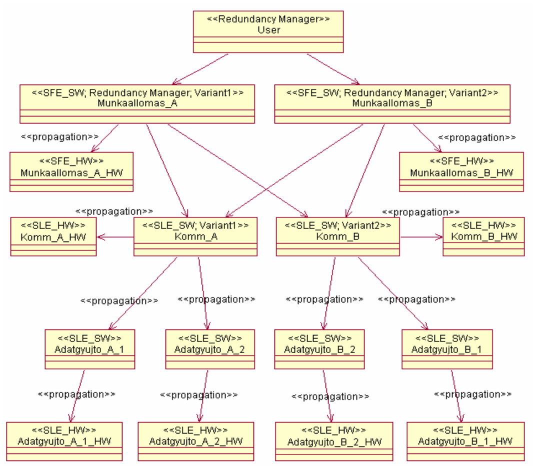
Formal Modelling and Verification

Design and Integration of Embedded Systems

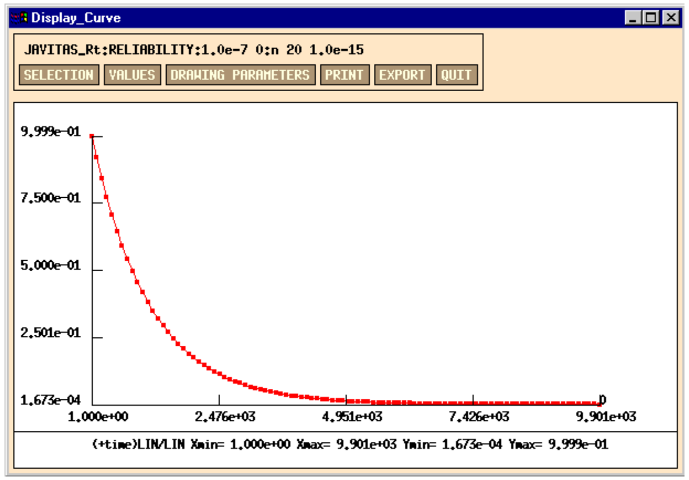
István Majzik

Department of Measurement and Information Systems

The role of formal verification



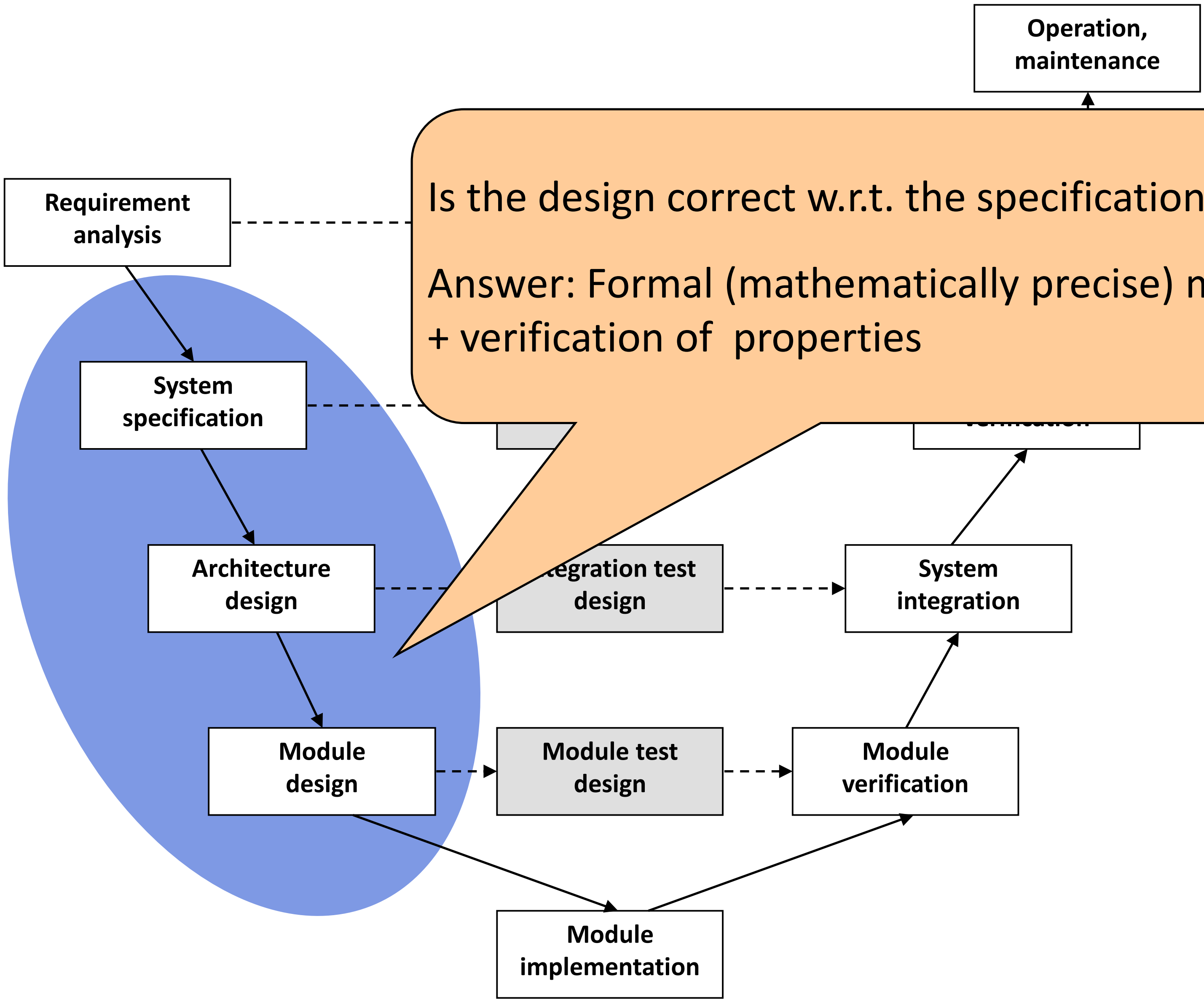
<<Redundancy Manager>> User
<<SFE_SW; Redundancy Manager; Variant1>> Munkaallomas_A
<<SFE_SW; Redundancy Manager; Variant2>> Munkaallomas_B
<<propagation>>
<<SFE_HW>> Munkaallomas_A_HW
<<SFE_HW>> Munkaallomas_B_HW
<<SLE_HW>> Komm_A_HW
<<SLE_SW; Variant1>> Komm_A
<<SLE_SW; Variant2>> Komm_B
<<SLE_HW>> Komm_B_HW
<<SLE_SW>> Adatgyujto_A_1
<<SLE_SW>> Adatgyujto_A_2
<<SLE_SW>> Adatgyujto_B_2
<<SLE_SW>> Adatgyujto_B_1
<<SLE_HW>> Adatgyujto_A_1_HW
<<SLE_HW>> Adatgyujto_A_2_HW
<<SLE_HW>> Adatgyujto_B_2_HW
<<SLE_HW>> Adatgyujto_B_1_HW



JAVITAS_Rt:RELIABILITY:1,0e-7 0:n 20 1,0e-15

(+time)LIN/LIN Xmin= 1,000e+00 Xmax= 9,901e+03 Ymin= 1,673e-04 Ymax= 9,999e-01

Example software lifecycle (V-model)



Is the design correct w.r.t. the specification?

Answer: Formal (mathematically precise) modeling + verification of properties

Techniques and measures in standards

- IEC 61508: Functional safety in electrical / electronic / programmable electronic safety-related systems
- Example: Software architecture design

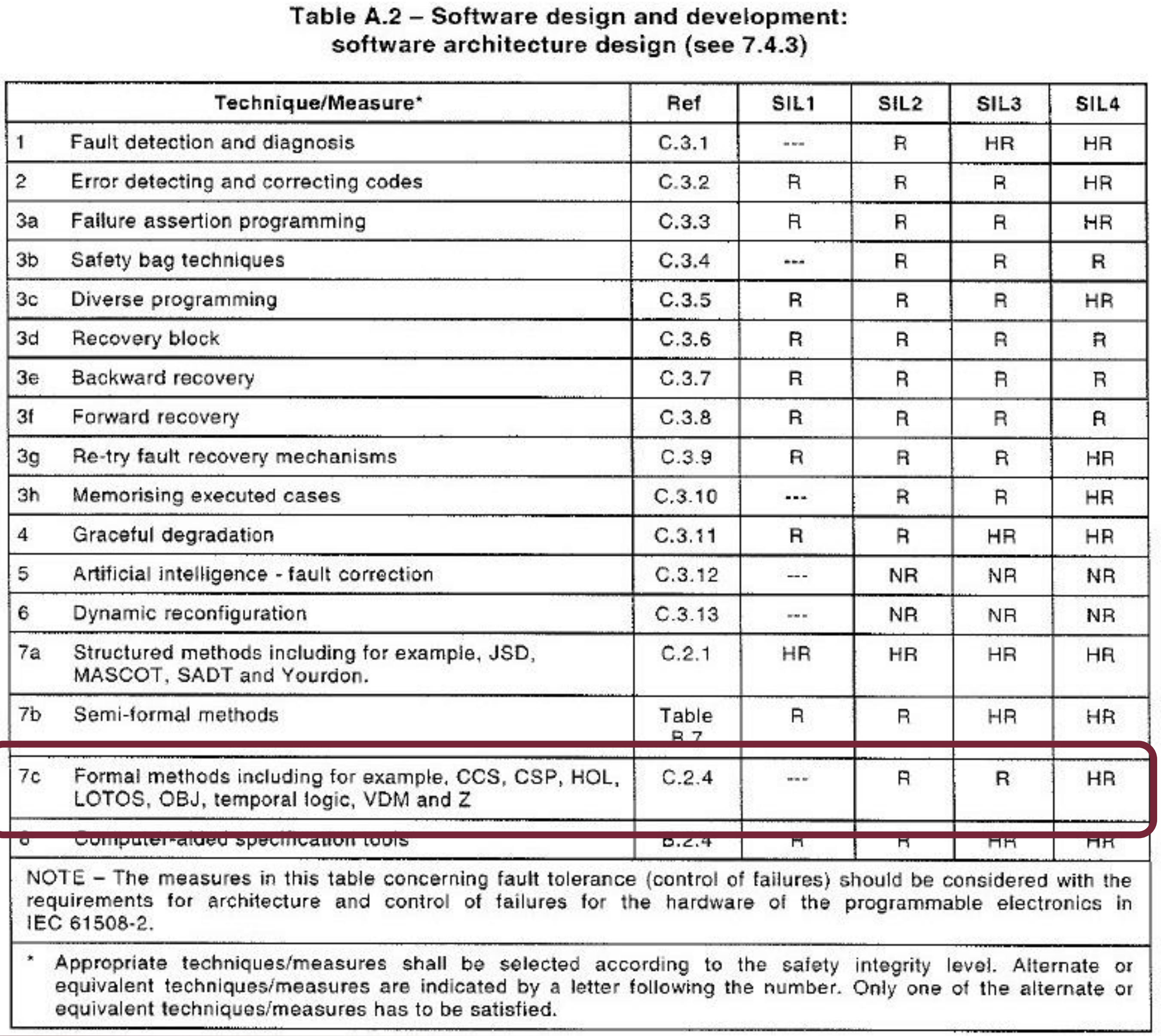


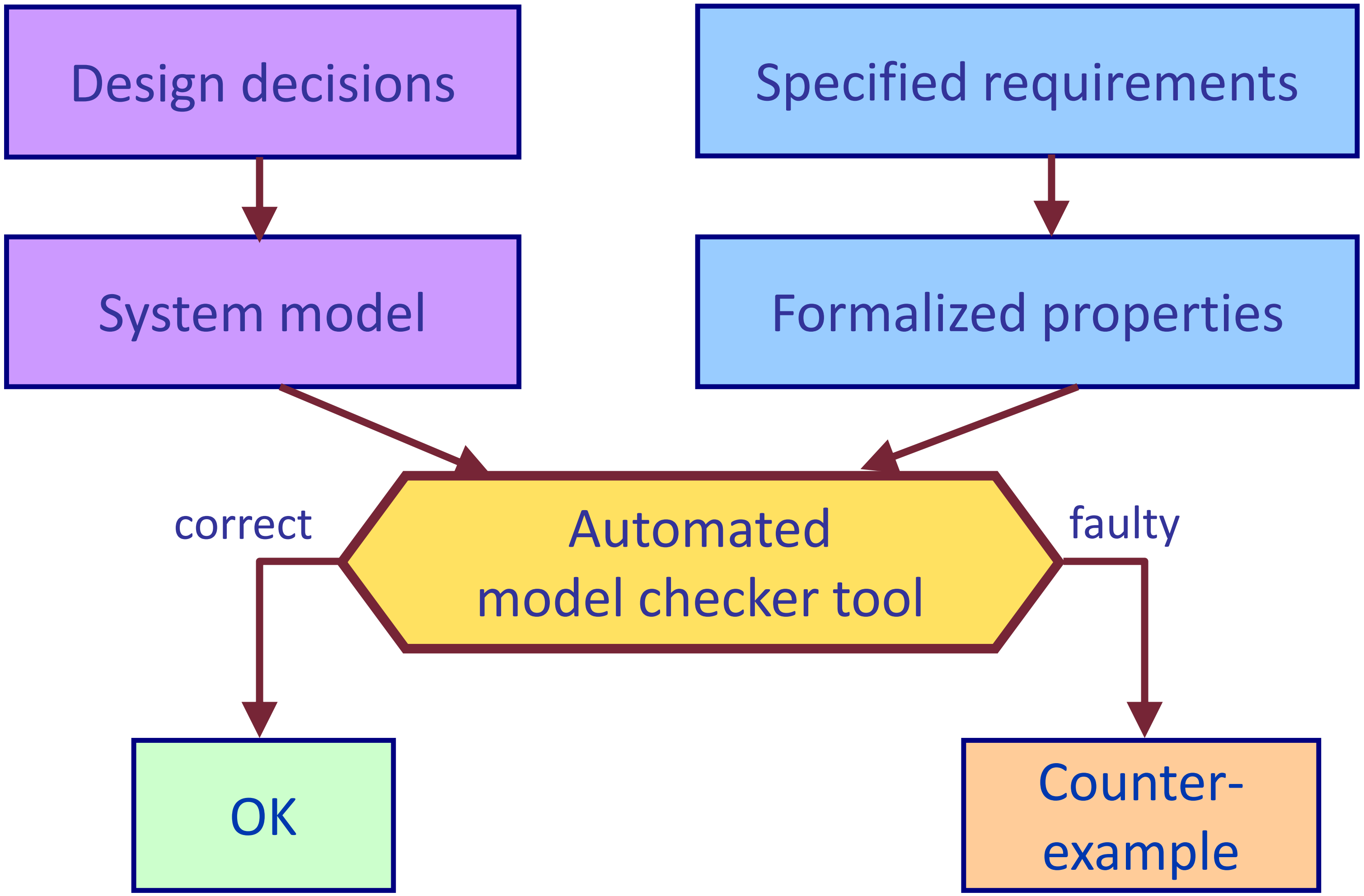
Table A.2 – Software design and development: software architecture design (see 7.4.3)

