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
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
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:
  – algebraic equational specification for data types/functional aspects
  – rewrite rules define instantaneous transitions
  – time elapse modeled by tick rewrite rules

\[
\{t\} \Rightarrow \{t'\} \text{ in time } \tau \text{ if } cond
\]

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

\( \text{tm} \text{od DENSE-CLOCK is pr POSRAT-TIME-DOMAIN .} \)
\( \text{ops clock stopped-clock : Time } \rightarrow \text{ System .} \)
\( \text{vars R R'} : \text{ Time .} \)
\( \text{crl [tickWhenRunning] :} \)
\( \{ \text{clock(R)} \} \Rightarrow \{ \text{clock(R + R')} \} \text{ in time R'} \)
\( \text{if R' } \leq 24 - R \text{ [nonexec]} . \)
\( \text{rl [tickWhenStopped] :} \)
\( \{ \text{stopped-clock(R)} \} \Rightarrow \{ \text{stopped-clock(R)} \} \)
\( \text{in time R'} \text{ [nonexec]} . \)
\( \text{rl [reset] :} \text{ clock}(24) \Rightarrow \text{ clock}(0) . \)
\( \text{rl [batteryDies] :} \)
\( \text{clock}(R) \Rightarrow \text{ stopped-clock}(R) . \)
\( \text{endtm} \)
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
Timed rewriting simulates one possible behavior:

Maude> (trew \{\text{clock}(0)\} \text{ in time } \leq 100 .)

Result ClockedSystem :
\{\text{stopped-clock}(24)\} \text{ 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 \( \{ \text{clock}(25) \} \) be reached from \( \{ \text{clock}(0) \} \) within time 100?

\[
\text{(tsearch } [1] \{ \text{clock}(0) \} \Rightarrow \ast \{ \text{clock}(25) \} \\
\text{ in time } \leq 100 .)
\]

• Finite reachable state space: unbounded search:

\[
\text{(utsearch } [1] \{ \text{clock}(0) \} \Rightarrow \ast \{ \text{clock}(25) \} .)
\]

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

\[
\text{(utsearch } [1] \{ \text{clock}(0) \} \Rightarrow \ast \{ \text{clock}(1/2) \} .)
\]

• Deadlock possible?

\[
\text{(utsearch } [1] \{ \text{clock}(0) \} \Rightarrow ! \{ \text{S:System} \} .)
\]
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 $\sim$, $\land$, $[\ ]$, $<>$, $\cup$
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 \ \Box \sim \text{clock-is}(25) \).\)

- Clock shows elapsed time until time 24 or until it dies:
  \[(mc \{\text{clock}(0)\} |=t \ \text{clockEqualsTime} \ U \ (\text{clock-is}(24) \lor \text{clock-dead}) \text{ in time } \leq 100 .\)

- Counter-example: clock value 4 not always reached:
  \[(mc \{\text{clock}(0)\} |=u \ \Diamond \sim \text{clock-is}(4) .\)

Sept 25–29, 2006 UIUC’06
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:

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

An object might be

\[
< "Kitchen" : Thermostat \mid \text{currTemp : 74, heater : on, desiredTemp : 72} >
\]
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 \texttt{Msg}:

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

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:

```
{< "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)}
```
Rules

- 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

```ml
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:

```ml
```
eq delta(< O : ThermoStat | currTemp : N,
heater : off >, T) =

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

eq mte(< O : ThermoStat | currTemp : M, heater : off,
desiredTemp : N >) =
M 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`:

```plaintext
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
Modeling CASH in Real-Time Maude (III)

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

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

```plaintext
op DEADLINE-MISS : -> Configuration [ctor] .

crl [deadlineMiss] :
  < S_i : Server | state : STATE, usedOfBudget : T,
  timeToDeadline : T',
  maxBudget : Q_i >

=>
   DEADLINE-MISS

if (Q_i - T) > T'

   \( \land \) STATE == waiting or STATE == executing .
```
Analyzing CASH in Real-Time Maude (I)

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

\[
\begin{align*}
\text{op init2} & : \rightarrow \text{GlobalSystem} . \\
\text{eq init2} = \\
 & \{< s_1 : \text{Server} | \text{maxBudget} : 2, \text{period} : 5, \\
 & \quad \text{timeExecuted} : 0, \text{usedOfBudget} : 0, \\
 & \quad \text{state} : \text{idle}, \text{timeToDeadline} : 0 > \\
 & \quad < s_2 : \text{Server} | \text{maxBudget} : 4, \text{period} : 7, \ldots > \\
 & \quad [\text{CASH: emptyQueue}] \\
& \quad \text{AVAILABLE-PROCESSOR} \} .
\end{align*}
\]
Time-bounded reachability analysis

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

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

Solution 1

C:Configuration <- ... ; TIME_ELAPSED:Time <- 12

- hard tasks \texttt{cannot} be guaranteed!

