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 Overview I

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

- 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
Time-Sensitive Cryptographic Protocols

- 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
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:
  – 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 \( \{ \text{clock} (r) \} \)
- dense time domain
- clock can stop at any time due to battery failure
- retrograde clock: \( \text{clock}(24) \) should be reset to \( \text{clock}(0) \)
Example: Retrograde Clock

(tmod DENSE-CLOCK is pr POSRAT-TIME-DOMAIN .
ops clock stopped-clock : Time -> System .
vars R R' : Time .
crl [tickWhenRunning] :
  \{clock(R)\} => \{clock(R + R')\} in time R'
  if R' <= 24 - R [nonexec] .
rl [tickWhenStopped] :
  \{stopped-clock(R)\} => \{stopped-clock(R)\}
  in time R' [nonexec] .
rl [reset] :  clock(24) => clock(0) .
rl [batteryDies] :
  clock(R) => 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 ∆
  – 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
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
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 \{\text{clock}(25)\} be reached from \{\text{clock}(0)\} within time 100?

\[
\text{(tsearch [1] \{\text{clock}(0)\} =>* \{\text{clock}(25)\} in time <= 100.)}
\]

• Finite reachable state space: unbounded search:

\[
\text{(utsearch [1] \{\text{clock}(0)\} =>* \{\text{clock}(25)\}.)}
\]

• State \{\text{clock}(1/2)\} not found because of time sampling:

\[
\text{(utsearch [1] \{\text{clock}(0)\} =>* \{\text{clock}(1/2)\}.)}
\]

• Deadlock possible?

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

- Propositional linear temporal logic
- **Not metric** temporal logic
  - propositions may involve elapsed time ("clocked properties")
- Time bound $\Rightarrow$ termination of model checking
- Provides counterexamples
- Uses *Maude*’s high-performance model checker
- Formula:
  - user-defined atomic propositions
  - temporal logic operators such as $\sim$, $\lor$, $[$, $\lhd$, $\triangleright$, $\bigcup$
Example: Defining Propositions

- **clock-dead**: holds if clock has stopped
  
  \[
  \text{op clock-dead : } \rightarrow \text{Prop [ctor]} .
  \]
  
  \[
  \text{vars } R \ R' : \text{Time} .
  \]
  
  \[
  \text{eq } \{\text{stopped-clock}(R)\} \mid= \text{clock-dead} = \text{true} .
  \]

- **clock-is(r)**: **running** clock shows \(r\)
  
  \[
  \text{op clock-is : } \text{Time} \rightarrow \text{Prop [ctor]} .
  \]
  
  \[
  \text{eq } \{\text{clock}(R)\} \mid= \text{clock-is}(R') = (R == R') .
  \]

- **clockEqualsTime**: **running** clock shows the current time elapse
  
  \[
  \text{op clockEqualsTime : } \rightarrow \text{Prop [ctor]} .
  \]
  
  \[
  \text{eq } \{\text{clock}(R)\} \text{ in time } R' \mid= \text{clockEqualsTime} = (R == R') .
  \]
Example: Temporal Logic Model Checking

- The clock is never 25:

  \[(mc \{\text{clock}(0)\} |\!\!u [] \sim \text{clock-is}(25).)\]

- Clock shows elapsed time until time 24 or until it dies:

  \[(mc \{\text{clock}(0)\} |\!\!t \text{clockEqualsTime} U
  
  \quad (\text{clock-is}(24) \lor \text{clock-dead})
  
  \quad \text{in time} \leq 100.)\]

- Counter-example: clock value 4 not always reached:

  \[(mc \{\text{clock}(0)\} |\!\!u <> \text{clock-is}(4).)\]
Classes and Objects

Class declaration:

\[ \text{class } C \mid att_1 : s_1, \ldots, att_n : s_n. \]

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

\[ < O : C \mid att_1 : val_1, \ldots, att_n : val_n > \]

Example:

\[ \text{class Thermostat} \mid \text{currTemp} : \text{PosRat}, \text{heater} : \text{OnOff}, \]
\[ \text{desiredTemp} : \text{Nat}. \]

An object might be

\[ < "Kitchen" : \text{ThermoStat} \mid \text{currTemp} : 74, \text{heater} : \text{on}, \]
\[ \text{desiredTemp} : 72 > \]
An instantaneous rewrite rule could be

\[
\text{var } O : \text{Oid} . \quad \text{vars } M N : \text{Nat} .
\]

\[
\text{crl } [\text{turnOffHeater}] :
\]

\[
< O : \text{ThermoStat} \mid \text{currTemp} : M, \text{heater} : \text{on}, \text{desiredTemp} : N >
\]

\[
\Rightarrow
\]

\[
< O : \text{ThermoStat} \mid \text{heater} : \text{off} >
\]

\[
\text{if } M \geq N + 6 .
\]
A message can be seen as a term of sort \( \text{Msg} \):

\[
\text{msg \ setDesTemp} : \text{Oid Nat} \rightarrow \text{Msg} .
\]