	Technique/Measure*	Ref	SIL1	SIL2	SIL3	SIL4
1	Fault detection and diagnosis	C.3.1	---	R	HR	HR
2	Error detecting and correcting codes	C.3.2	R	R	R	HR
3a	Failure assertion programming	C.3.3	R	R	R	HR
3b	Safety bag techniques	C.3.4	---	R	R	R
3c	Diverse programming	C.3.5	R	R	R	HR
3d	Recovery block	C.3.6	R	R	R	R
3e	Backward recovery	C.3.7	R	R	R	R
3f	Forward recovery	C.3.8	R	R	R	R
3g	Re-try fault recovery mechanisms	C.3.9	R	R	R	HR
3h	Memorising executed cases	C.3.10	---	R	R	HR
4	Graceful degradation	C.3.11	R	R	HR	HR
5	Artificial intelligence - fault correction	C.3.12	---	NR	NR	NR
6	Dynamic reconfiguration	C.3.13	---	NR	NR	NR
7a	Structured methods including for example, JSD, MASCOT, SADT and Yourdon.	C.2.1	HR	HR	HR	HR
7b	Semi-formal methods	Table B.7	R	R	HR	HR
7c	Formal methods including for example, CCS, CSP, HOL, LOTOS, OBJ, temporal logic, VDM and Z	C.2.4	---	R	R	HR
8	Computer-aided specification tools	B.2.4	R	R	HR	HR

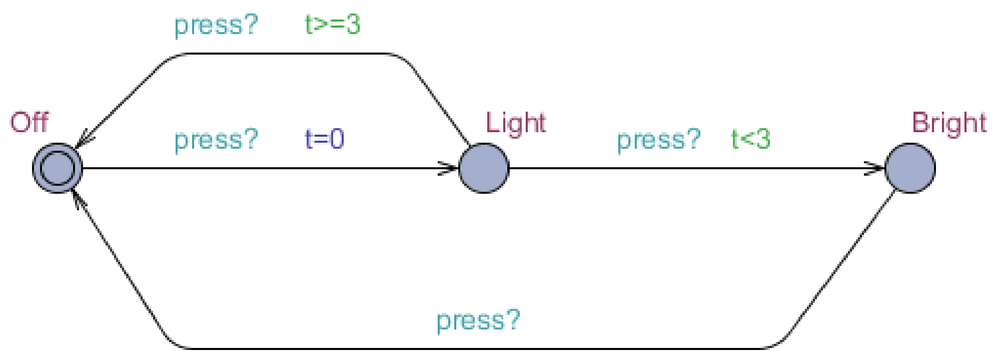
NOTE – The measures in this table concerning fault tolerance (control of failures) should be considered with the requirements for architecture and control of failures for the hardware of the programmable electronics in IEC 61508-2.

* Appropriate techniques/measures shall be selected according to the safety integrity level. Alternate or equivalent techniques/measures are indicated by a letter following the number. Only one of the alternate or equivalent techniques/measures has to be satisfied.

Goals of formal modeling and verification

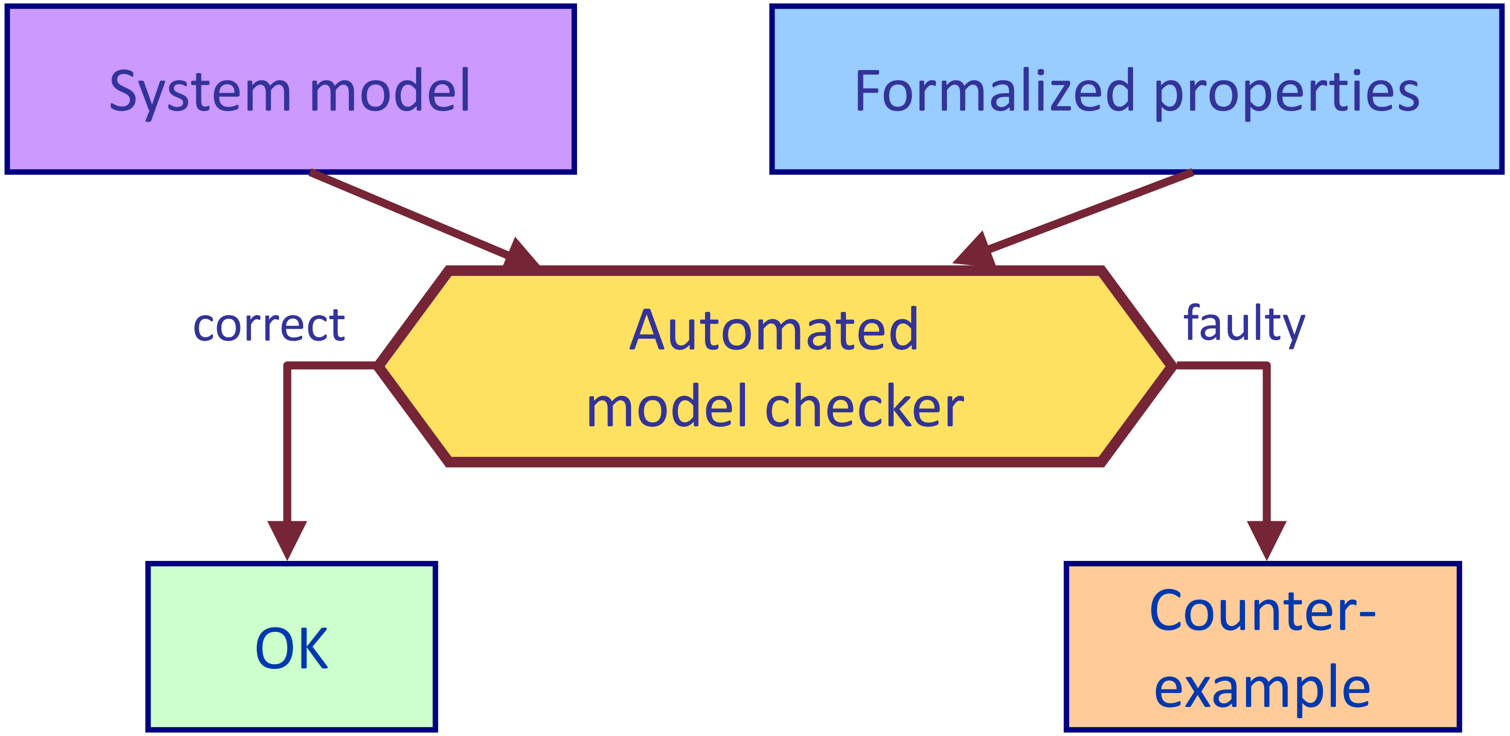


Modeling with timed automata



Goals of formal modeling and verification

- Modeling with timed automata
- Timed automata can also be derived from higher-level models (e.g., from UML state machines)



Automata and variables

- Goal: Modeling event driven, state based behavior
- Basic formalism: Finite state machine (FSM)
  - Control locations (with names), as part of the state of the FSM
  - Transitions among control locations
- Extension: Using integer variables
  - Modelling computations with integer arithmetic
  - Types and ranges of potential values can be specified
  - Constants can be defined
- Using integer variables on transitions
  - Guard: Conditions on variables
    (guard shall be true in order to enable the transition)
  - Action: Assignments to the variables

Example: Automaton with variables

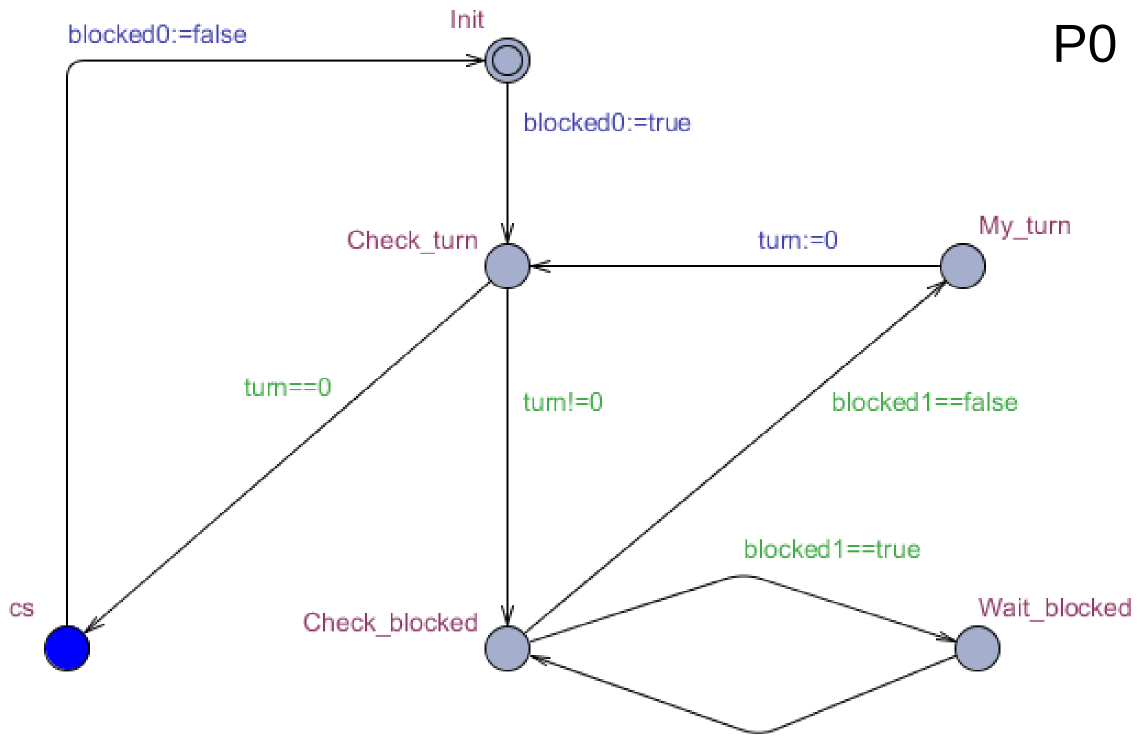
Declarations:

bool blocked0 = false;
bool blocked1 = false;
int turn = 0;

Pseudo-code and model of an automaton:

P0

```
while (true) {
   blocked0 := true;
   while (turn!=0) {
        while (blocked1==true) {
                skip;
        }
        turn := 0;
   }
   // Critical section (cs)
   blocked0 := false;
   // Do other things
}
```



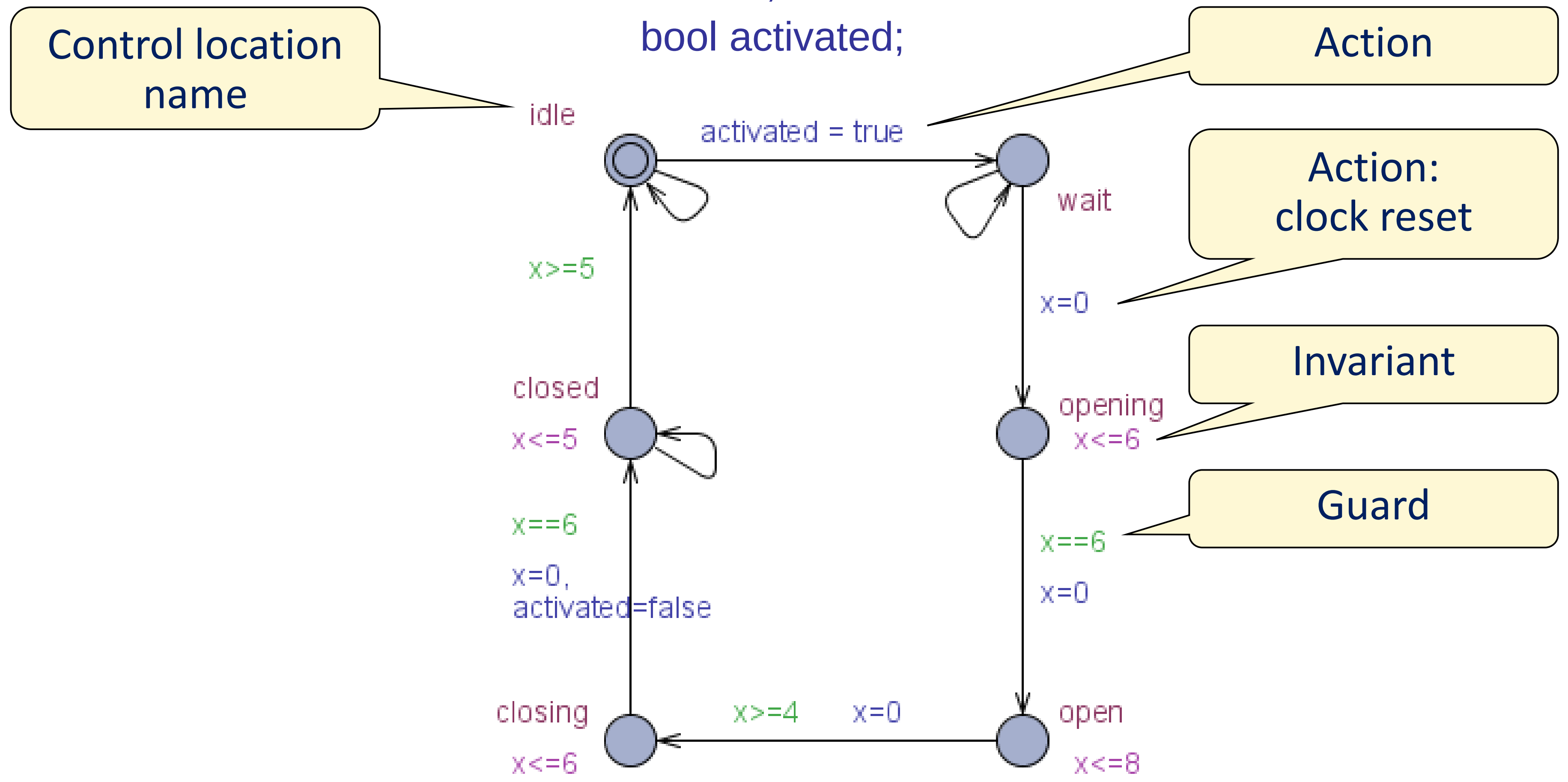
Extensions using clock variables

- Goal: Modelling time dependent behavior
  - Time passes in given states of the component
  - Relative time measurement by resetting and reading timers; behavior depends on timer value (e.g., timeout)
- Model extension: Clock variables
  - Represent timers
  - Automatically measure time elapse by a uniform constant rate
- Using clock variables on transitions:
  - Guard: Condition over clock variables and constants
  - Action: Resetting selected clock variables (independently)
- Use of clock variables in control locations:
  - Location invariant (state invariant): Condition over clock variables, being in a location is valid until its invariant holds