- search took 140 sec.

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

- 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”

- local clocks may drift
Formal Model

- Large time scale (600,000 ms) 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

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

\[
\text{eq \ mte}(\langle C : \text{Controller} \mid \text{clock} : T \rangle) = \text{INF}.
\]
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:
  \[
  \text{rl [VMreadPause]} : \\
  \quad (\text{to VentMachine pause 2sec in 1sec}) \\
  \quad < V : \text{VentMachine} \ | \ > \\
  \quad => \\
  \quad < V : \text{VentMachine} \ | \ \text{state : breatheUntil(1000)} > .
  \]

  --- Wait timer expired:
  \[
  \text{rl [stopBreathing]} : \\
  \quad < V : \text{VentMachine} \ | \ \text{state : breatheUntil(0)} > \\
  \quad => \\
  \quad < V : \text{VentMachine} \ | \ \text{state : stopBreathing(2000)} > .
  \]
--- 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

\[
\text{eq } \delta(< V : \text{VentMachine} | \text{state} : \text{breathing} >, T) = < V : \text{VentMachine} | \text{state} : \text{breathing} >.
\]
\[
\text{eq } \text{mte}(< V : \text{VentMachine} | \text{state} : \text{breathing} >) = \text{INF}.
\]
\[
\text{eq } \delta(< V : \text{VentMachine} | \text{state} : \text{breatheUntil}(T) >, T') = < V : \text{VentMachine} | \text{state} : \text{breatheUntil}(T \text{ monus } T') >.
\]
\[
\text{eq } \text{mte}(< V : \text{VentMachine} | \text{state} : \text{breatheUntil}(T) >) = T.
\]
\[
\text{eq } \delta(< V : \text{VentMachine} | \text{state} : \text{stopBreathing}(T) >, T') = < V : \text{VentMachine} | \text{state} : \text{stopBreathing}(T \text{ monus } T') >.
\]
\[
\text{eq } \text{mte}(< V : \text{VentMachine} | \text{state} : \text{stopBreathing}(T) >) = T.
\]
VM with Drift

If the drift of VM's clock is $\text{DRIFT}$ per time unit, $\text{mte}$ and $\text{delta}$ must be redefined:

\[
\text{eq } \text{delta}(< V : \text{VentMachine} \mid \text{state} : \text{breatheUntil}(T) >, T') = < V : \text{VentMachine} \mid \text{state} : \\
\quad \text{breatheUntil}(T \text{ minus } (T' + \text{DRIFT} \times T')) > .
\]

\[
\text{eq } \text{mte}(< V : \text{VentMachine} \mid \\
\quad \text{state} : \text{breatheUntil}(T) >) = T / (1 + \text{DRIFT}) .
\]
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:
rl [XrayReadMsg] :
  (to X-ray takeXray in 2sec)
  < X : X-ray | >
=>

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

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

\[
\text{rl [idle]} : \\
\langle X : \text{X-ray} \mid \text{state} : \text{takingXray} \rangle \\
\Rightarrow \\
\langle X : \text{X-ray} \mid \text{state} : \text{idle} \rangle .
\]

