \text{clock} : T >, T') = < C : \text{Controller} | \text{clock} : T + T' >. \]

\[ \text{eq} \ \text{mte(< C : \text{Controller} | \text{clock} : T >)} = \text{INF}. \]

**Ventilation Machine Class**

\[
\text{class VentMachine} \ | \ \text{state} : \text{BreathingState}.
\]

\[
\text{sort} \ \text{BreathingState}.
\]

\[
\text{op} \ \text{breathing} : \rightarrow \text{BreathingState} \ [\text{ctor}].
\]

\[
\text{ops} \ \text{breatheUntil} \ \text{stopBreathing} : \ \\
\text{Time} \rightarrow \text{BreathingState} \ [\text{ctor}].
\]

The latter two are "timers"

**Ventilation Machine Rules**

Stop request received after delay:

- should pause for two seconds after one second:

  --- Read message from Controller:

  \[
  \text{rl [VMreadPause]} : \ \\
  \text{(to VentMachine pause 2sec in 1sec)} \ \\
  < V : \text{VentMachine} | > \ \\
  \rightarrow \ \\
  < V : \text{VentMachine} | \ \text{state} : \text{breatheUntil(1000)} >.
  \]

  --- Wait timer expired:

  \[
  \text{rl [stopBreathing]} : \ \\
  < V : \text{VentMachine} | \ \text{state} : \text{breatheUntil(0)} > \ \\
  \rightarrow \ \\
  < V : \text{VentMachine} | \ \text{state} : \text{stopBreathing(2000)} >.
  \]

  --- start to breathe again:

  \[
  \text{rl [restartBreathing]} : \ \\
  < V : \text{VentMachine} | \ \text{state} : \text{stopBreathing(0)} > \ \\
  \rightarrow \ \\
  < V : \text{VentMachine} | \ \text{state} : \text{breathing} >.
  \]
VM Time Behavior

- "Timers" decreased with elapse of time
- Time advance must stop when a timer can reach 0

\[
\begin{align*}
\text{eq } & \delta(< V : \text{VentMachine} | \text{state} : \text{breathing} >, T) = \\
& < V : \text{VentMachine} | \text{state} : \text{breathing} > . \\
\text{eq } & \text{mte}(< V : \text{VentMachine} | \text{state} : \text{breathing} >) = \text{INF} . \\
\text{eq } & \delta(< V : \text{VentMachine} | \text{state} : \text{breatheUntil}(T) >, T') = \\
& < V : \text{VentMachine} | \text{state} : \text{breatheUntil}(T \monus T') > . \\
\text{eq } & \text{mte}(< V : \text{VentMachine} | \text{state} : \text{breatheUntil}(T) >) = T . \\
\text{eq } & \delta(< V : \text{VentMachine} | \text{state} : \text{stopBreathing}(T) >, T') = \\
& < V : \text{VentMachine} | \text{state} : \text{stopBreathing}(T \monus T') > . \\
\text{eq } & \text{mte}(< V : \text{VentMachine} | \text{state} : \text{stopBreathing}(T) >) = T .
\end{align*}
\]

X-ray Class

\[
\begin{align*}
\text{class } X-ray & | \text{state} : \text{XRstate} . \\
\text{sort } & \text{XRstate} . \\
\text{ops } & \text{idle \ takingXray} : \rightarrow \text{XRstate \ [ctor]} . \\
\text{op } & \text{wait} : \text{Time} \rightarrow \text{XRstate \ [ctor]} .
\end{align*}
\]

Rules for X-ray Machine

Get message; wait for two seconds; then take (instantaneous?) X-ray:

--- Getting message:
rl [XrayReadMsg] :
\[
\begin{align*}
&(\text{to X-ray takeXray in 2sec}) \\
& < X : X-ray | > \\
& \rightarrow \\
& < X : X-ray | \text{state} : \text{wait}(2000) > .
\end{align*}
\]

--- This request overrides previous requests ... correct?

--- Take X-ray:
rl [takeXray] :
\[
\begin{align*}
& < X : X-ray | \text{state} : \text{wait}(0) > \\
& \rightarrow \\
& < X : X-ray | \text{state} : \text{takingXray} > .
\end{align*}
\]

--- Afterwards, the system should immediately idle:
rl [idle] :
\[
\begin{align*}
& < X : X-ray | \text{state} : \text{takingXray} > \\
& \rightarrow \\
& < X : X-ray | \text{state} : \text{idle} > .
\end{align*}
\]

Behavior in time is trivial
For execution environment, add a user

- push button every X time units (e.g., every minute)
- read wait-messages:

```
class User | pushButtonTimer : TimeInf, pushInterval : Time.
```

```
rl [pushButton] :
  < U : User | pushButtonTimer : 0, pushInterval : T > =>
  < U : User | pushButtonTimer : T >
dly(pusherButton, MIN-DELAY, 50).
```

```
rl [readUserWait] :
  (userWaitUntil T) < U : User | > => < U : User | >
```

### Initial State and Time Sampling Strategy

Initial state:
```plaintext
op initState : -> GlobalSystem.
eq initState =
  {< u : User | pushButtonTimer : 0, pushInterval : 60000 >
  < ct : Controller | clock : 0, lastPauseTime : 0 >
  < vm : VentMachine | state : breathing >
```

For execution: maximal time sampling:

```plaintext
(set tick max def 1000.)
```

### Analysis I

**Safety:** machine not breathing when X-ray taken!

- search for state violating property:

```plaintext
{tsearch [1] initState =>* {C:Configuration
  < xr : X-ray | state : takingXray >
  < vm : VentMachine | state : breathing >}
in time <= 70000.)
```

```plaintext
{tsearch [1] initState =>* {C:Configuration
  < xr : X-ray | state : takingXray >
  < vm : VentMachine | state : breathing >}
in time <= 70000.)
```

No such state found (in 2 resp 3 seconds!)

### Analysis II

**Usability:** After first push of button (at time 0), an X-ray must be taken within 3 seconds:

```plaintext
(find latest initState =>* {C:Configuration
  < xr : X-ray | state : takingXray >
  < vm : VentMachine | state : breathing >}
  with no time limit.)
```

Result: {...} in time 2100

An X-ray will always be taken within time 2100.
Analysis III

Safety:
- ventilation not paused for longer than 2 seconds
- not more than one pause within 10 minutes

Not yet done!
- Not expressible in non-metric TL in our spec
- Must store history in VM
  - add extra “mythical attributes” pauseTime and timeSinceLastPause
    and update them according to “correct” time elapsed
  - then, analysis trivial: search for bad breaches