Timed automata (in the UPPAAL tool)

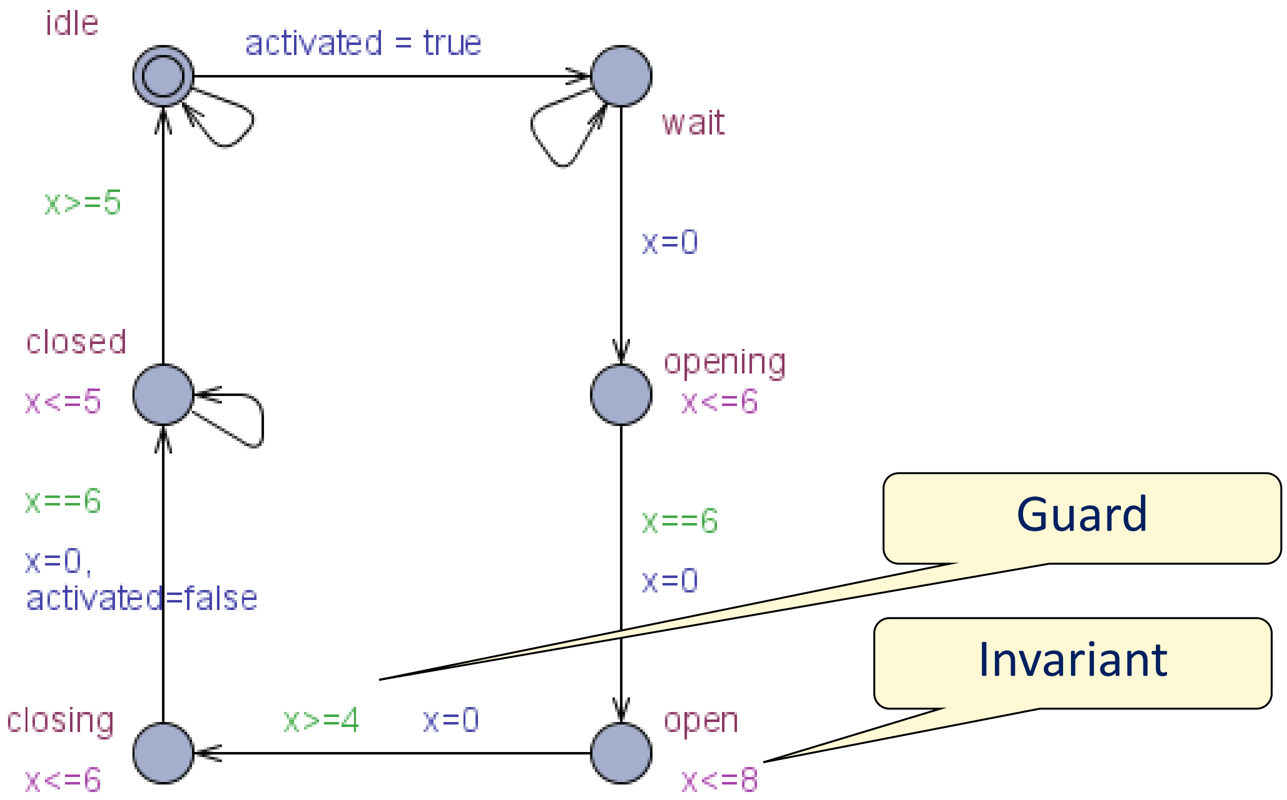
Example: Revolving door

clock x;
bool activated;

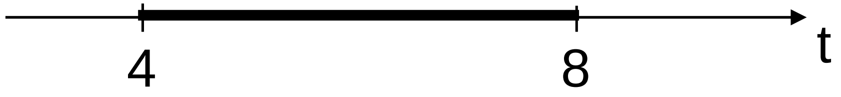


Role of state invariants and guards

clock x;
bool activated;

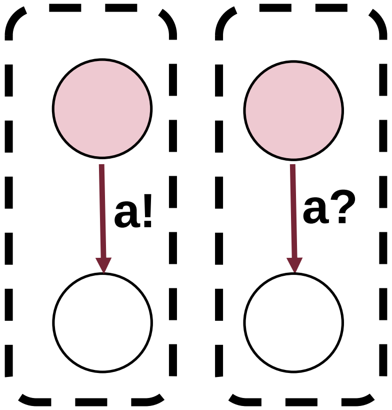


The value of clock x is in the range [4, 8] when leaving the location open



Extensions for modeling distributed systems

- Goal: Modeling networks of interacting timed automata
  - Interaction: Simultaneous execution of transitions in different automata
  - Represents synchronous communication (rendezvous)
    - Sending and receiving of a message occurs at the same time
    - This primitive can also be used to model asynchronous communication
- Model extension: Synchronized actions
  - Channels for message exchange (synchronous channels)
  - Message sending action: ! operator on the channel
    Message receiving action: ? operator on the channel
  - E.g., on the channel a the actions are a! and a?
- Parameterization
  - Arrays of channels (indexed)
    - E.g., a[id] is a channel indexed by the value of variable id
    - Useful in case of several participants and interactions



chan a;

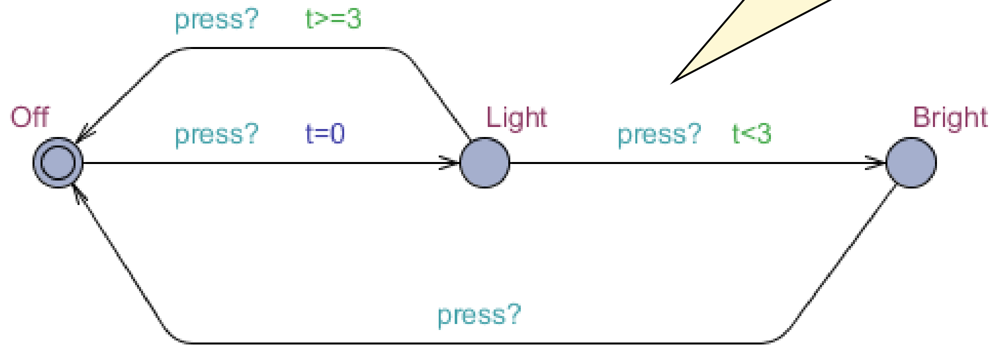
Example: Modeling an interaction (pushing a button)

Declarations:

clock t, u;
chan press;

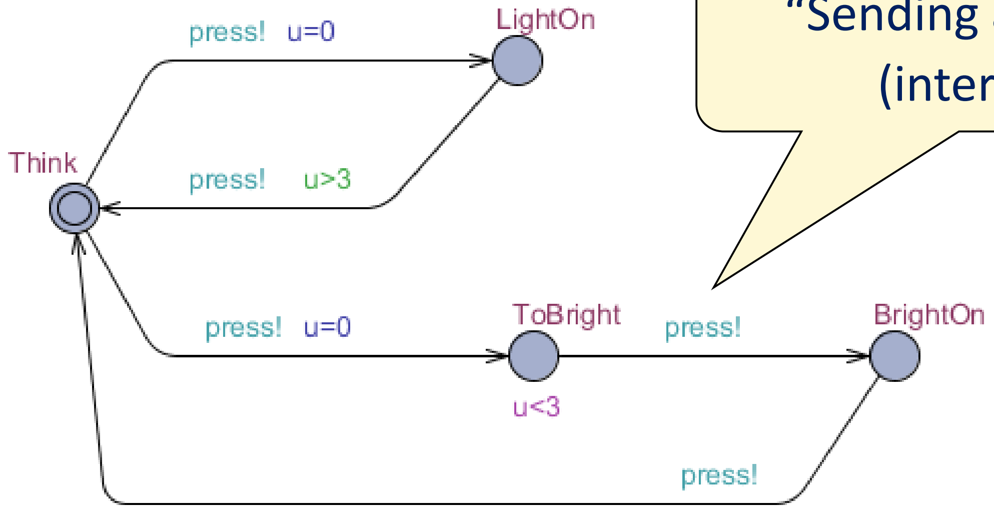
“Receiving a message” (interaction)

Switch:



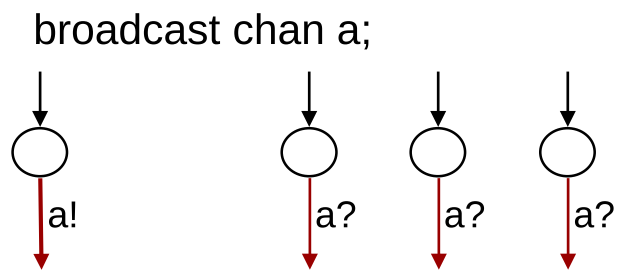
User:

“Sending a message” (interaction)

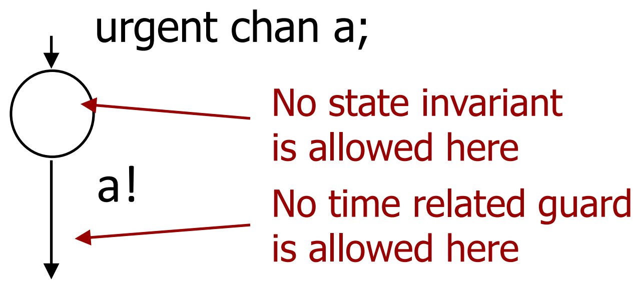


Further extensions

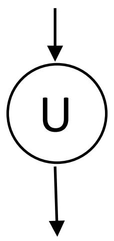
- Broadcast channel
  - Single sender (able to send without receiver)
  - Several receivers (all synchronized that are ready for synchronization)



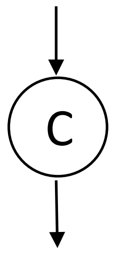
- Urgent channel: prohibit time delay
  - The synchronization is executed without delay (instant transitions are possible before it)


urgent chan a;
a!
No state invariant is allowed here
No time related guard is allowed here

- Urgent state: prohibit time delay
  - Time is not allowed to progress in the state

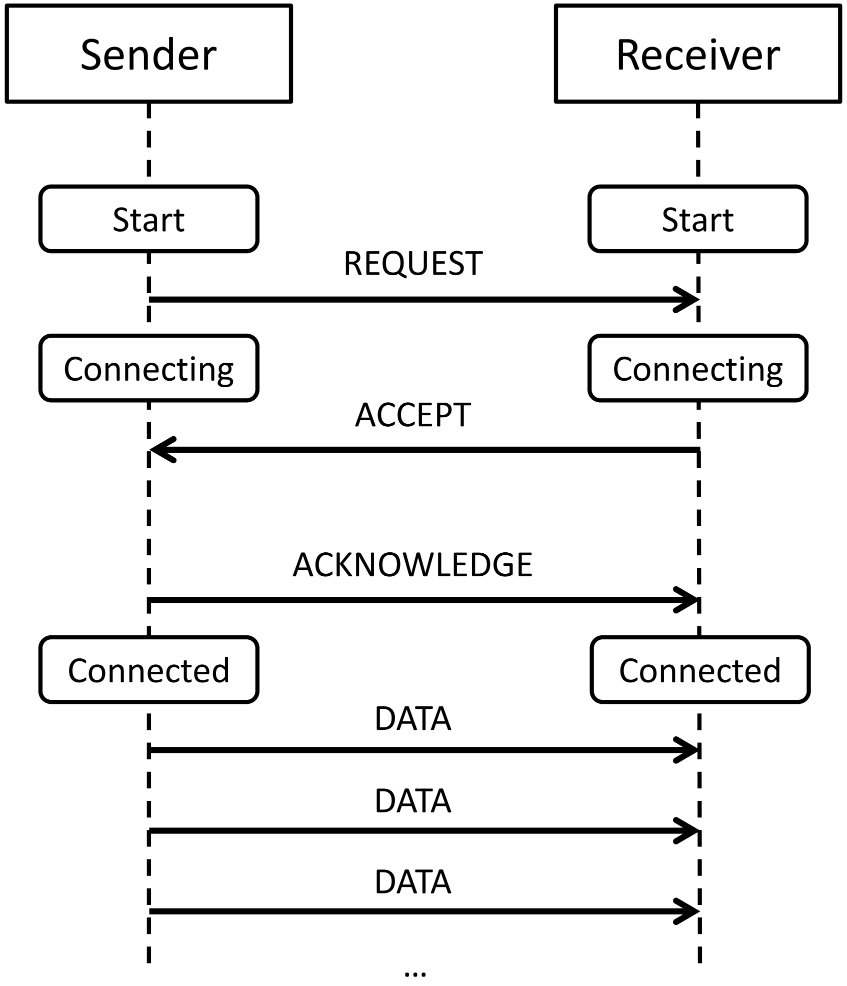


- Committed state: Atomic state transitions
  - Before executing the outgoing transition, execution of a transition of another automaton is not allowed: the incoming and the outgoing transitions are executed in an atomic operation

