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

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, . . .
  - probablistic 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

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

$$\{t\} \quad \texttt{=>} \quad \{t'\} \ \texttt{in time} \ \tau \quad \texttt{if} \ cond$$

- 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 can stop at any time due to battery failure
- retrograde clock: `clock(24)` should be reset to `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 Δ
  - 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)}
               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 $\Rightarrow$ termination of model checking
- Provides counterexamples
- Uses Maude's high-performance model checker
- Formula:
  - user-defined atomic propositions
  - temporal logic operators such as ~, \/, [], <>, U

Example: Defining Propositions

- `clock-dead`: holds if clock has stopped

```
op clock-dead : -> Prop [ctor] .
vars R R' : Time .
eq {stopped-clock(R)} |= clock-dead = true .
```

- `clock-is(`r`)`: running clock shows r

```
op clock-is : Time -> Prop [ctor] .
eq {clock(R)} |= clock-is(R') = (R == R') .
```

- `clockEqualsTime`: running clock shows the current time elapse

```
op clockEqualsTime : -> Prop [ctor] .
eq {clock(R)} in time R' |= clockEqualsTime
        = (R == R') .
```

Example: Temporal Logic Model Checking

- The clock is never 25:

```
(mc {clock(0)} |=u [] ~ clock-is(25) .)
```

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

```
(mc {clock(0)} |=t clockEqualsTime U
                  (clock-is(24) \/ clock-dead)
         in time <= 100 .)
```

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

```
(mc {clock(0)} |=u <> clock-is(4) .)
```

Classes and Objects

Class declaration:

$\texttt{class}\ C\ |\ att_1\ :\ s_1,\ \ldots,\ att_n\ :\ s_n\ .$

A object instance O of class C is a term

$\texttt{<}\ O\ :\ C\ |\ att_1\ :\ val_1,\ \ldots,\ att_n\ :\ val_n\ \texttt{>}$

Example:

```
class Thermostat | currTemp : PosRat, heater : OnOff,
                   desiredTemp : Nat .
```

An object might be

```
< "Kitchen" : ThermoStat | currTemp : 74, heater : on,
                           desiredTemp : 72 >
```

More Objects

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 .
```

Messages

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

```
msg setDesTemp : Oid Nat -> Msg .
```

A concrete message is then

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

For transmission delays:

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

will be “ready” in time 5

Configurations

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

OO and Time

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) =
        < O : ThermoStat | currTemp : N + 2 * T > .
```

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

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`:

```
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:

```
rl [idleToActive] :
   < Si : Server | period : Ti, state : idle,
                   timeToDeadline : T >
   < Sj : Server | state : executing, timeToDeadline : T' >
  =>
   if (T + Ti) < T' then    --- preempt Sj
      (< Si : Server | state : executing, timeExecuted : 0,
                       timeToDeadline : T + Ti >
       < Sj : Server | state : waiting >)
   else                     --- wait for processor
      (< Si : Server | state : waiting, timeExecuted : 0,
                       timeToDeadline : T + Ti >
       < Sj : Server | >)
   fi .
```

Modeling CASH in Real-Time Maude (IV)

Remaining budget larger than relative deadline:

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

crl [deadlineMiss] :
    < Sᵢ : Server | state : STATE, usedOfBudget : T,
                    timeToDeadline : T',
                    maxBudget : Qᵢ >
  =>
    DEADLINE-MISS
  if (Qᵢ - T) > T'
     /\ STATE == waiting or STATE == executing .
```

Analyzing CASH in Real-Time Maude (I)

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

```
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} .
```

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

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 comunication 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

```
eq delta(< C : Controller | clock : T >, T') =
     < C : Controller | clock : T + T' > .

eq mte(< C : Controller | clock : T >) = 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”

Ventilation Machine Rules

Stop request received after delay:

- should pause for two seconds after one second:

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

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

```
eq delta(< V : VentMachine | state : breathing >, T) =
         < V : VentMachine | state : breathing > .
eq mte(< V : VentMachine | state : breathing >) = INF .

eq delta(< V : VentMachine | state : breatheUntil(T) >, T') =
   < V : VentMachine | state : breatheUntil(T monus T') > .
eq mte(< V : VentMachine | state : breatheUntil(T) >) = T .

eq delta(< V : VentMachine | state : stopBreathing(T) >, T') =
   < V : VentMachine | state : stopBreathing(T monus T') > .
eq mte(< V : VentMachine | state : stopBreathing(T) >) = T .
```

VM with Drift

If the drift of VM's clock is `DRIFT` per time unit, `mte` and `delta` must be redefined:

```
eq delta(< V : VentMachine | state : breatheUntil(T) >, T') =
   < V : VentMachine | state :
              breatheUntil(T monus  (T' + DRIFT * T')) > .
eq mte(< V : VentMachine |
              state : breatheUntil(T) >) = T / (1 + DRIFT) .
```

X-ray Class

```
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 | >
  =>
   < X : X-ray | state : wait(2000) > .
--- 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:
rl [idle] :
   < X : X-ray | state : takingXray >
  =>
   < X : X-ray | state : idle > .
```

Behavior in time is trivial

User

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(pushButton, MIN-DELAY, 50) .

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

Initial State and Time Sampling Strategy

Initial state:

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

For execution: maximal time sampling:

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

Analysis (Without Drift) I

Safety: machine not breathing when X-ray taken!

- search for state violating property:

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

(tsearch [1] initState =>*
   {C:Configuration
    < xr : X-ray | state : takingXray >
    < vm : VentMachine | state : breatheUntil(NZT:NzTime) >}
  in time <= 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:

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

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

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

Analysis (Without Drift) III

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

Time 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' > .
```

Analysis (Without Drift)

“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

```
eq delta(< V : VentMachine |
                state : breatheUntil(T),
                timeSinceLastPause : TI >, T') =
        < V : VentMachine |
                state :
                  breatheUntil(T monus
                                (T' monus (T' * DRIFT))),
                timeSinceLastPause : TI + T'  > .


eq mte(< V : VentMachine | state : breatheUntil(T) >)
      = T / (1 monus DRIFT) .
```

```
eq delta(< V : VentMachine |
              state : stopBreathing(T),
              pauseTime : T'',
              timeSinceLastPause : TI >, T') =
        < V : VentMachine |
              state :
                 stopBreathing(T monus
                              (T' monus (T' * DRIFT))),
              pauseTime : T'' + T',
              timeSinceLastPause : TI + T' > .

eq mte(< V : VentMachine | state : stopBreathing(T) >) =
             T / (1 monus DRIFT) .
```

Analysis With Drift

"Can VM pause for more than 2 seconds?"

Same search as before:

```
(tsearch [1] initState =>*
     {C:Configuration
      < vm : VentMachine | pauseTime : T:Time >}
         such that T:Time > 2000 in time <= 700000 .)
```

This time bad state found:

```
Solution 1
    ...
    REMAINING_ATTRIBUTES_OF_vm:AttributeSet
        --> state : stopBreathing(0), ... ;
    T:Time --> 20000/9 ; TIME_ELAPSED:Time --> 10000/3
```