UPPAAL: A Crash Course

Mohammad Mousavi

HG 6.79, Department of Computer Science
Eindhoven University of Technology

OGO 1.3, April 2007

Outline

Outline

Outline

Outline

Outline

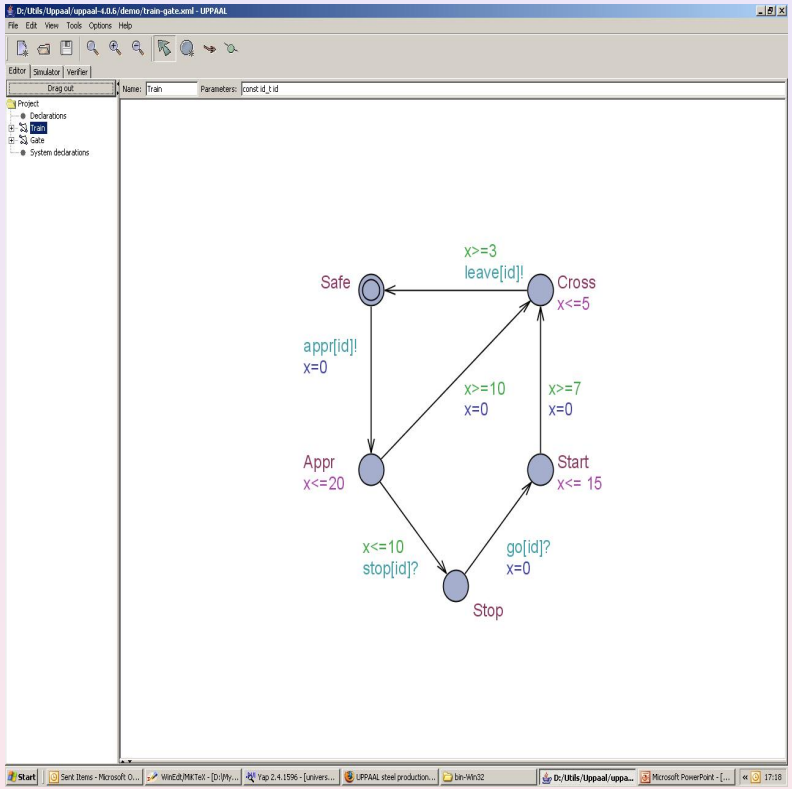
Outline

Outline

What?