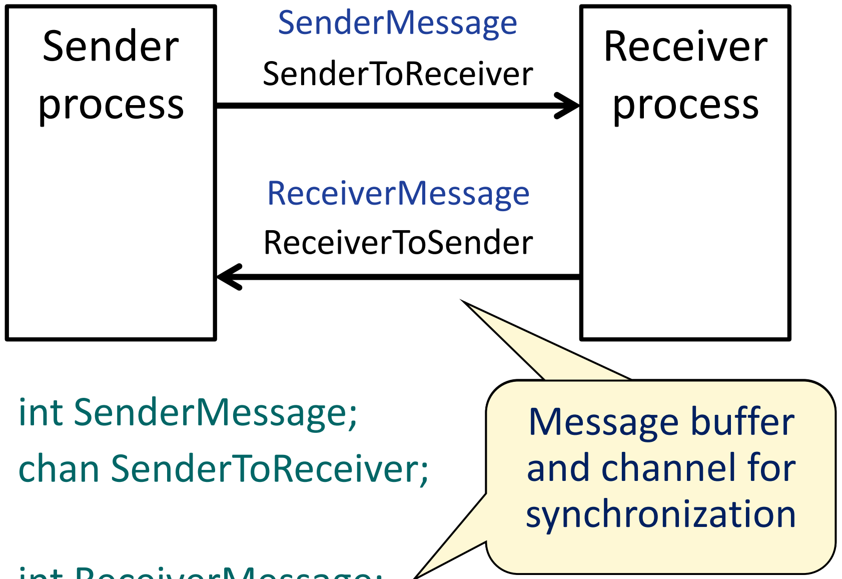


Example: Modeling the transfer of messages

Message sequence:



Structure of the model:



```
int SenderMessage;
chan SenderToReceiver;

int ReceiverMessage;
chan ReceiverToSender;

const int REQUEST = 1;
const int ACCEPT = 2;
const int ACKNOWLEDGE = 3;
const int DATA = 4;
```

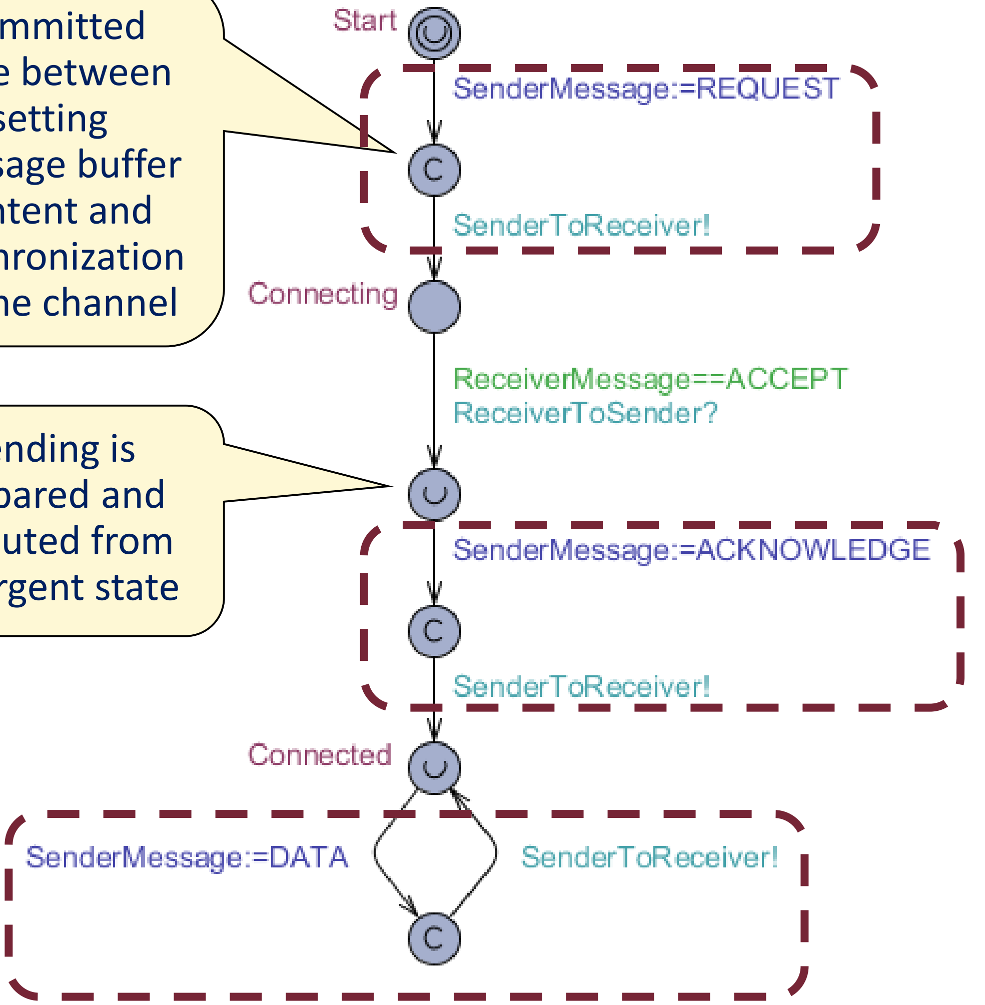
Example: Automata models

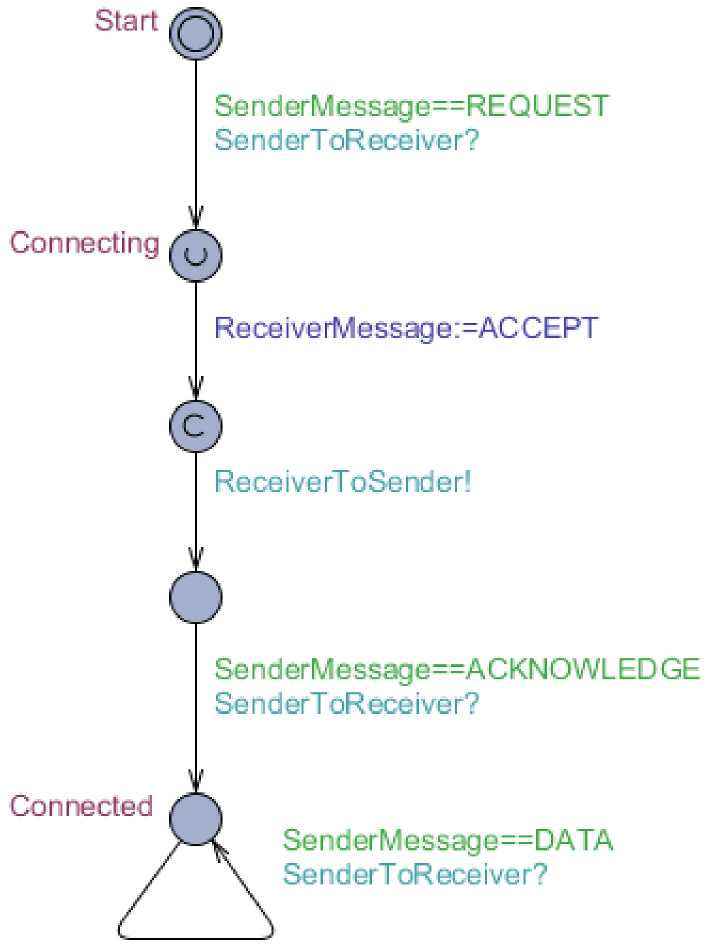
Sender:

Receiver:

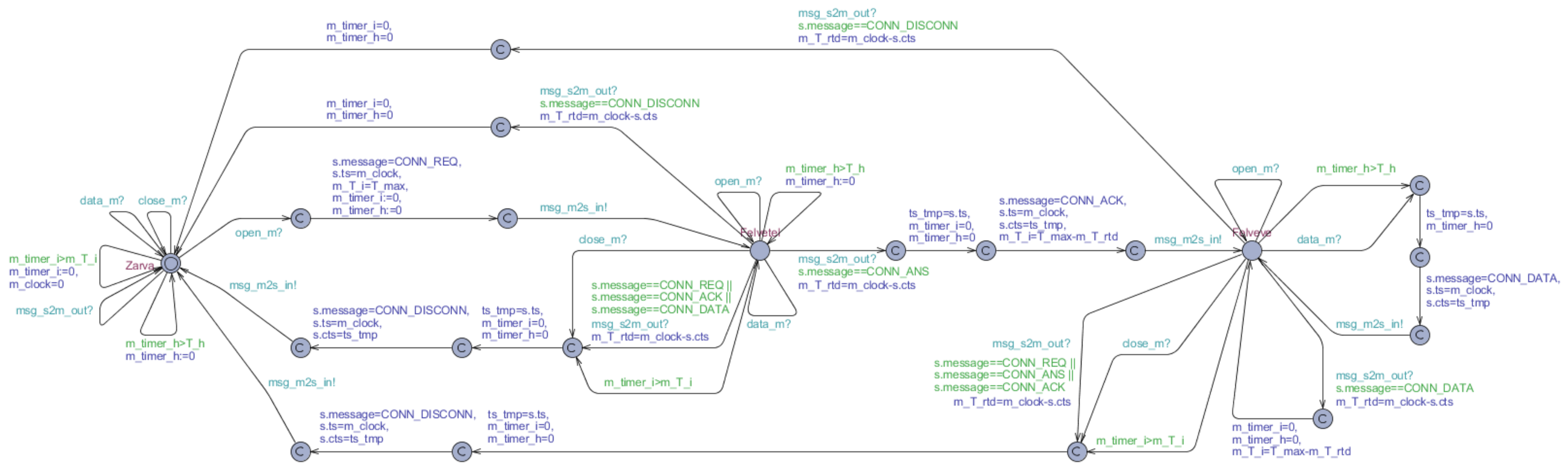
Committed state between setting message buffer content and synchronization on the channel

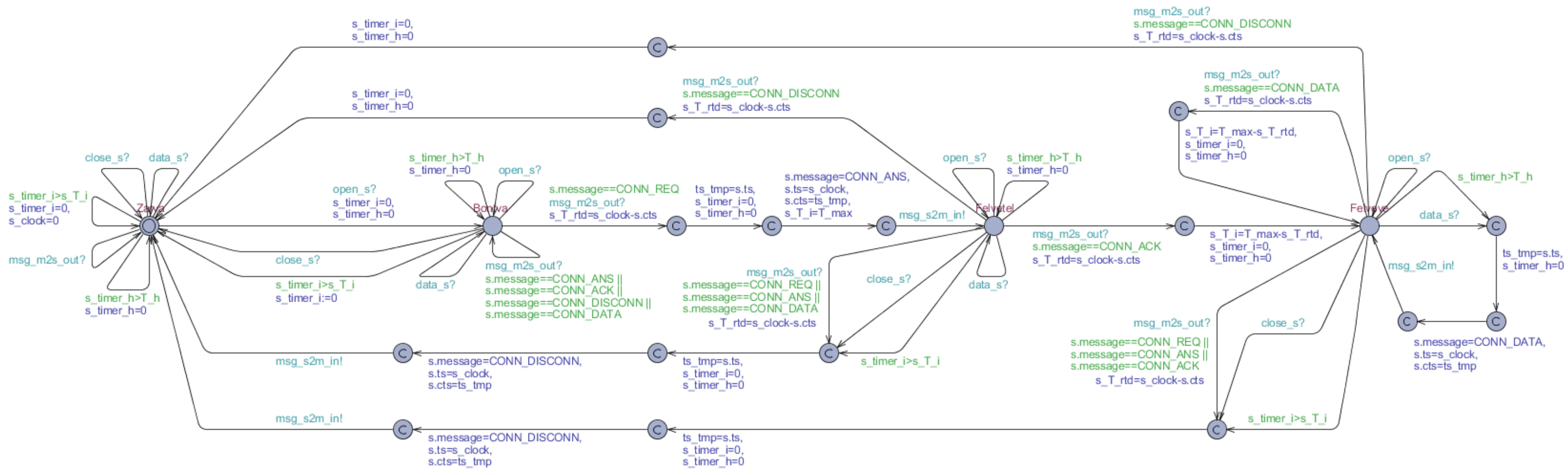
Sending is prepared and executed from an urgent state