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.
```

```plaintext
rl [pushButton] :
< U : User | pushButtonTimer : 0, pushInterval : T >
=>
< U : User | pushButtonTimer : T >
dly(pushButton, MIN_DELAY, 50).
```

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

Initial state:

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

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 *)
\[
\{C:\text{Configuration}
\quad < x_r : \text{X-ray} \mid \text{state} : \text{takingXray} >
\quad < v_m : \text{VentMachine} \mid \text{state} : \text{breathing} >\}
\quad \text{in time} \leq 70000 .)\]

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

No such state found (in 2 resp 3 seconds!)
Analysis (Without Drift) II

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

\[
\text{(find latest initState =>}\text{*}}
\]
\[
\text{\{C:Configuration}
\]
\[
\text{< xr : X-ray \mid state : takingXray >}}
\]
\[
\text{with no time limit .}
\]

Result: \{...\} in time 2100

An X-ray will always be taken within time 2100.
Safety:

- ventilation not paused for longer than 2 seconds (with drift?)
- not more than one pause within 10 minutes
- 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
  - should not impact protocol behavior
  - then, analysis trivial: search for bad states
Extending the VM

Class extended with “observer attributes”:

class VentMachine | state : BreathingState, 
pauseTime : Time, 
timeSinceLastPause : TimeInf .
Updating the Extra Attributes

rl [stopBreathing] :
  < V : VentMachine | state : breatheUntil(0) >
=>
  < V : VentMachine | state : stopBreathing(2000),
  pauseTime : 0 > .

rl [restartBreathing] :
  < V : VentMachine | state : stopBreathing(0) >
=>
  < V : VentMachine | state : breathing,
  pauseTime : 0,
  timeSinceLastPause : 0 > .

Extra attributes not in LHS: no change of behavior
eq delta(< V : VentMachine | state : breathing,
        timeSinceLastPause : TI >, T) =
        < V : VentMachine | state : breathing,
        timeSinceLastPause : TI + T >.

--- same for state breatheUntil

eq delta(< V : VentMachine | state : stopBreathing(T),
        pauseTime : T'',
        timeSinceLastPause : TI >, T'') =
        < V : VentMachine | state :
        stopBreathing(T monus T'),
        pauseTime : T'' + T',
        timeSinceLastPause : TI + T' >.
“The VM cannot be paused for more than 2 seconds each time:"

Search for bad state:

(tsearch [1] initState =>*

{C:Configuration
 < vm : VentMachine | pauseTime : T:Time >}
 such that T:Time > 2000 in time <= 700000 .)

No bad state found
More Analysis (Without Drift)

“VM cannot be paused more than once within 10 minutes:”

(tsearch [1] initState =>*
   {C:Configuration
    < vm : VentMachine | timeSinceLastPause : T:Time,
    state : stopBreathing(T’:Time) >}
   such that T:Time < 600000 in time <= 700000 .)

No bad state found
Model With Drift

Assume that the VM’s internal clock drifts by minus DRIFT (10%)

- rational numbers time domain

- only definition of delta and mte changed

\[
\text{eq } \delta(< V : \text{VentMachine} \mid \\
\text{state : breatheUntil}(T), \\
\text{timeSinceLastPause} : TI >, T') = \\
< V : \text{VentMachine} \mid \\
\text{state : \\
\quad breatheUntil}(T - (T' - (T' \times \text{DRIFT}))), \\
\text{timeSinceLastPause} : TI + T' > .
\]

\[
\text{eq } \text{mte}(< V : \text{VentMachine} \mid \text{state : breatheUntil}(T) >) \\
= T / (1 - \text{DRIFT}) .
\]
\textbf{eq delta}(< V: \text{VentMachine} |
\begin{align*}
&\text{state} : \text{stopBreathing}(T), \\
&\text{pauseTime} : T'', \\
&\text{timeSinceLastPause} : TI >, T') = \\
<V : \text{VentMachine} |
\begin{align*}
&\text{state} : \\
&\text{stopBreathing}(T \text{ monus} \\
&\quad (T' \text{ monus} (T' \times \text{DRIFT}))), \\
&\text{pauseTime} : T'' + T', \\
&\text{timeSinceLastPause} : TI + T' >.
\end{align*}
\end{align*}
\textbf{eq mte}(< V: \text{VentMachine} \mid \text{state} : \text{stopBreathing}(T) >) = \\
T / (1 \text{ monus} \text{DRIFT}) .
“Can VM pause for more than 2 seconds?”

Same search as before:

\[
\text{tsearch} \ [1] \ \text{initState} \Rightarrow *
\]
\[
\{ \text{C:Configuration} \}
\]
\[
\langle \text{vm} : \text{VentMachine} \mid \text{pauseTime} : \text{T:Time} \rangle
\]
\[
\text{such that } \text{T:Time} > 2000 \text{ in time } \leq 700000 .
\]

This time bad state found:

Solution 1

...  

\[
\text{REMAINING\_ATTRIBUTES\_OF\_vm:AttributeSet}
\]
\[
\rightarrow \text{state} : \text{stopBreathing(0), ... ;}
\]
\[
\text{T:Time} \rightarrow 20000/9 ; \text{TIME\_ELAPSED:Time} \rightarrow 10000/3
\]