

On Real-Time Requirements in Specification-Level UML Models

Risto Pitkänen and Tommi Mikkonen
Tampere University of Technology
Finland
{risto.pitkanen,tommi.mikkonen@tut.fi}

Motivation for high-level modeling

- Abstraction is a powerful thinking tool
- High-level models
 - Better grasp of system as a whole
 - Early validation and verification
- Real-time systems should be no exception
 - Inherent behavioral complexity
- High-level facilities for expressing time-related issues largely missing from UML

Use cases

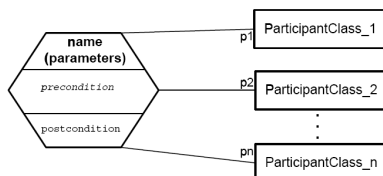
- Requirements/specification level modeling in UML
 - E.g. Unified Process
- Serious handicaps from the point of view of real-time specification:
 - Use case interaction is not supported
 - use case A cannot require that use case B has been executed
 - No explicit time constructs

Formalization of use cases

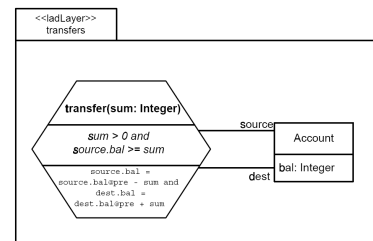
- Catalysis [D'Souza and Wills 1999]
 - Use case = *joint action*
 - Joint actions originate from *DisCo*

```
action assign_mentor(subject: Instructor,
                    watchdog: Instructor)
post subject.mentor = watchdog and
let ex_mentee = watchdog.mentee @pre in
ex_mentee <> null ==> ex_mentee.mentor = null
```

Our approach: joint action



Layered Action Diagrams (LADs)



Properties of LADs

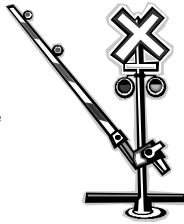
- Simple LADs are superposed onto each other to form a total system
 - Aspect-oriented structuring of a model
- Semi-executable:
 - Generate all joint action instances whose preconditions hold true in present state
 - Pick one, modify participant states such that postcondition becomes true

Extending LADs with real-time issues

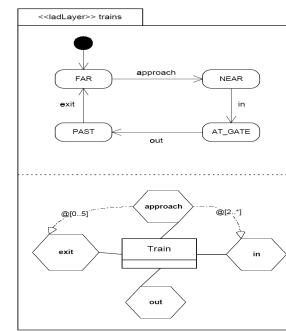
- LADs provide a suitable high-level modeling formalism also for real-time systems
- We want to be able to express real-time dependencies between joint actions
- Aim: easy and natural formalization of requirements such as *"When gas valve is opened, it must be closed within 5 seconds unless the burner is ignited successfully."*

Generalized Railroad Crossing (GRC)

- System controls gate at a railroad crossing
- **Requirements:**
 - **Safety:** Whenever there is at least one train in the crossing, the gate must be down.
 - **Utility:** When there is no train in the crossing, the gate must be raised within a reasonable time.

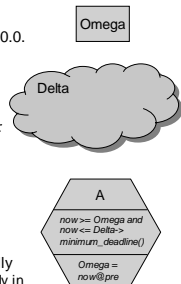


GRC with LADs: Trains

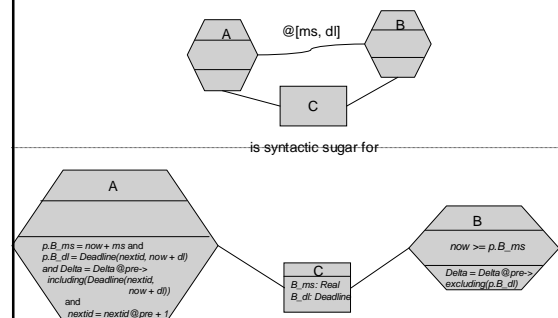


Real-time semantics

- Global clock Ω , typed $Real$, initialized to 0.0.
- Initially empty set Δ of deadlines.
 - Holds pairs $(DeadlineId, Real)$
- In each joint action, an implicit parameter now : $Time$.
- Each joint action has an implicit precondition
 - $now \geq \Omega$ and $now \leq \Delta \rightarrow minimum_deadline()$
- and an implicit postcondition
 - $\Omega = now@pre$
- In effect, this causes time to grow monotonically
 - never exceeding the minimum deadline currently in Δ



Semantics of a real-time constraint



Another way of putting it

Diagram illustrating a distributed system with three nodes (A, B, C) and a central cloud labeled Delta.