Example: Design of real protocols



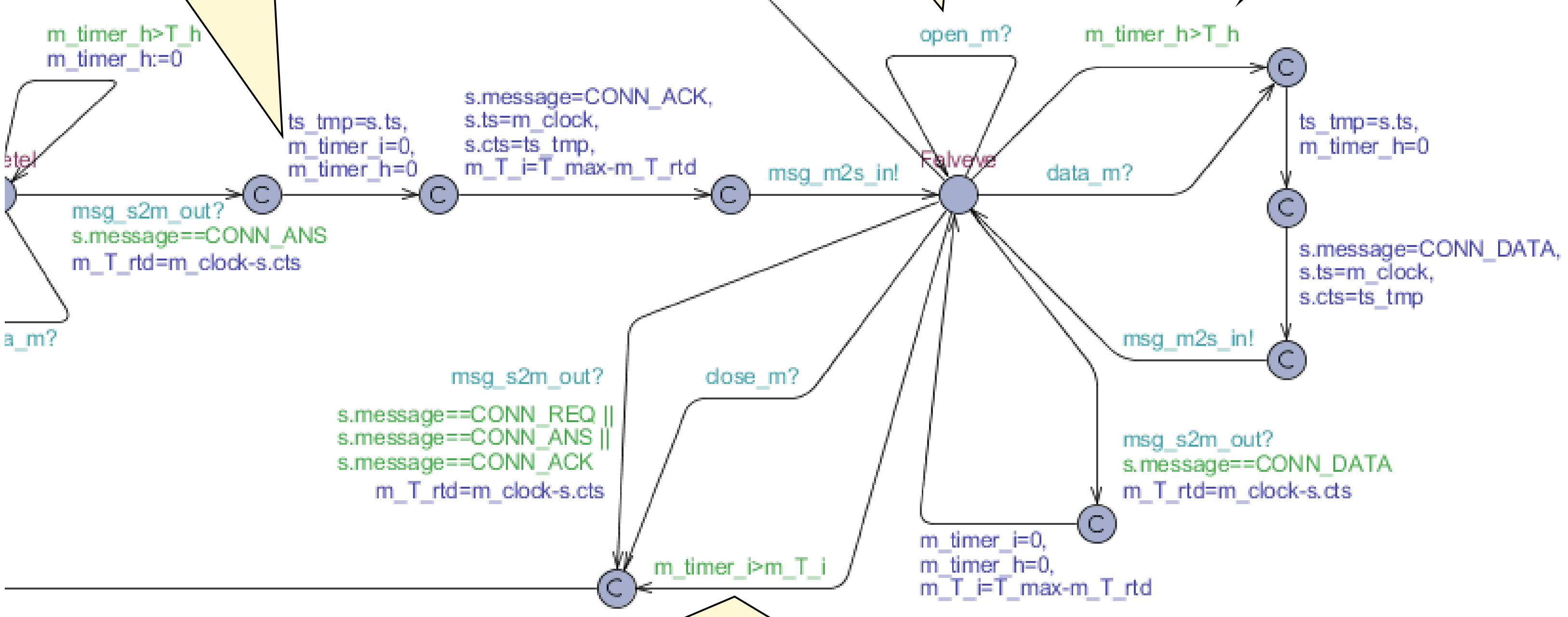


Example: Design of real protocols

Setting timers for measuring timeout and assuring timely communication

What happens in case of irrelevant input messages

Time-dependent transfer of messages

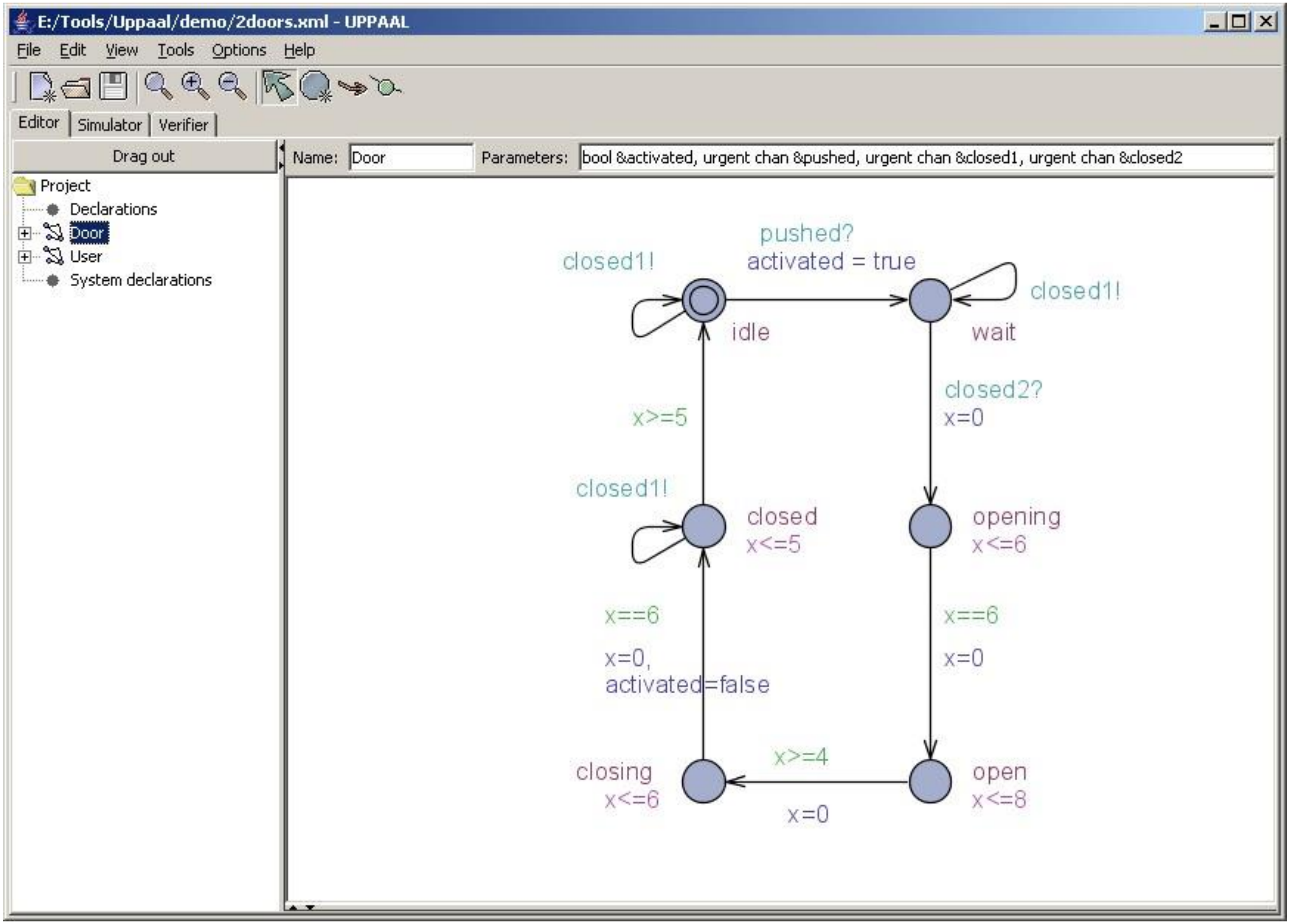


What happens in case of timeout waiting for an expected input message

The UPPAAL tool set

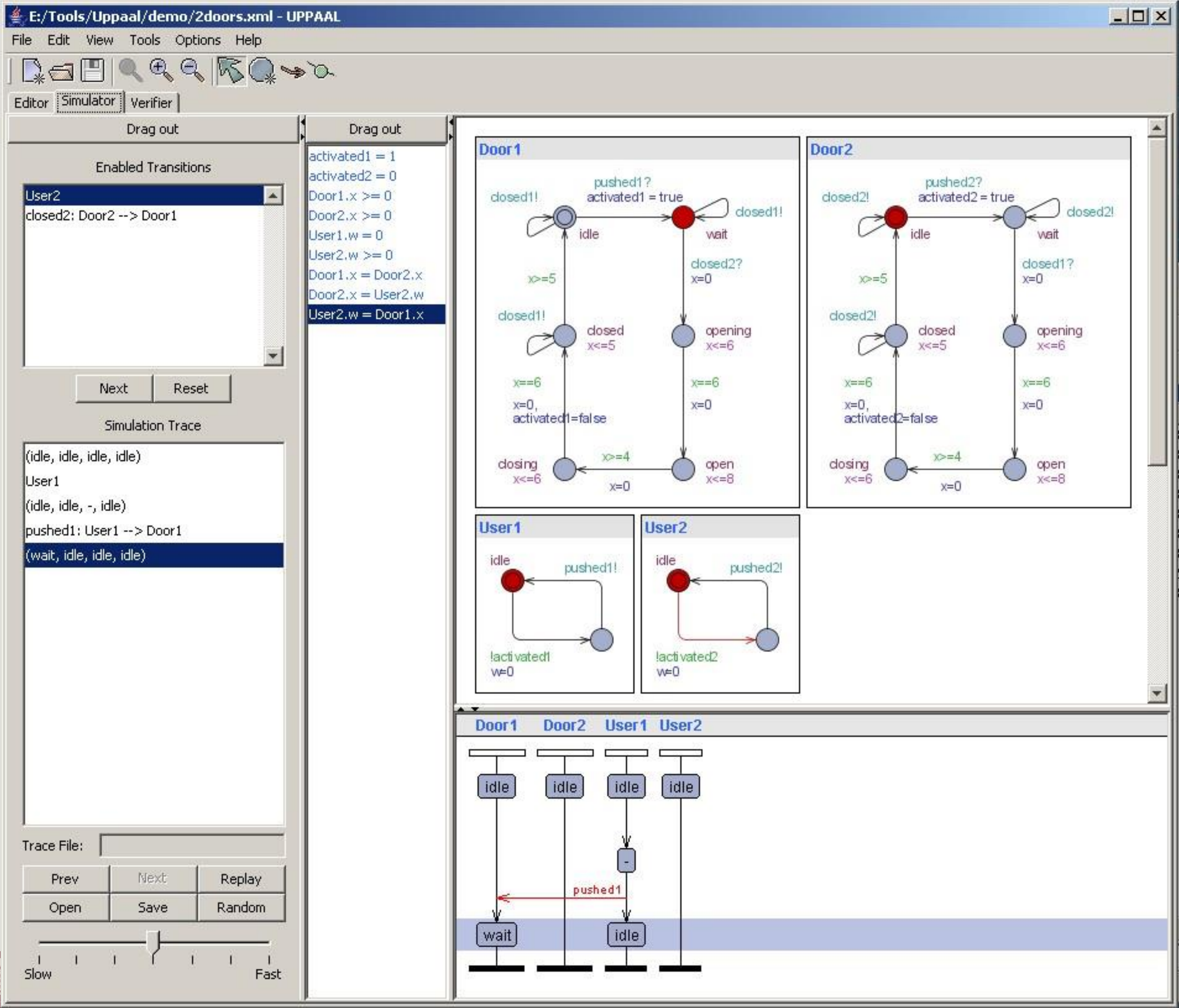
- Development (1999-):
  - Uppsala University, Sweden
  - Aalborg University, Denmark
- Web page (information, downloading, examples):
  http://www.uppaal.org/
- Related tools:
  - UPPAAL CoVer: Test generation
  - UPPAAL TRON: On-line testing
  - UPPAAL PORT: Designing component based systems
  - …
- Commercial version:
  http://www.uppaal.com/

Automaton model



Name: Door Parameters: bool &activated, urgent chan &pushed, urgent chan &closed1, urgent chan &closed2

Simulator



Enabled Transitions:

User2
closed2: Door2 --> Door1

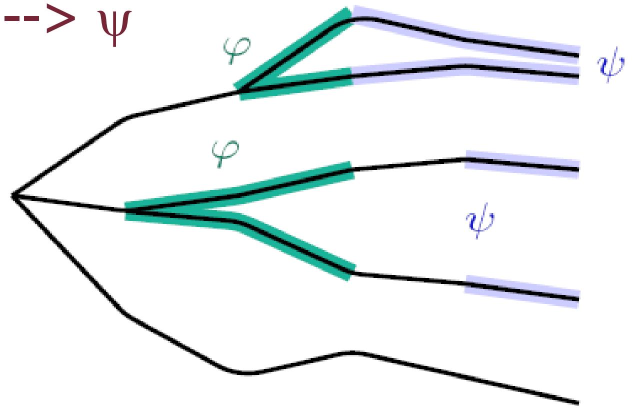
Simulation Trace:

(idle, idle, idle, idle)
User1
(idle, idle, -, idle)
pushed1: User1 --> Door1
(wait, idle, idle, idle)

activated1 = 1
activated2 = 0
Door1.x >= 0
Door2.x >= 0
User1.w = 0
User2.w >= 0
Door1.x = Door2.x
Door2.x = User2.w
User2.w = Door1.x

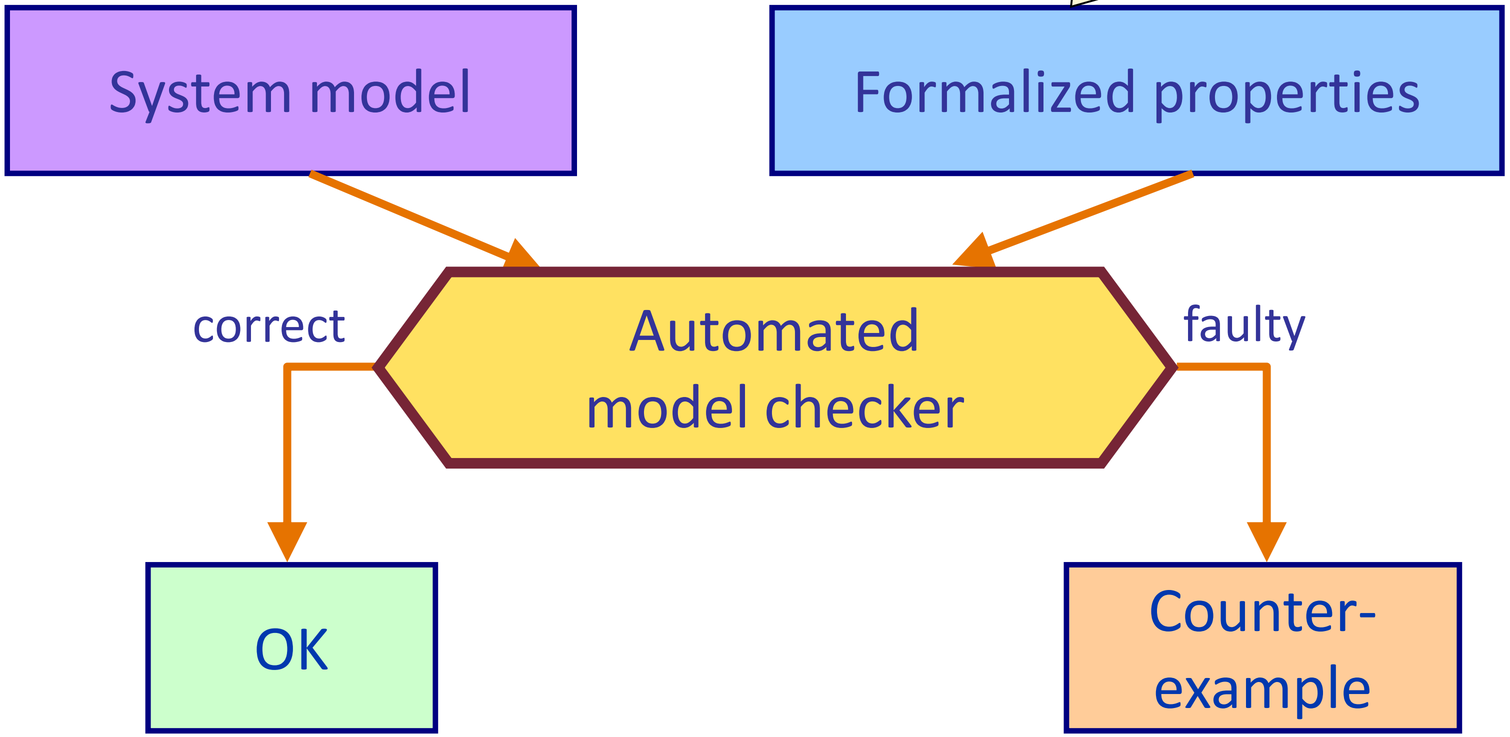
Formalizing requirements with temporal logics

φ --> ψ



Goals of formal modeling and verification

- Precise formalization of properties (requirements) to support automated checking



What are the formalized properties?