A graphical environment for

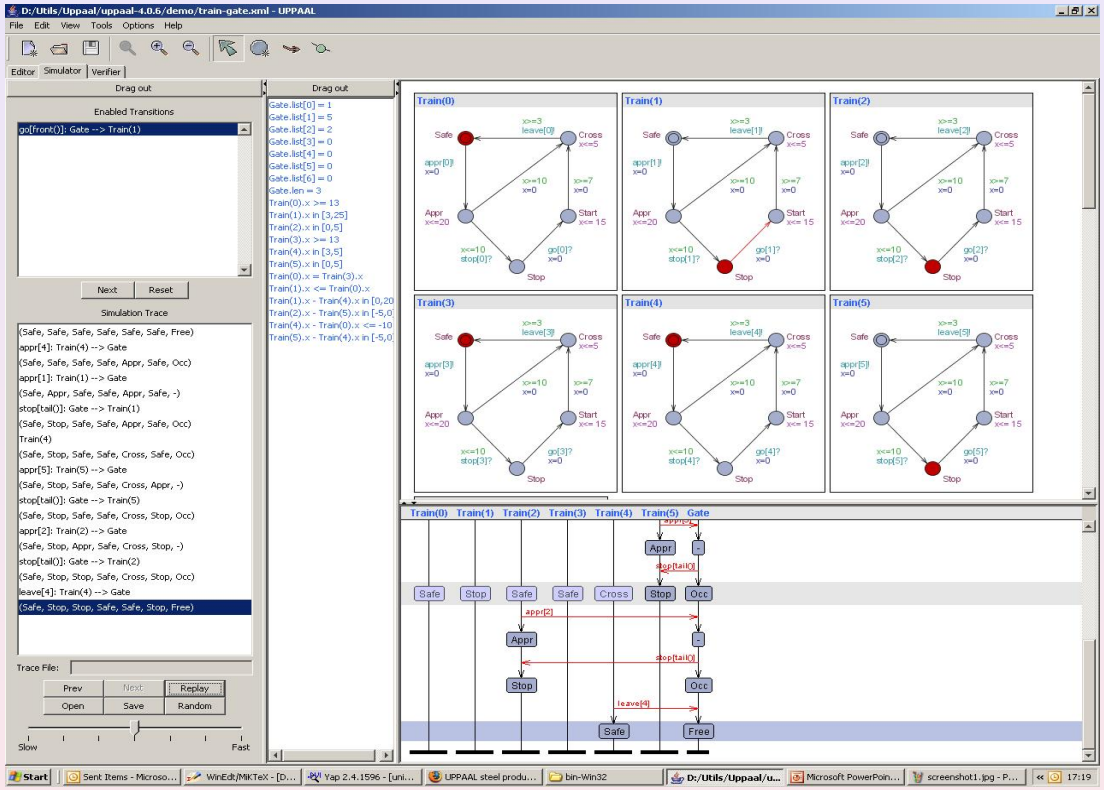
1. specification,



What?

A graphical environment for

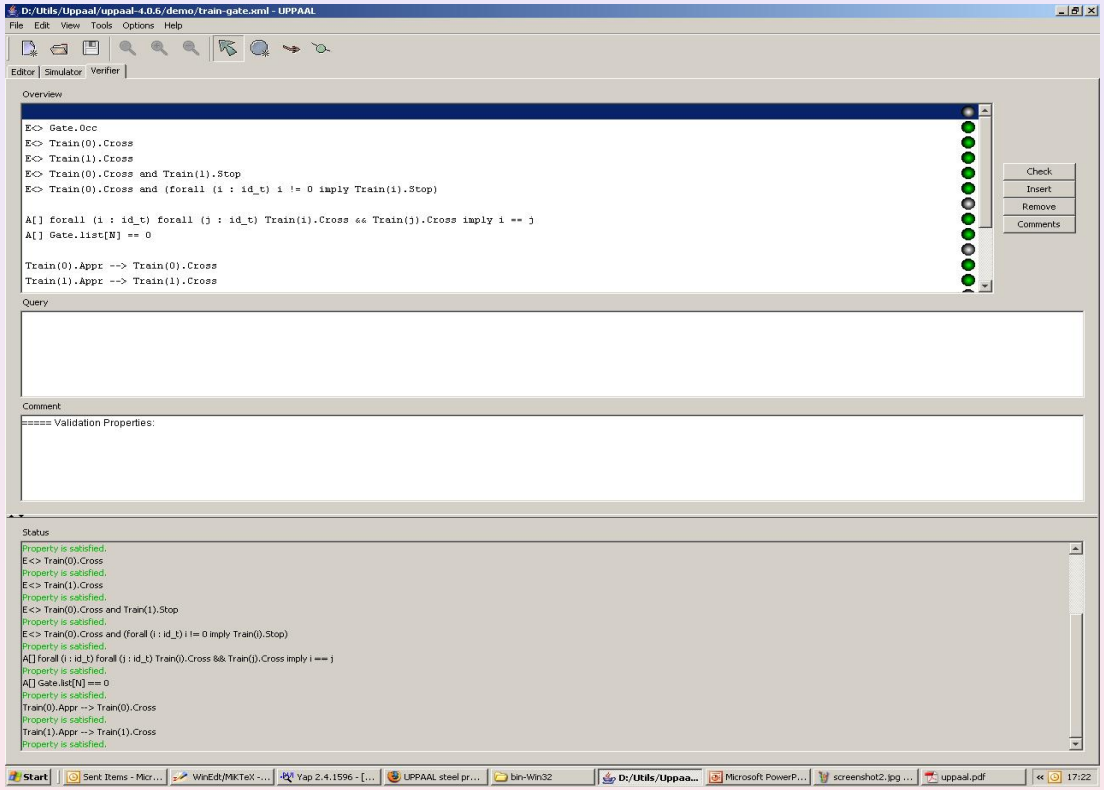
1. specification,
2. simulation,



What?

