Formal modelling and verification

István Majzik slides

Modified by András Vörös on 15. October 2013

Budapest University of Technology and Economics

Dept. of Measurement and Information Systems

Ariane-5 Flight 501

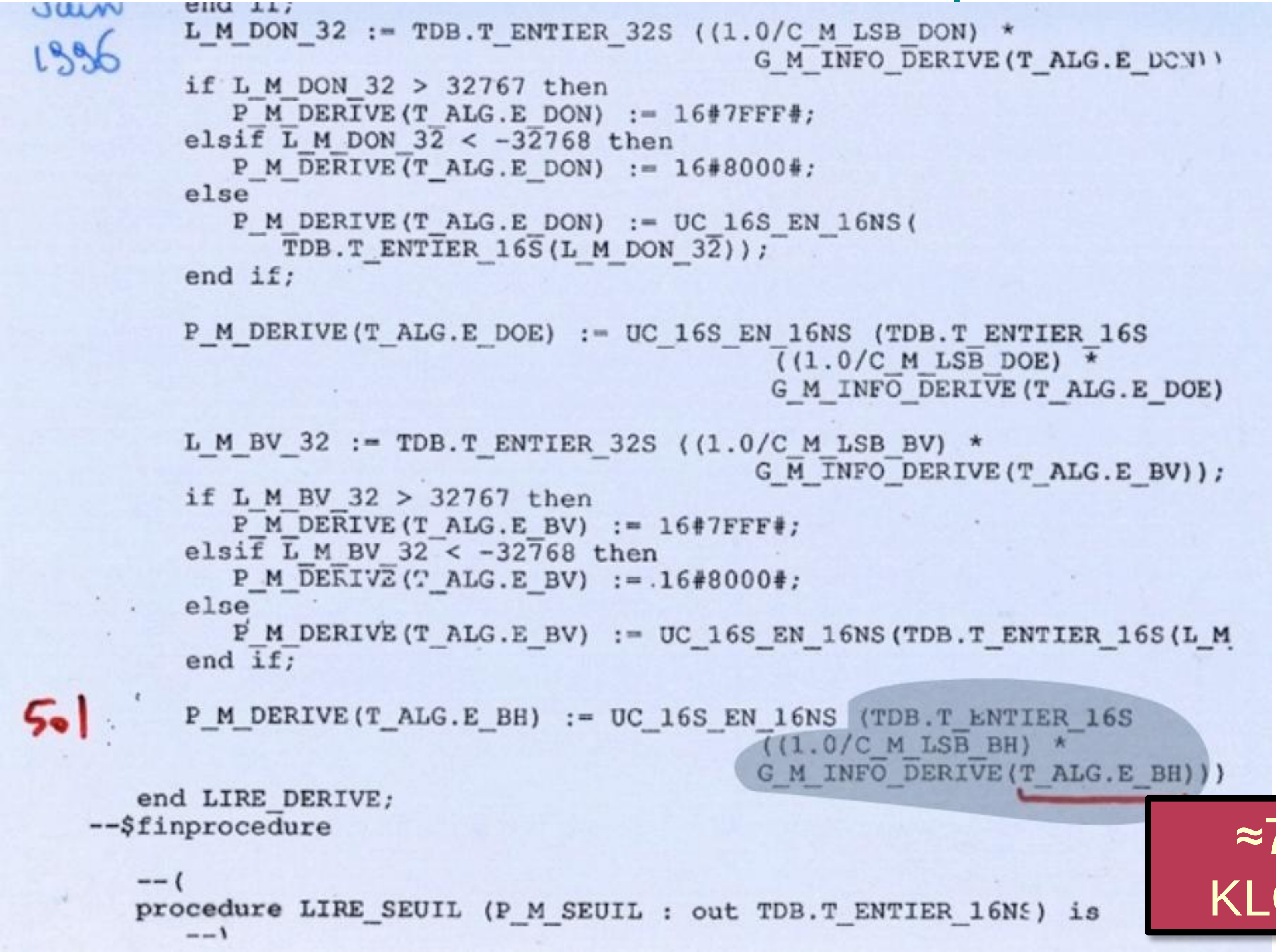
Ariane-5 Flight 501

- **Ariane-5** first test flight (4. June 1996.)
  - http://www.youtube.com/watch?v=c9Hf4qTxdxs