An example to illustrate the properties to be formalized:

- The operating modes of an air-conditioner:
  - Switched-off, switched-on, faulty,
    light cooling, strong cooling, heating, ventilating
- Requirements for the air-conditioner:
  - After switched-on, it shall start ventilating
  - Strong cooling is allowed only after light cooling
  - Heating shall be followed by ventilating
  - The faulty air-conditioner shall not perform heating
  - ...

State based properties

- Local: Properties to be evaluated in a given state
  - Evaluation is possible using the current values of the state variables (and clock variables)
  - Example: „In the initial state ventilating shall be provided”
- Reachability: Properties to be evaluated on a sequence (trace) of states
  - Evaluation is possible on the state space of the system
    - Example: „Heating shall be followed by ventilating”
  - Typical categories of reachability properties:
    - „Safety” of the system
    - „Liveness” of the system

Safety and liveness properties

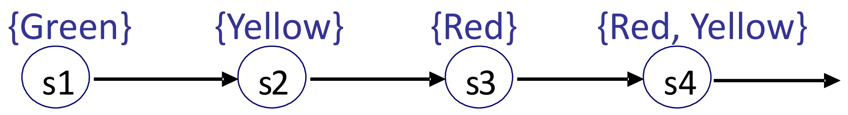
- Safety properties: Specify that each state shall be safe, i.e., “something bad shall never happen”
  - ”In each state the pressure shall be lower than the critical value.”
  - ”In each operating state the door shall be closed.”
  - “There is no deadlock in the protocol.”
  - Invariant properties (i.e., for each state)
- Liveness properties: Specify that a desired state is reachable, i.e., ”something good will happen”
  - “After switch-on, the press shall eventually produce the plate.”
  - “After sending a request the reply shall be received”
  - “The process shall compute the required result”
  - Existential properties (i.e., for the desired state)

Language to formalize reachability properties

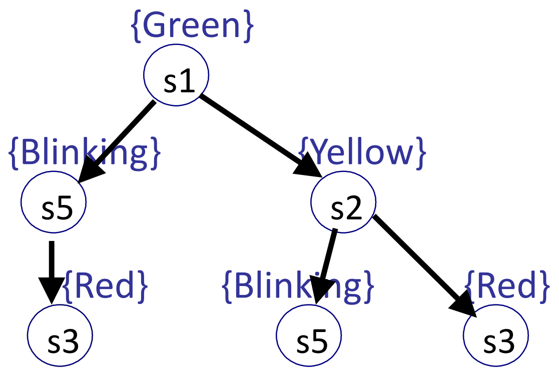
- Reachable states are considered in logic time:
    - The present: The current state
    - The next time point: The subsequent state(s)
- Temporal operators (referring to logic time) are defined to express the reachability properties
    - Typical temporal operators: „always”, „eventually”, „before”, „until”, „after”, …
    - Temporal logic: Formal language to express propositions qualified in terms of logic time

Temporal logics

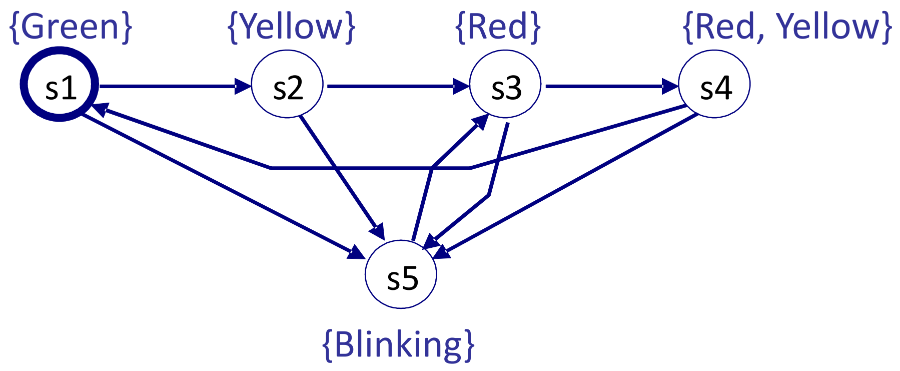
- Linear time:
  The subsequent states form a linear sequence: each state has only one successor
  → logic time forms a linear timeline



- Branching time:
  The subsequent states form a tree structure: each state may have multiple successors
  → logic time forms branching timelines



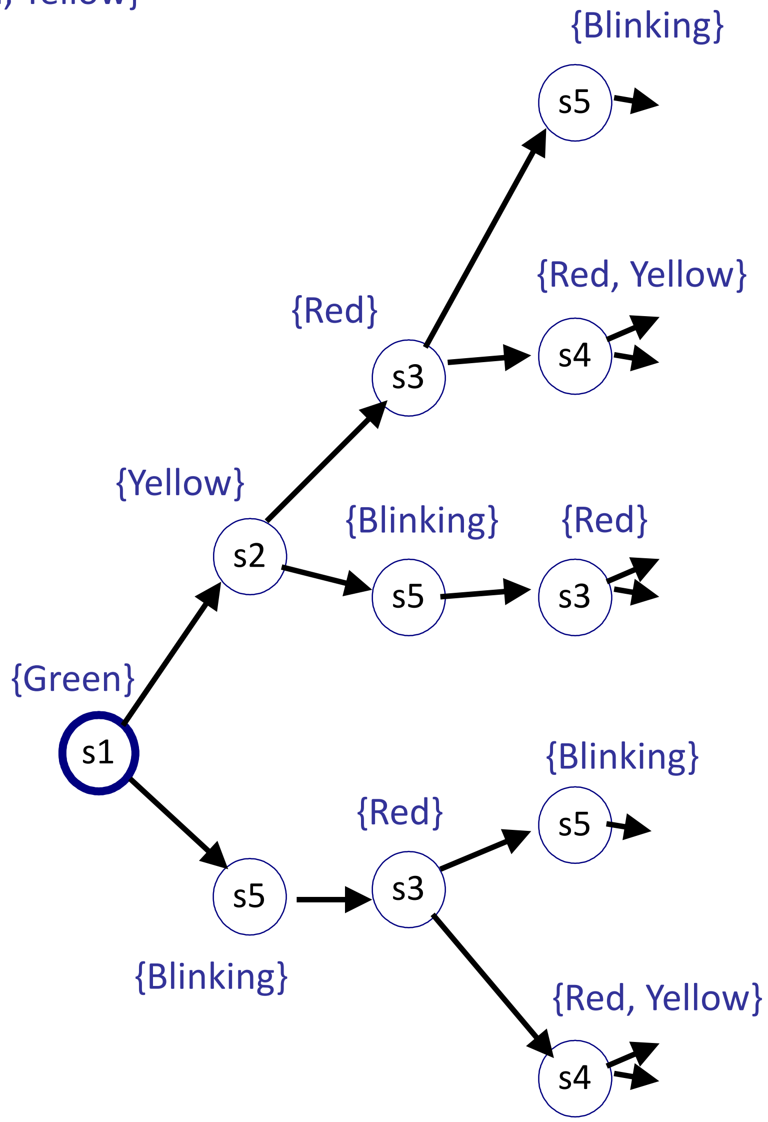
The computational tree



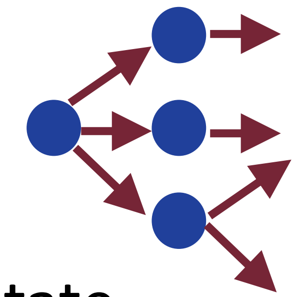
↑

Automaton (FSM)
with labelled states

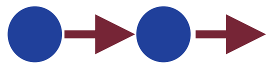
Computational tree:
Structure of the potential successor states



Quantifying paths and characterizing states



- Operators that quantify the paths starting from a given state:
  - A: for all paths from the given state
  - E: for at least one (existing) path from the given state



- Operators that characterize states along a given path:
  - F: for a state eventually along the path (“future”)
  - G: for all states along the path (“globally”)
  - X: for the next state of the path (“next”)
  - U: for states until reaching a specified state (“until”)
    - E.g., Yellow U Red means that states shall be labeled with Yellow until reaching a state labeled with Red

The Computational Tree Logic (CTL)

- Composite operators are formed
  - First quantifying paths using operators A, E; then characterizing states along the path by operators F, G, X, U
  - Composite operators:
    - For all paths: AF, AG, AX, A(. U .)
    - For at least one path: EF, EG, EX, E(. U .)
  - Examples:
    - EF Red: There shall exist a path where a state with Red is reached
    - AG Green: For all paths, all states shall be labeled with Green
    - E(Yellow U Red): For at least one path, states shall be labeled with Yellow until a state with label Red is reached
- UPPAAL: Restricted version of CTL is used
  - AF, AG, EF, EG operators at the beginning of the formula

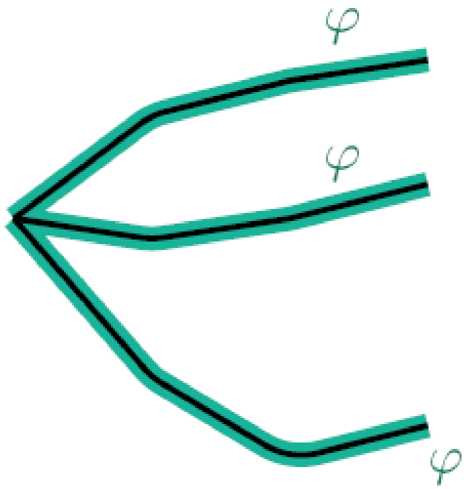
Summary of temporal operators in UPPAAL

Operator	Informal semantics	UPPAAL notation
AG φ	For all paths, for all states φ	A[] φ
AF φ	For all paths, for a state eventually φ	A<> φ
EG φ	For at least one path, for all states φ	E[] φ
EF φ	For at least one path, for a state eventually φ	E<> φ
AG(φ => AF ψ)	After φ always ψ	φ --> ψ
	There is no deadlock	AG not deadlock

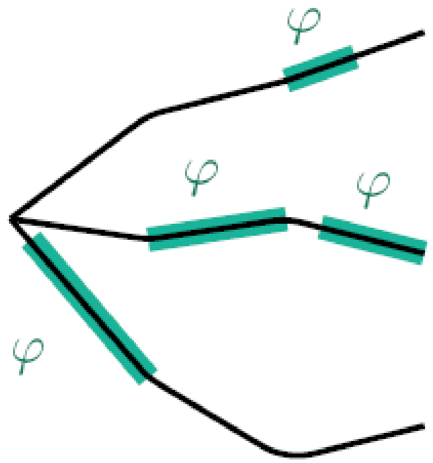
φ and ψ are Boolean expressions on clocks, variables and location names

Composite operators for all paths

AG φ



AF φ

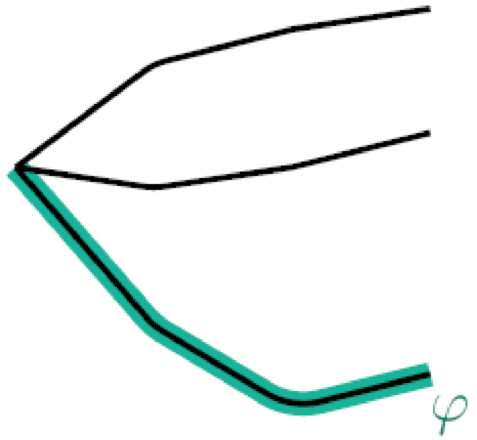


AG φ: For all paths,
for all states φ is true

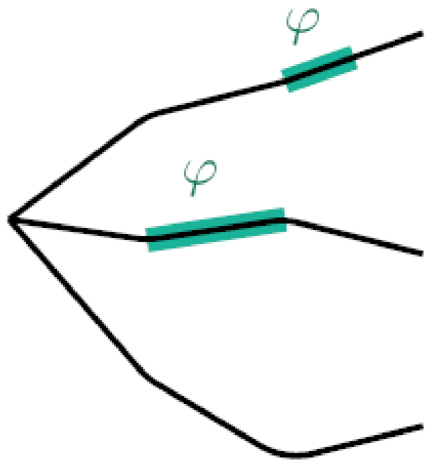
AF φ: For all paths,
for a state eventually φ
becomes true

Composite operators for at least one path

EG φ



EF φ



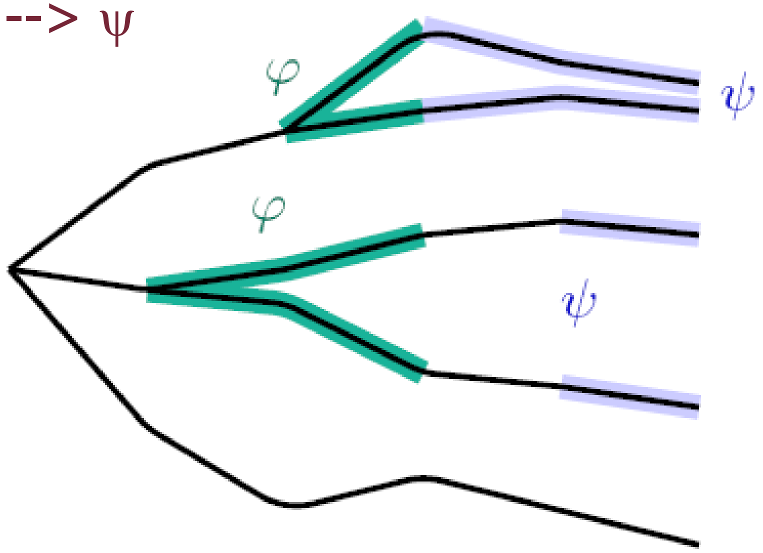
EG φ: There is at least one path, where for all states φ is true

EF φ: There is at least one path, where eventually φ becomes true

- Is there a relation between AG and EF?
- Is there a relation between AF and EG?