A graphical environment for

1. specification,
2. simulation, and
3. verification

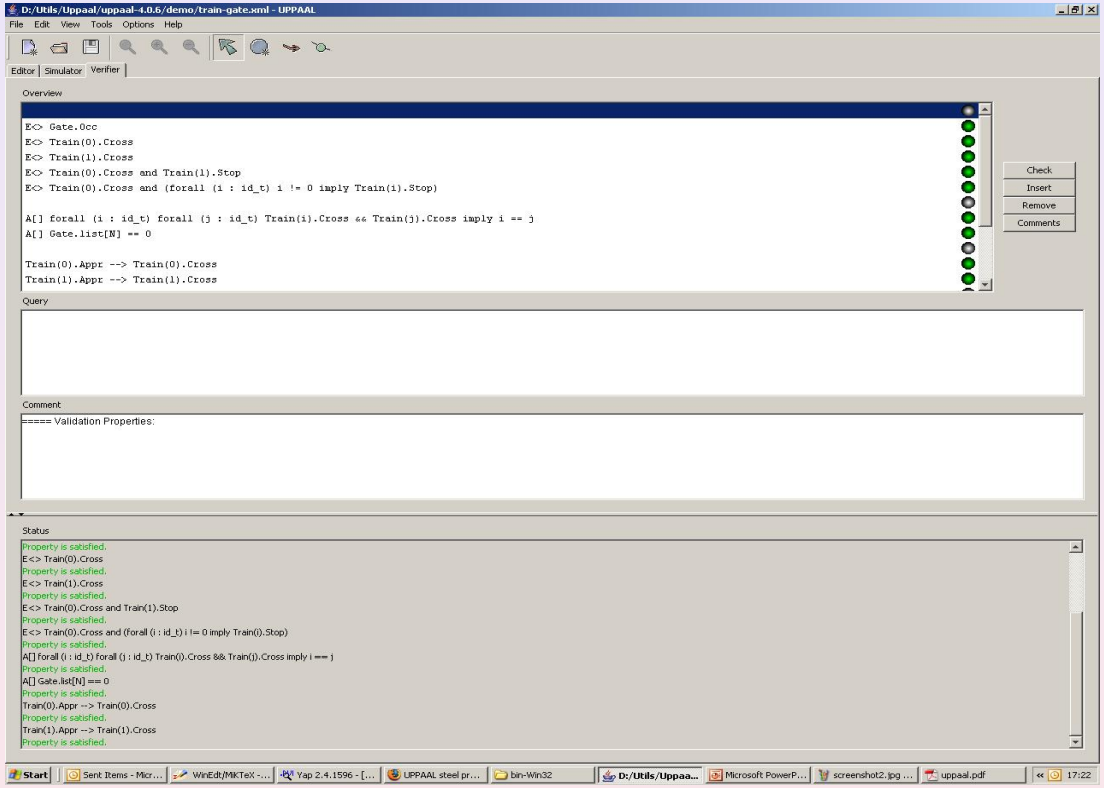


What?

A graphical environment for

1. specification,
2. simulation, and
3. verification
4. planning, scheduling, ... (not related to OGO 1.3)

of real-time systems.