Node A is connected to Node B via a link labeled $@[ms, dl]$. Node C is connected to both A and B.

Node A is labeled "records minimum separation and deadline in common participant". Node B is labeled "checks minimum separation".

A curved arrow labeled "adds deadline to" points from Node A to the cloud. A curved arrow labeled "removes deadline (remember that time cannot proceed beyond minimum deadline)" points from the cloud to Node B.

Gate

Controller

The diagram illustrates a **Controller** class and its associated state machines.

Controller Class:

- Attributes:**
 - `<<id_type>> name`
 - `<<id_type>> gate`
 - `<<id_type>> controller`
- Operations:**
 - `<<id_type>> raise`
 - `<<id_type>> lower`

State Machines:

- raise:**
 - Initial state: `c.getState(TO_RAISE)`
 - Final state: `c.getState(WAIT_FIRST)`
- lower:**
 - Initial state: `c.getState(TO_LOWER)`
 - Final state: `c.getState(WAIT_LAST)`
- exit:**
 - Initial state: `0`
 - Final state: `1`
 - Transitions:
 - `0 --> 1` on `raise` and `c.getState(WAIT_LAST) == c.getState(TO_RAISE)`
 - `1 --> 0` on `lower`

Controller Logic:

- raise:**

```

if (c.getState(WAIT_FIRST) == c.getState(TO_LOWER))
    and
    (c.getState(TO_RAISE) == c.getState(WAIT_LAST)
    and
    c.name != c.nameGate + 1)
    goto raise

```
- lower:**

```

if (c.getState(WAIT_LAST) == c.getState(TO_LOWER))
    goto lower

```

Sequence Diagram:

- Starts at a black circle.
- Transitions to `WAIT_FIRST` and `WAIT_LAST` states.
- Transitions from `WAIT_FIRST` to `TO_LOWER` and from `WAIT_LAST` to `TO_RAISE` states.

Controller real-time constraints

`<<cladLayer>> controller`

```

stateDiagram-v2
    state Controller
    state raise
    state lower
    state approach
    state exit

    Controller --> raise : c
    Controller --> lower : c
    Controller --> approach : c
    Controller --> exit : c

    raise --> Controller : c
    lower --> Controller : c
    approach --> Controller : c
    exit --> Controller : c

    Controller --> raise : c@pre.oclnState(TO_RAISE) == @CANCEL
    Controller --> lower : c@pre.oclnState(WAIT_FIRST) == @1
    Controller --> exit : c@pre.n_trains=1 and c.oclnState(WAIT_LAST) == @[0..1]
  
```

The diagram illustrates the real-time constraints for a Controller. The Controller state is the central component, with transitions to four other states: **raise**, **lower**, **approach**, and **exit**. The transitions are labeled with `c` and guarded by conditions:

- Controller to raise:** `c@pre.oclnState(TO_RAISE) == @CANCEL`
- Controller to lower:** `c@pre.oclnState(WAIT_FIRST) == @1`
- Controller to approach:** `c@pre.n_trains=1 and c.oclnState(WAIT_LAST) == @[0..1]`
- Controller to exit:** `c@pre.n_trains=1 and c.oclnState(WAIT_LAST) == @[0..1]`

The transitions from the other states back to the Controller are labeled with `c`.

Validation and Verification

- Either theorem proving (using TLA) or model checking (timed automata) can be used for verification and validation
- An equivalent GRC DisCo model has been model checked using the Kronos tool after mapping it to timed automata
- Theorem proving applies to generic models, model checking currently only to specific instances (e.g. two trains)

Conclusions

- Successful application of earlier RT modeling results and techniques in a UML profile based setting
- New approach for expressing real-time requirements for formalized use cases (*joint actions*)
 - Real-time constraint associations
- Methodological and notational support for separating real-time issues from the underlying control logic