A concrete message is then

\[
\text{setDesTemp("LivingRoom", 66)}
\]

For transmission delays:

\[
\text{dly(setDesTemp("LivingRoom", 66), 5)}
\]

will be “ready” in time 5
The whole state: a multiset of objects and (delayed and ripe) messages:

\{
\langle \text{"Kitchen"} : \text{ThermoStat} \mid \text{currTemp} : 74, \text{heater} : \text{on}, \text{desiredTemp} : 72 \rangle
\langle \text{"BedRoom"} : \text{ThermoStat} \mid \text{currTemp} : 68, \text{heater} : \text{off}, \text{desiredTemp} : 66 \rangle
\langle \text{"Bathroom"} : \text{ThermoStat} \mid \text{currTemp} : 73, \text{heater} : \text{off}, \text{desiredTemp} : 78 \rangle
\text{dly}(\text{setDesTemp("BedRoom", 64), 2})
\text{setDesTemp("Bathroom", 80)}\}
Rules

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

Rule:

\texttt{rl [readNewDesTemp] : setDesTemp(O, N) \langle O : \text{ThermoStat} \mid \rangle \Rightarrow \langle O : \text{ThermoStat} \mid \text{desiredTemp} : N \rangle .}

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

```
var C : Configuration . var T : Time .

crl {C} => {delta(C, T)} in time T if T <= mte(C) .
```

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

```
eq delta(< O : ThermoStat | currTemp : N ,
           heater : on >, T) =
```
\textbf{eq \texttt{delta}}( < 0 : \text{ThermoStat} \mid \text{currTemp} : N, \\
heater : \text{off} >, T) = \\
< 0 : \text{ThermoStat} \mid \text{currTemp} : N - T > .

\textbf{eq \texttt{mte}}( < 0 : \text{ThermoStat} \mid \text{currTemp} : M, \text{heater} : \text{on}, \\
\text{desiredTemp} : N >) = \\
((N + 6) \text{ monus } M) / 2 .

\textbf{eq \texttt{mte}}( < 0 : \text{ThermoStat} \mid \text{currTemp} : M, \text{heater} : \text{off}, \\
\text{desiredTemp} : N >) = \\
M \text{ monus } (N - 6) .
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**:

  ```
  class Server |
  maxBudget : NzTime,
  period : NzTime,
  state : ServerState,
  usedOfBudget : Time,
  timeToDeadline : Time,
  timeExecuted : Time .
  
  sort ServerState .
  ops idle waiting executing : -> ServerState [ctor] .
  ```

- Dynamic behavior modeled by 10 rewrite rules
Dynamic behavior: job arrives at server $S_i$ while server $S_j$ is executing:

\[
\text{rl } \text{[idleToActive]} : \\
\langle S_i : \text{Server} \mid \text{period} : T_i, \text{state} : \text{idle}, \\
\text{timeToDeadline} : T \rangle \\
\langle S_j : \text{Server} \mid \text{state} : \text{executing}, \text{timeToDeadline} : T' \rangle \\
\Rightarrow \\
\begin{cases} 
\text{if } (T + T_i) < T' \text{ then } \text{ --- preempt } S_j \\
\langle S_i : \text{Server} \mid \text{state} : \text{executing}, \text{timeExecuted} : 0, \\
\text{timeToDeadline} : T + T_i \rangle \\
\langle S_j : \text{Server} \mid \text{state} : \text{waiting} \rangle \end{cases} \\
\text{else } \text{ --- wait for processor} \\
\langle S_i : \text{Server} \mid \text{state} : \text{waiting}, \text{timeExecuted} : 0, \\
\text{timeToDeadline} : T + T_i \rangle \\
\langle S_j : \text{Server} \mid \rangle \\
\text{fi}.
\]
Remaining budget larger than relative deadline:

\[
\text{op DEADLINE-MISS} : \rightarrow \text{Configuration [ctor].}
\]

\[
crl \text{ [deadlineMiss]} :
\begin{array}{c}
< S_i : \text{Server} \mid \text{state} : \text{STATE}, \text{usedOfBudget} : \text{T}, \\
\text{timeToDeadline} : \text{T'}, \\
\text{maxBudget} : Q_i > \\
\end{array}
\]

\[
=\rightarrow
\text{DEADLINE-MISS}
\]

\[
\text{if } (Q_i - T) > T'
\]

\[
\text{\textbackslash } \text{ STATE == waiting or STATE == executing .}
\]
Analyzing CASH in Real-Time Maude (I)

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

\begin{verbatim}
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}
\end{verbatim}
Time-bounded reachability analysis

- search for state containing \texttt{DEADLINE-MISS}:

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

Solution 1

\begin{verbatim}
C:Configuration <- ... ; TIME_ELAPSED:Time <- 12
\end{verbatim}

- hard tasks \texttt{cannot} be guaranteed!

- search took 140 sec.