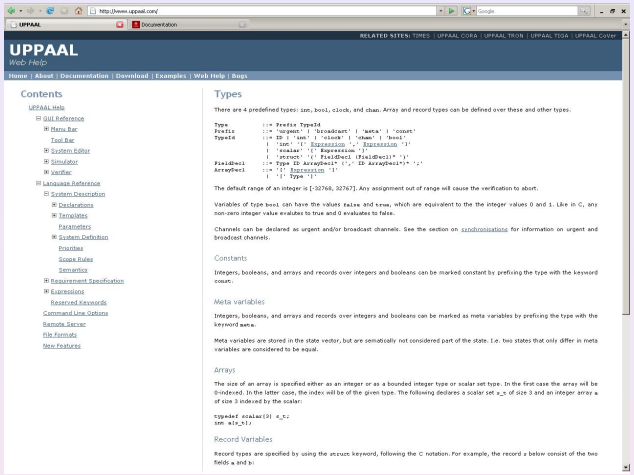
What?

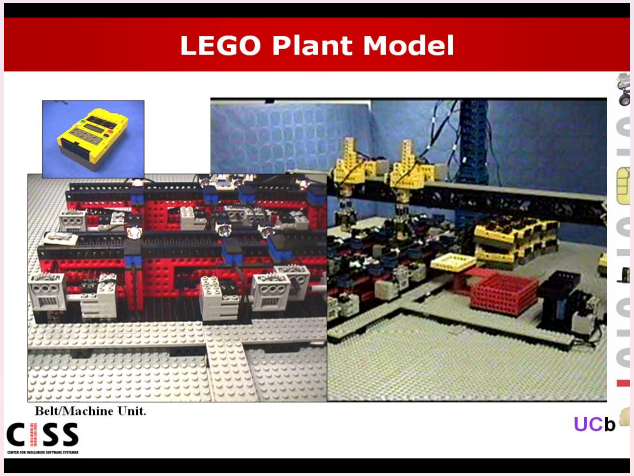
Pros:

1. firm formal foundations,
2. long experience and reasonable stability;
3. user-friendly environment,
4. applied to industrial case-studies,
5. good support and constant development,
6. detailed documentation and help.

Always check:

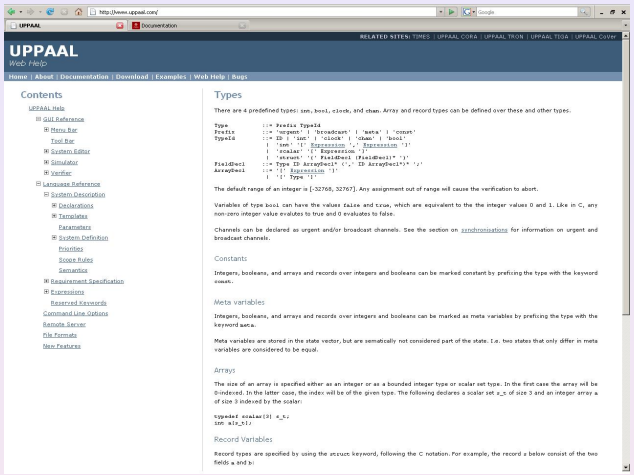
1. help menu (or web help) and
2. `http://www.uppaal.com`

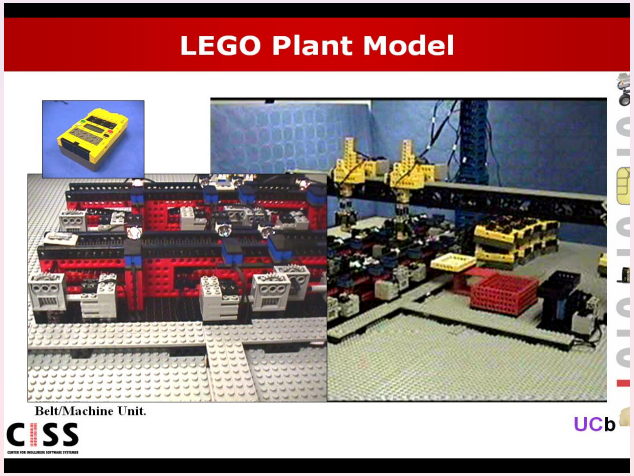




How?