Ariane-5 Flight 501

- **Ariane-5** first test flight (4. June 1996.)
  - http://www.youtube.com/watch?v=c9Hf4qTxdxs
  - Exploded after 40 s (self destruction)
  - damage: at least US$ 370,000,000 (others say: 8 billion $)
  - Cause: **software bug** (not found by tests)
  - The control software of Ariane-4 was used
  - The software contains a float-int conversion
  - The acceleration of Ariane-5 was higher than the acceleration of Ariane-4
  - The acceleration value stored in a float variable of 64 bit could not be converted into a signed integer variable of 16 bit. This lead to unhandled exception

Ariane-5 Ada code example



Juin
1996

```
      end if;
      L_M_DON_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_DON) *
                                         G_M_INFO_DERIVE(T_ALG.E_DON))
      if L_M_DON_32 > 32767 then
         P_M_DERIVE(T_ALG.E_DON) := 16#7FFF#;
      elsif L_M_DON_32 < -32768 then
         P_M_DERIVE(T_ALG.E_DON) := 16#8000#;
      else
         P_M_DERIVE(T_ALG.E_DON) := UC_16S_EN_16NS(
            TDB.T_ENTIER_16S(L_M_DON_32));
      end if;

      P_M_DERIVE(T_ALG.E_DOE) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                          ((1.0/C_M_LSB_DOE) *
                                          G_M_INFO_DERIVE(T_ALG.E_DOE)

      L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
                                         G_M_INFO_DERIVE(T_ALG.E_BV));
      if L_M_BV_32 > 32767 then
         P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
      elsif L_M_BV_32 < -32768 then
         P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
      else
         P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M
      end if;

      P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
                                         ((1.0/C_M_LSB_BH) *
                                         G_M_INFO_DERIVE(T_ALG.E_BH)))

   end LIRE_DERIVE;
--$finprocedure

   --(
   procedure LIRE_SEUIL (P_M_SEUIL : out TDB.T_ENTIER_16NS) is
      --)
```