Conditional reachability

φ --> ψ



- AG(φ => AF ψ) ≡ φ --> ψ
  For all paths, for all states: if φ is true then it implies that on all paths eventually a state occurs in which ψ becomes true
- Reachability with a timing condition: φ --> (ψ and x <= t)
  where x is a clock variable that is reset when φ becomes true

Examples: formalizing properties using temporal logic

Let us consider an air-conditioner

- States are characterized using the following local properties: {Switched-off, Switched-on, Faulty, Cooling, Heating, Ventilating}

To formalize requirements:

- The local properties can be used in the requirements
- In a state several local properties may hold
- The reachability properties are defined considering behaviour from the initial state of the system
- The behaviour of the air-conditioner may not be known when the properties are formalized

Examples: formalizing properties using temporal logic

States of the air-conditioner are characterized using propositions:

{Switched-off, Switched-on, Faulty, Cooling, Heating, Ventilating}

Examples for formalized properties:

- The air-conditioner shall not perform cooling and heating at the same time:

  $AG\ (\neg(\text{Cooling} \wedge \text{Heating}))$

- The ventilating mode shall eventually be turned on:

  $AF\ (\text{Ventilating})$

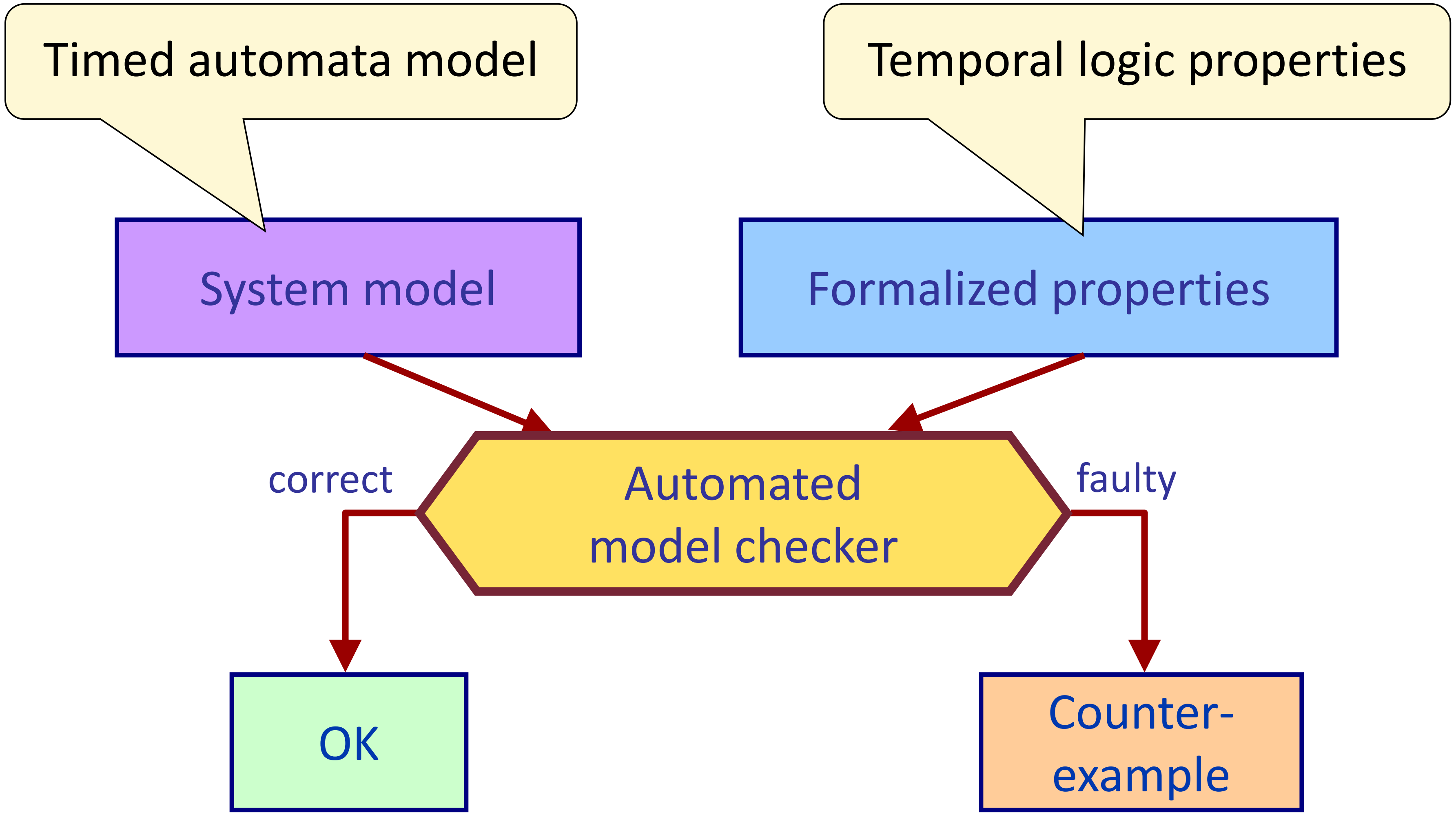
- The air-conditioner can be operated (being switched on) in such a way that it does not perform cooling:

  $EG\ (\text{Switched-on} \wedge (\neg\ \text{Cooling}))$

- If the air-conditioner is faulty then it shall eventually be switched off:

  AG(Faulty => AF (Switched-off)) or Faulty --> Switched-off

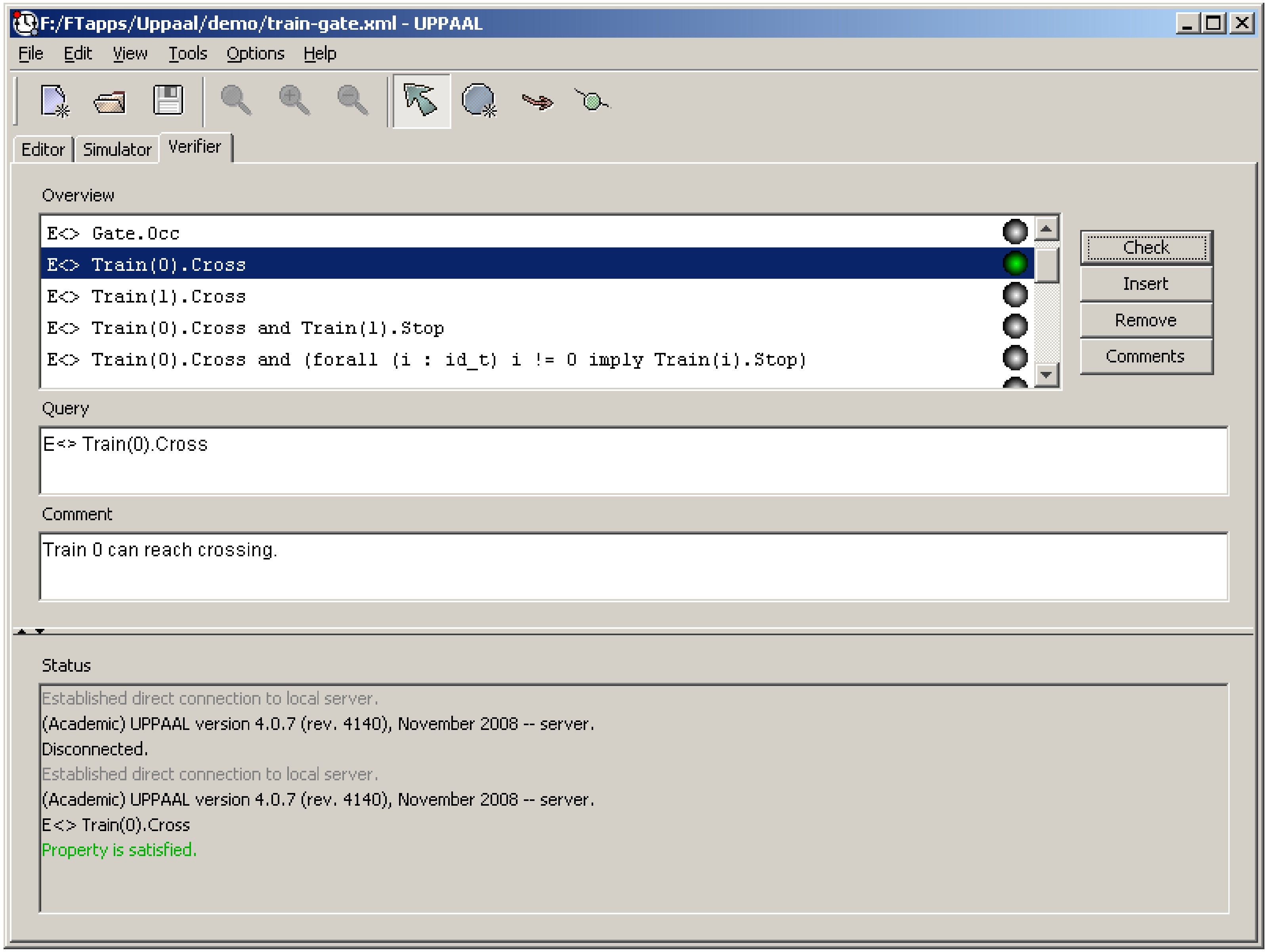
Model checking



The UPPAAL model checker

- Properties can be formalized using temporal logic
  - Verification of the properties is automated
- Verification is performed by an exhaustive exploration of the state space of the model
  - Breadth-first, or depth-first search can be configured
- Diagnostic trace can be generated
  - Counter-example (for safety properties) or witness (for liveness properties)
  - Shortest, fastest, or some (any) diagnostic trace can be configured
  - The diagnostic trace can be loaded into the simulator to investigate and debug the behaviour

The UPPAAL model checker


F:/FTapps/Uppaal/demo/train-gate.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator Verifier
Overview
Check
Insert
Remove
Comments
Query
Comment
Status

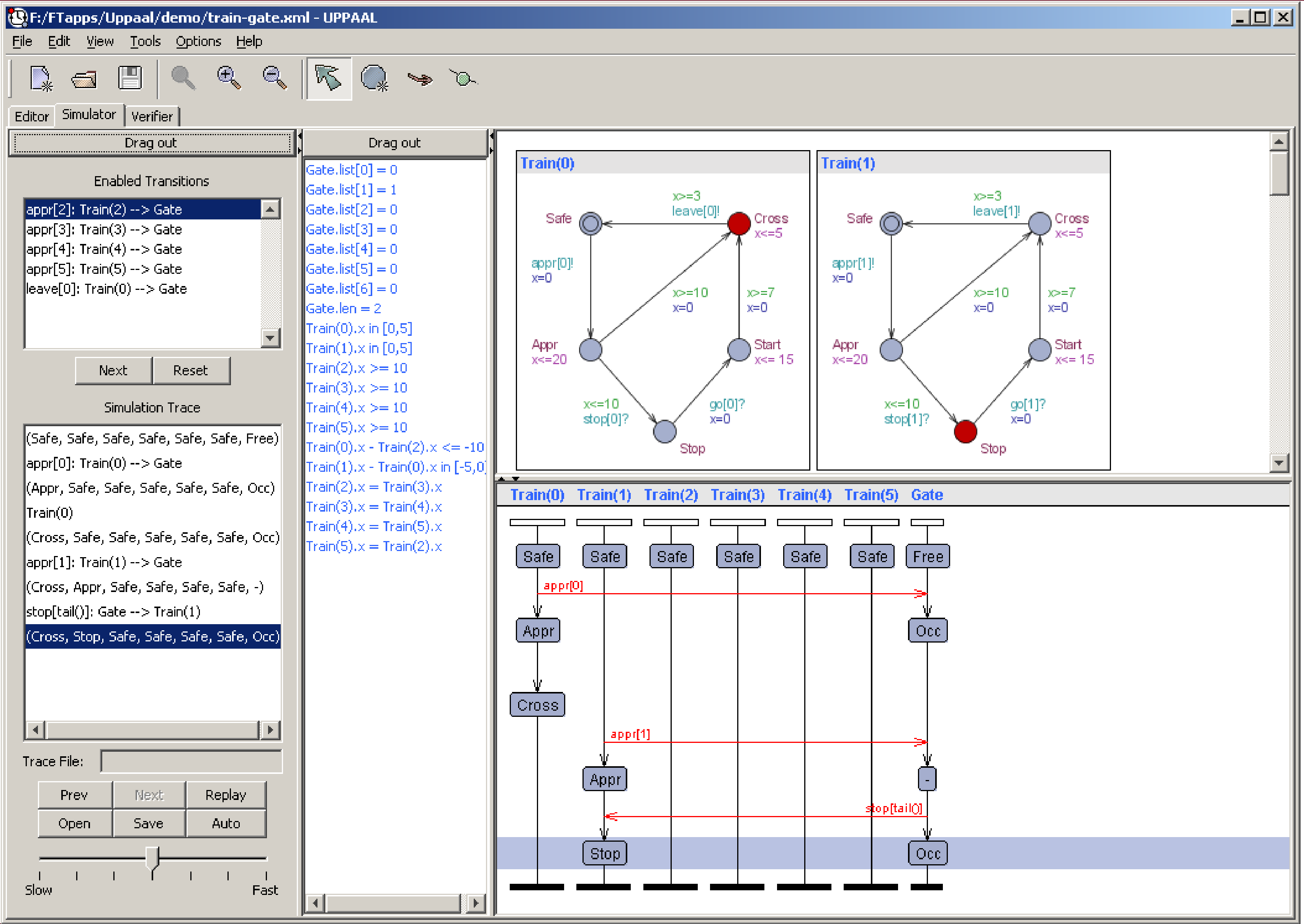
```
E<> Gate.Occc
E<> Train(0).Cross
E<> Train(1).Cross
E<> Train(0).Cross and Train(1).Stop
E<> Train(0).Cross and (forall (i : id_t) i != 0 imply Train(i).Stop)
```

Query: E<> Train(0).Cross

Comment: Train 0 can reach crossing.

```
Established direct connection to local server.
(Academic) UPPAAL version 4.0.7 (rev. 4140), November 2008 -- server.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.7 (rev. 4140), November 2008 -- server.
E<> Train(0).Cross
Property is satisfied.
```

Counter-example in the simulator



F:/FTapps/Uppaal/demo/train-gate.xml - UPPAAL

Enabled Transitions

appr[2]: Train(2) --> Gate
appr[3]: Train(3) --> Gate
appr[4]: Train(4) --> Gate
appr[5]: Train(5) --> Gate
leave[0]: Train(0) --> Gate