1. specification: networks of timed-automata = automata + communication + clocks and constraints,
2. simulation: timed-traces and message sequence charts,
3. verification: requirement specification language (a temporal logic)





Installation

Peanuts!

1. install JRE: Java Runtime Environment from `http://www.java.com/`,
2. download and extract UPPAAL from `http://www.uppaal.com/`
3. run uppaal.jar

Running Example

Train

1. keep on approaching the bridge;
2. at most one should pass the bridge at a time;
3. announce the arrival to the controller;
4. do not hear anything within 10 seconds $\Rightarrow$ proceed to the bridge;
5. receive a stop signal $\Rightarrow$ wait for the go signal;
6. re-starting takes 10-20 seconds;
7. passing the bridge takes 4-5 seconds;
8. announce the departure

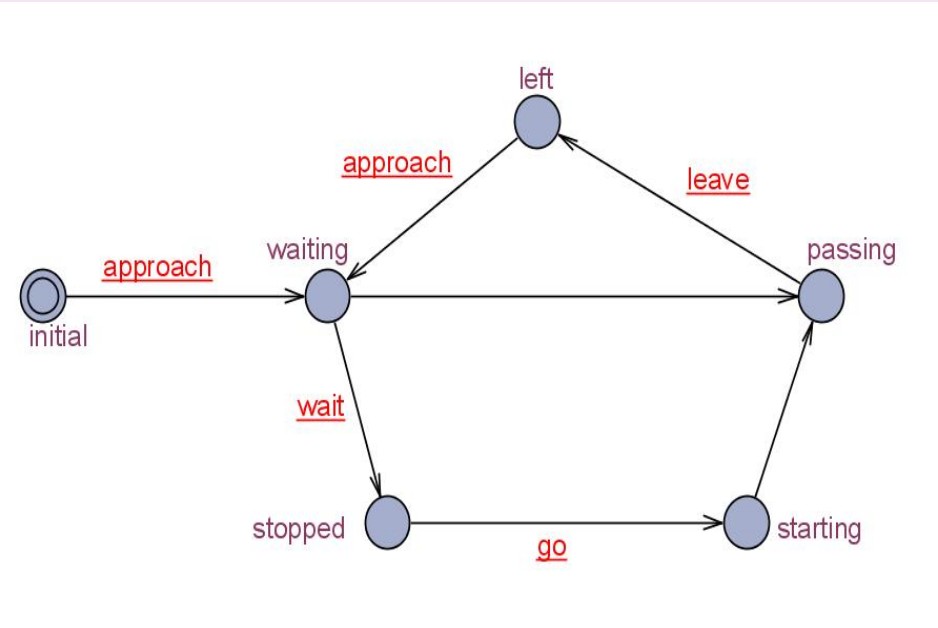
Running Example

Train

1. keep on approaching the bridge;
2. at most one should pass the bridge at a time;
3. announce the arrival to the controller;
4. do not hear anything within 10 seconds $\Rightarrow$ proceed to the bridge;
5. receive a stop signal $\Rightarrow$ wait for the go signal;
6. re-starting takes 10-20 seconds;
7. passing the bridge takes 4-5 seconds;
8. announce the departure

Running Example

Train1



Communication

Channels

1. definition: `chan approach, wait, go, leave;`
2. send *approach*!
3. receive *approach*?

Bells and Whistles

Declaration

1. types: `typedef int[1,4] id;` predefined types:
   1. `int`
   2. `bool`
   3. arrays: `int a[2];`
   4. records: `struct  int a; bool b;  s;`
   5. `chan`
   6. `clock` (bear with me, till the end)
   7. `scalar` (not used for OGO)
2. variables: `id i = 2;`
3. constants: `const bool[2] bs = {true, false};`

Bells and Whistles

Template

1. name: `Train`
2. parameters: `const int[1,3] id`

System

1. instantiate templates: `train1 = Train(1);`
2. compose: `system controller, train1, train2;`