- no ingenuity in choice of initial state
Example: Ventilation Machine
Ventilation Machine Example

- 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

class Controller | clock : Time, lastPauseTime : Time.

rl [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

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

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

class VentMachine | state : BreathingState .

sort BreathingState .
op breathing : -> BreathingState [ctor] .
ops breatheUntil stopBreathing :
    Time -> BreathingState [ctor] .

The latter two are “timers”
Stop request received after delay:

- should **pause for two seconds** after one second:

  --- Read message from Controller:
  \[rl \text{[VMreadPause]} : \]
  (to VentMachine pause 2sec in 1sec)
  \[
  < V : \text{VentMachine} | >
  
  
  =>
  
  < V : \text{VentMachine} | state : breatheUntil(1000) > .

  --- Wait timer expired:
  \[rl \text{[stopBreathing]} : \]
  \[
  < V : \text{VentMachine} | state : breatheUntil(0) >
  
  
  =>
  

--- start to breathe again:

rl [restartBreathing] :
    < V : VentMachine | state : stopBreathing(0) >
=>
    < V : VentMachine | state : breathing > .
VM Time Behavior

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

\[
\text{eq } \delta(< V : \text{VentMachine} | \text{state} : \text{breathing} >, T) = < V : \text{VentMachine} | \text{state} : \text{breathing} >.
\]

\[
\text{eq } \text{mte}(< V : \text{VentMachine} | \text{state} : \text{breathing} >) = \text{INF}.
\]

\[
\text{eq } \delta(< V : \text{VentMachine} | \text{state} : \text{breatheUntil}(T) >, T') = < V : \text{VentMachine} | \text{state} : \text{breatheUntil}(T \text{ minus } T') >.
\]

\[
\text{eq } \text{mte}(< V : \text{VentMachine} | \text{state} : \text{breatheUntil}(T) >) = T.
\]

\[
\text{eq } \delta(< V : \text{VentMachine} | \text{state} : \text{stopBreathing}(T) >, T') = < V : \text{VentMachine} | \text{state} : \text{stopBreathing}(T \text{ minus } T') >.
\]

\[
\text{eq } \text{mte}(< V : \text{VentMachine} | \text{state} : \text{stopBreathing}(T) >) = T.
\]
class X-ray | state : XRstate .

sort XRstate .
ops idle takingXray : -> XRstate [ctor] .
op wait : Time -> XRstate [ctor] .
Rules for X-ray Machine

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

--- Getting message:

```haskell
rl [XrayReadMsg] :
    (to X-ray takeXray in 2sec)
    < X : X-ray | >
=>
--- This request overrides previous requests ... correct?
```

--- Take X-ray:

```haskell
rl [takeXray] :
    < X : X-ray | state : wait(0) >
=>
    < X : X-ray | state : takingXray > .
```
--- Afterwards, the system should immediately idle:

rl [idle] :
  < X : X-ray | state : takingXray >
=>
  < X : X-ray | state : idle > .

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

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

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

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

rl [readUserWait] :  
  (userWaitUntil T)  < U : User | >  =>  < U : User | >  
```
Initial State and Time Sampling Strategy

Initial state:

\[
\text{op initState} : \rightarrow \text{GlobalSystem}.
\]
\[
\text{eq initState} =
\]
\[
\{\langle u : \text{User} \mid \text{pushButtonTimer} : 0, \text{pushInterval} : 60000 > \}
\]
\[
\{\langle ct : \text{Controller} \mid \text{clock} : 0, \text{lastPauseTime} : 0 > \}
\]
\[
\{\langle vm : \text{VentMachine} \mid \text{state} : \text{breathing} > \}
\]
\[
\{\langle xr : \text{X-ray} \mid \text{state} : \text{idle} > \}
\].

For execution: maximal time sampling:

\[
\text{(set tick max def 1000 }).
\]
Safety: machine not breathing when X-ray taken!

- search for state violating property:

\[
(t\text{search} \ [1] \ \text{initState} \Rightarrow^* \\
\{C:\text{Configuration} \\
\quad < \ x_r : \text{X-ray} \ | \ \text{state} : \text{takingXray} > \\
\quad < \ v_m : \text{VentMachine} \ | \ \text{state} : \text{breathing} >\} \\
in \text{time} \leq 70000 .)
\]

\[
(t\text{search} \ [1] \ \text{initState} \Rightarrow^* \\
\{C:\text{Configuration} \\
\quad < \ x_r : \text{X-ray} \ | \ \text{state} : \text{takingXray} > \\
\quad < \ v_m : \text{VentMachine} \ | \ \text{state} : \text{breatheUntil(NZT:NzTime)} >\} \\
in \text{time} \leq 700000 .)
\]

No such state found (in 2 resp 3 seconds!)
Usability: After first push of button (at time 0), an X-ray must be taken within 3 seconds:

\[
\text{(find latest initState =>*)}
\]

\[
\{C:\text{Configuration}
\quad \langle \text{xr : X-ray} \mid \text{state : takingXray} \rangle
\}
\]

\[
\text{with no time limit .}
\]

\[
\text{Result: \{...\} in time 2100}
\]

An X-ray will \textit{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