Simulation Trace

(Safe, Safe, Safe, Safe, Safe, Safe, Free)
appr[0]: Train(0) --> Gate
(Appr, Safe, Safe, Safe, Safe, Safe, Occ)
Train(0)
(Cross, Safe, Safe, Safe, Safe, Safe, Occ)
appr[1]: Train(1) --> Gate
(Cross, Appr, Safe, Safe, Safe, Safe, -)
stop[tail()]: Gate --> Train(1)
(Cross, Stop, Safe, Safe, Safe, Safe, Occ)

```
Gate.list[0] = 0
Gate.list[1] = 1
Gate.list[2] = 0
Gate.list[3] = 0
Gate.list[4] = 0
Gate.list[5] = 0
Gate.list[6] = 0
Gate.len = 2
Train(0).x in [0,5]
Train(1).x in [0,5]
Train(2).x >= 10
Train(3).x >= 10
Train(4).x >= 10
Train(5).x >= 10
Train(0).x - Train(2).x <= -10
Train(1).x - Train(0).x in [-5,0]
Train(2).x = Train(3).x
Train(3).x = Train(4).x
Train(4).x = Train(5).x
Train(5).x = Train(2).x
```

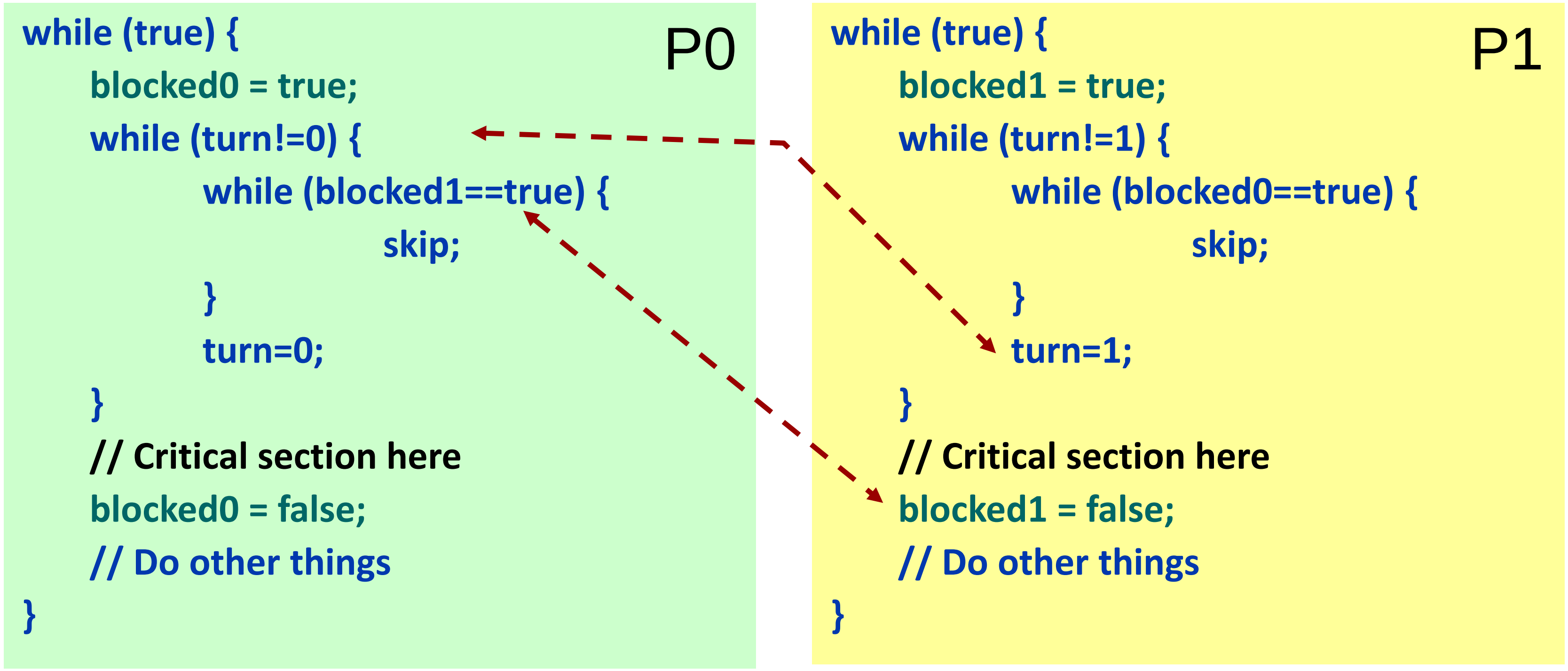
A case study

An engineering task

- Let us consider a concurrent (multi-process) system
- At most one process is allowed to access a shared resource at a time: mutual exclusion is required
    - Example: Use of communication channel as resource
    - Access to resource: “Critical sections” in the programs; at most one process is allowed to be in critical section
    - The platform (OS, framework) does not give support: no semaphore, no monitor, etc.
    - Only shared variables can be used (atomic reading/writing)
- How to do it?
    - Classical solutions (Peterson, Lamport, Fischer etc.)
    - Custom algorithm

A solution for the mutual exclusion problem

- 2 processes, 3 shared variables (H. Hyman, 1966)
  - **blocked0**: The first process (P0) wants to enter the critical section
  - **blocked1**: The second process (P1) wants to enter the critical section
  - **turn**: Which process will enter (P0 in case of 0, P1 in case of 1)



P0

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section here
    blocked0 = false;
    // Do other things
}
```

P1

```
while (true) {
    blocked1 = true;
    while (turn!=1) {
        while (blocked0==true) {
            skip;
        }
        turn=1;
    }
    // Critical section here
    blocked1 = false;
    // Do other things
}
```

Is this algorithm correct?

Properties to be verified

- Mutual exclusion:
  - At most one process can be in the critical section (it shall never happen that two processes are there)
- It is possible to enter the critical section:
  - P0 is able to enter the critical section
  - P1 is able to enter the critical section
- There is no starvation:
  - P0 will eventually enter the critical section on all paths
  - P1 will eventually enter the critical section on all paths
- Freedom from deadlock:
  - The two processes shall not stop executing

The model in UPPAAL (first version)

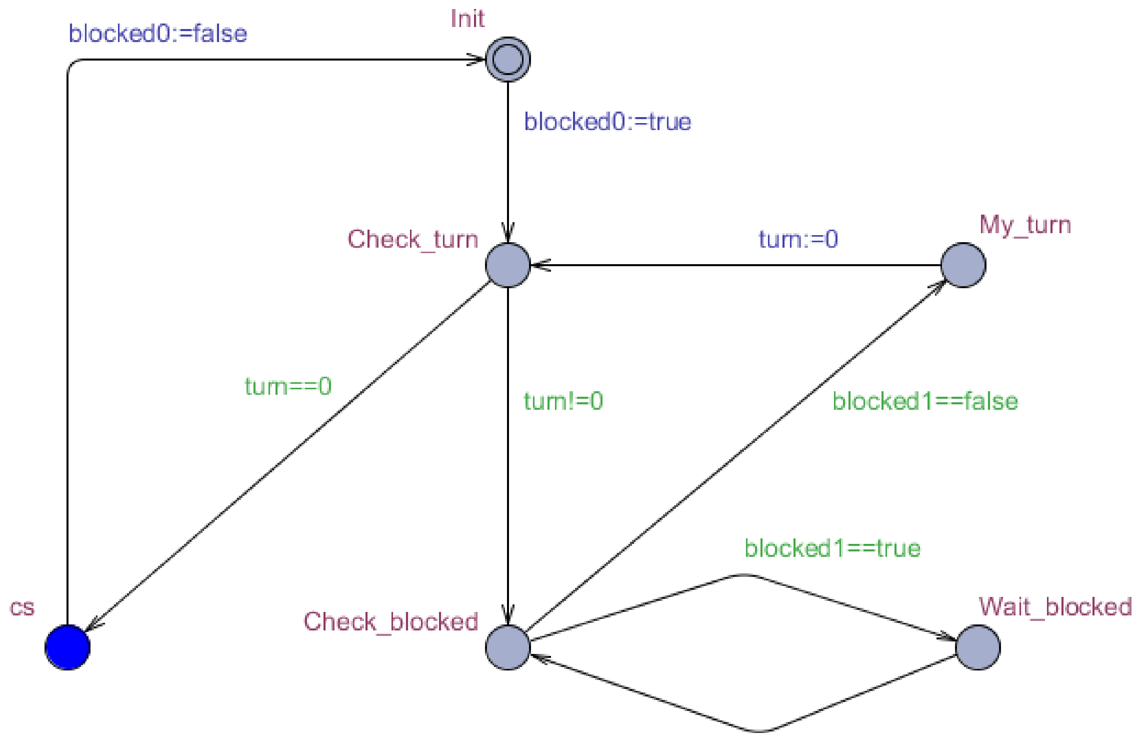
Declarations:

```
bool blocked0;
bool blocked1;
int[0,1] turn=0;
system P0, P1;
```

Modeling techniques used:

- Global declaration of shared variables
- Limiting the range of variables

The P0 automata:



P0

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

The model in UPPAAL (second version)

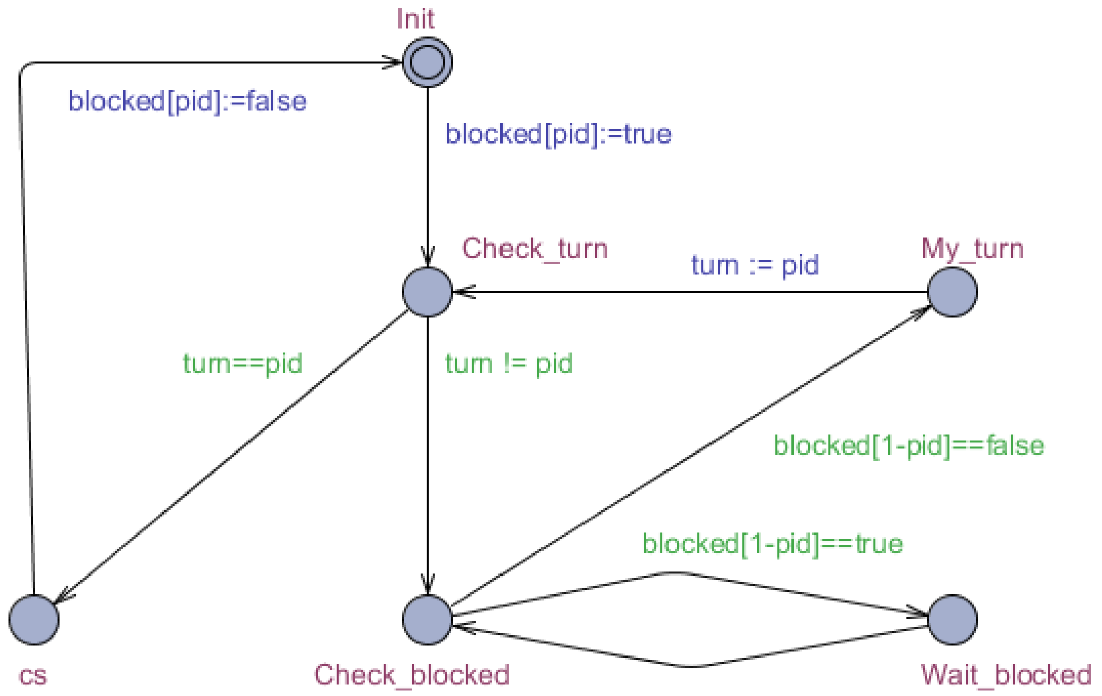
Declarations:

```
int[0,1] blocked[2];
int[0,1] turn;
P0 = P(0);
P1 = P(1);
system P0,P1;
```

Modeling techniques used:

- Global declaration of shared variables
- Limiting the range of variables
- The processes are instantiated using the same template
- Instantiation with parameters (here: pid)
- Using arrays for variables (here: blocked)

The P template with pid parameter:



P0

```
while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

Formalizing properties in UPPAAL

- Mutual exclusion:
  - Only one process may enter the critical section at the same time: A[] not (P0.cs and P1.cs)
- Freedom from deadlock:
  - The two processes shall not stop executing: A[] not deadlock
- It is possible to enter the critical section:
  - P0 is able to enter the critical section: E<>(P0.cs)
  - P1 is able to enter the critical section: E<>(P1.cs)
- There is no starvation:
  - P0 will eventually enter the critical section on all paths: A<>(P0.cs)
  - P0 will eventually enter the critical section on all paths: A<>(P1.cs)

Verifying the properties in UPPAAL

- There is no deadlock
- It is possible to enter the critical section
  - Each process is able to enter the critical section
- The mutual exclusion property is not satisfied!
  - The model checker produces a diagnostic trace (counter-example): There is a specific interleaved behavior in which both processes are in the critical section at the same time
  - The counter-example can be investigated in the simulator
- Starvation cannot be checked without modelling time-dependent behavior
  - Trivial counter-examples may include “waiting forever” in any state
  - Modifying the model: Urgent states (if valid)
  - Here: there is still a cyclic behavior that results in starvation

Correction of the mutual exclusion

New algorithm by Peterson

- For process P0
  (for P1 it is similar):

```
Hyman:

while (true) {
    blocked0 = true;
    while (turn!=0) {
        while (blocked1==true) {
            skip;
        }
        turn=0;
    }
    // Critical section
    blocked0 = false;
    // Do other things
}
```

```
Peterson:

while (true) {
    blocked0 = true;
    turn=1;
    while (blocked1==true &&
            turn!=0) {
                skip;
    }

    // Critical section
    blocked0 = false;
    // Do other things
}
```

Summary: Properties of model checking

- Advantages:
  - It offers a complete exploration of the state space of the model
  - It is possible to check huge state spaces (using compact representation)
    - 10^{20}, or even 10^{100} states can be checked automatically (in specific cases)
  - There are fully automated tools, there is no need to perform manual adjustment, mathematical operations, or heuristics
  - Diagnostic trace is generated, which supports debugging and correction
- Problems:
  - Scalability is limited (state space must fit into memory)
  - Effective for control-oriented models
    - Complex data structures result in huge state space
  - It is not easy to generalize the results
    - If a protocol is correct for 2 processes, is it correct for N processes as well?
  - The formalization of properties is difficult
    - There are different „temporal logic languages”