Simulator

Features

1. visualized execution:
   1. manual: choose the next move (backtracking always possible)
   2. random
   3. save, load, replay, change speed

Views

1. trace
2. automata
3. message sequence chart

Simulator

Features

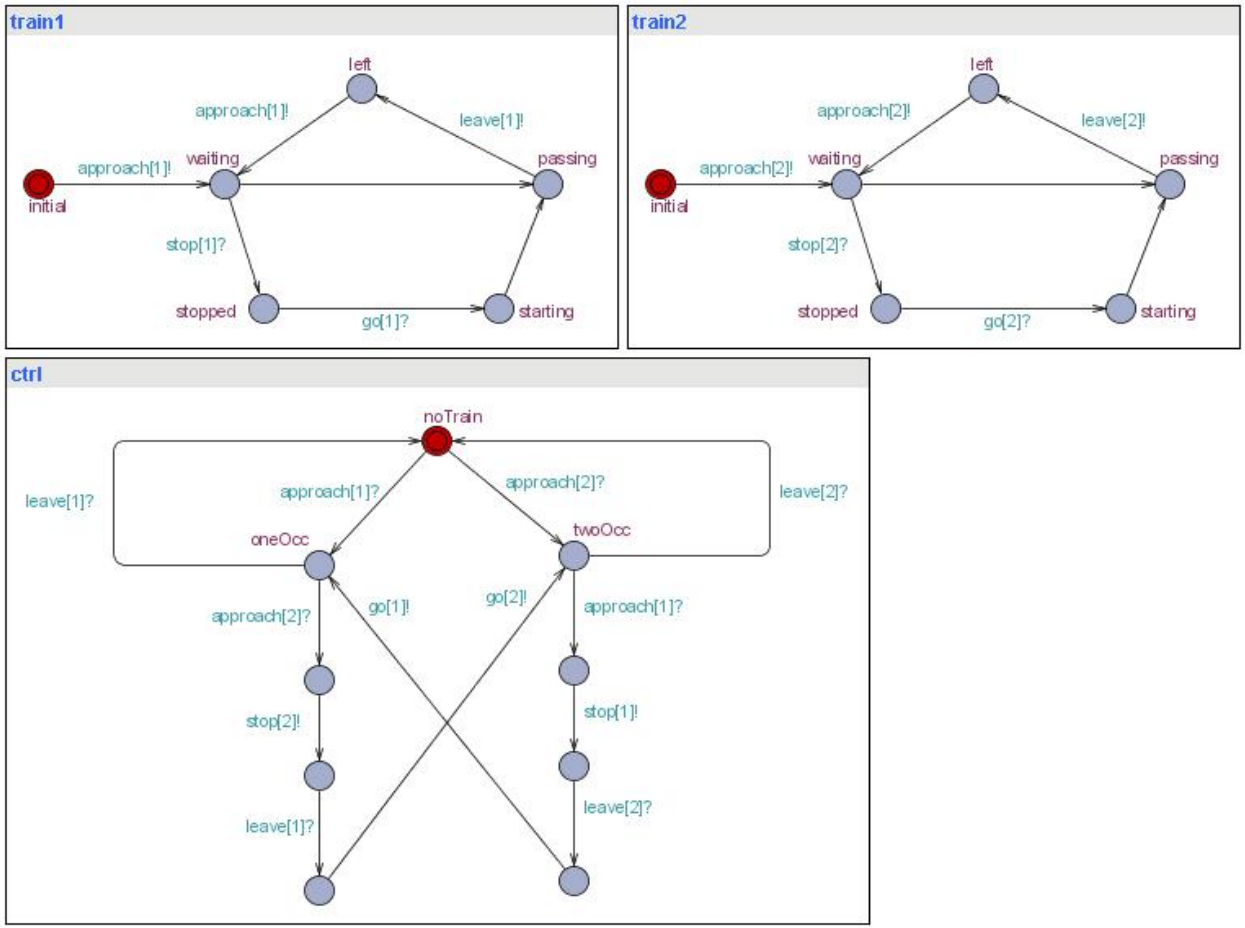
1. visualized execution:
   1. manual: choose the next move (backtracking always possible)
   2. random
   3. save, load, replay, change speed

Views

1. trace
2. automata
3. message sequence chart

Running Example: Revisited

Trains and Controller



Specification Language

Constructs

1. propositions: `train.passing` or `x <= 2` or `deadlock`;
2. invariants: `A[] not train1.passing or not train2.passing`
3. liveness: `A<> train1.passing`
4. deadlock: `E<> deadlock`
5. response: `train1.waiting --> train1.passing`

Verification Features

1. check the property (over all possible executions);
2. generate diagnostic trace (random, shortest, fastest, ...);
3. use simulator to visualize the trace

Specification Language

Constructs

1. propositions: `train.passing` or `x <= 2` or `deadlock`;
2. invariants: `A[] not train1.passing or not train2.passing`
3. liveness: `A<> train1.passing`
4. deadlock: `E<> deadlock`
5. response: `train1.waiting --> train1.passing`

Verification Features

1. check the property (over all possible executions);
2. generate diagnostic trace (random, shortest, fastest, ...);
3. use simulator to visualize the trace

Specification Language

Constructs

1. propositions: `train.passing` or `x <= 2` or `deadlock`;
2. invariants: `A[] not train1.passing or not train2.passing`
3. liveness: `A<> train1.passing`
4. deadlock: `E<> deadlock`
5. response: `train1.waiting --> train1.passing`

Verification Features

1. check the property (over all possible executions);
2. generate diagnostic trace (random, shortest, fastest, ...);
3. use simulator to visualize the trace

Specification Language

Constructs

1. propositions: `train.passing` or `x <= 2` or `deadlock`;
2. invariants: `A[] not train1.passing or not train2.passing`
3. liveness: `A<> train1.passing`
4. deadlock: `E<> deadlock`
5. response: `train1.waiting --> train1.passing`

Verification Features

1. check the property (over all possible executions);
2. generate diagnostic trace (random, shortest, fastest, ...);
3. use simulator to visualize the trace

Specification Language

Constructs

1. propositions: `train.passing` or `x <= 2` or `deadlock`;
2. invariants: `A[] not train1.passing or not train2.passing`
3. liveness: `A<> train1.passing`
4. deadlock: `E<> deadlock`
5. response: `train1.waiting --> train1.passing`

Verification Features

1. check the property (over all possible executions);
2. generate diagnostic trace (random, shortest, fastest, ...);
3. use simulator to visualize the trace

Specification Language

Constructs

1. propositions: `train.passing` or `x <= 2` or `deadlock`;
2. invariants: `A[] not train1.passing or not train2.passing`
3. liveness: `A<> train1.passing`
4. deadlock: `E<> deadlock`
5. response: `train1.waiting --> train1.passing`

Verification Features

1. check the property (over all possible executions);
2. generate diagnostic trace (random, shortest, fastest, ...);
3. use simulator to visualize the trace

Time

Clocks

1. real-valued variables;
2. increasing constantly at the same rate;
3. initially set to 0
4. can be reset at transitions;

Clock Expressions

1. clocks can be compared with integers
   `x <= 5 && y >= 5;`
2. but not with each other: `x <= y;`
3. can be used to specify:
   1. state invariants;
   2. transition guards

Time

Urgency

1. urgent channels: `urgent chan sneldienst;` time cannot pass once urgent synch. can happen no clock guards;
2. urgent locations: time cannot pass in an urgent location
   (committed location: a more restricted variant)

OGO 1.3

Tasks

1. model the system and the environment:
   1. hardware,
   2. control and user
2. specify at least 3 properties, such as:
   1. after emergency button is pressed, the system will stop in 1 second;
   2. after a wafer enters the system, in 10 seconds, it will either be lost, or in one of the two boxes
3. estimate the speed (timing) of components when necessary;
4. simulate and verify (prove) the properties correct.