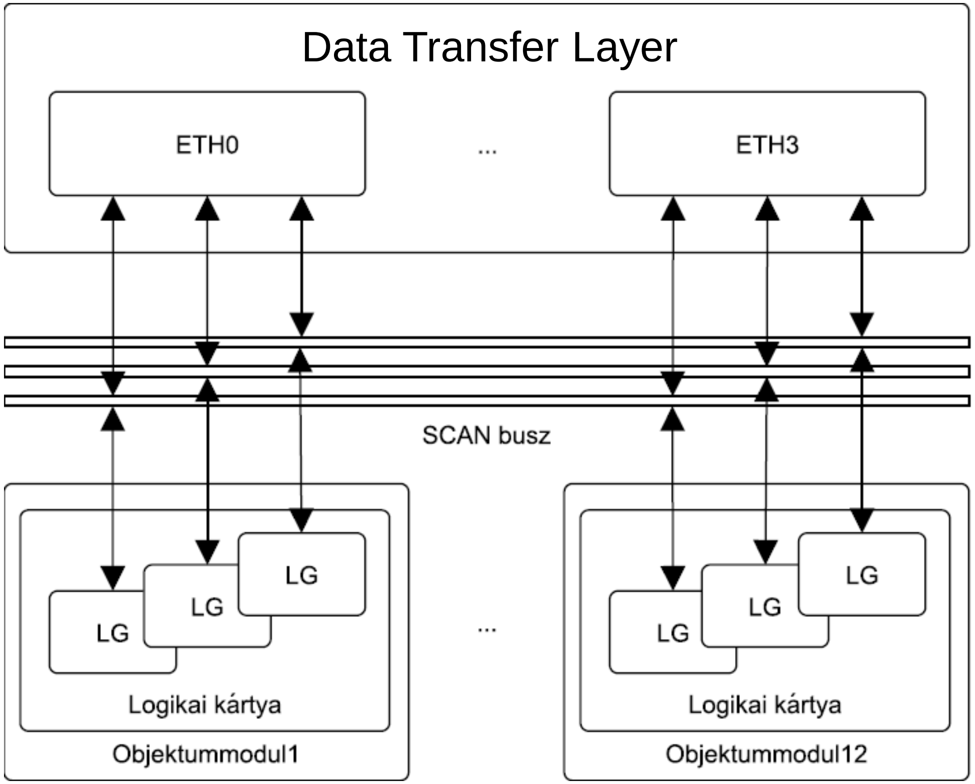
501

≈70 KLOC

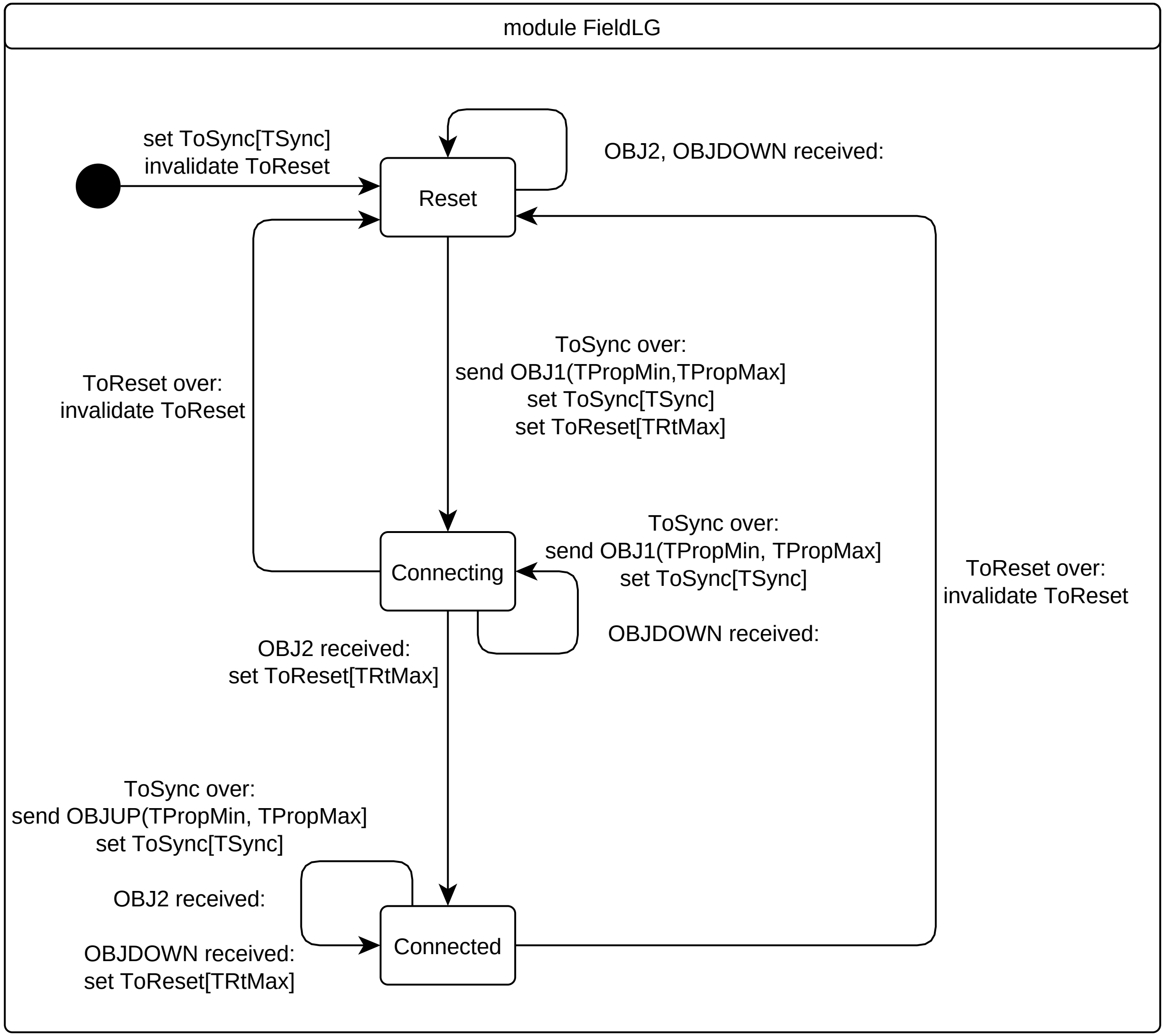
Case study: ProSigma SCAN protokoll

- A real time safety critical (SIL 4) protocol developed by Prolan Zrt.
  - Function: ensuring fault-free communication

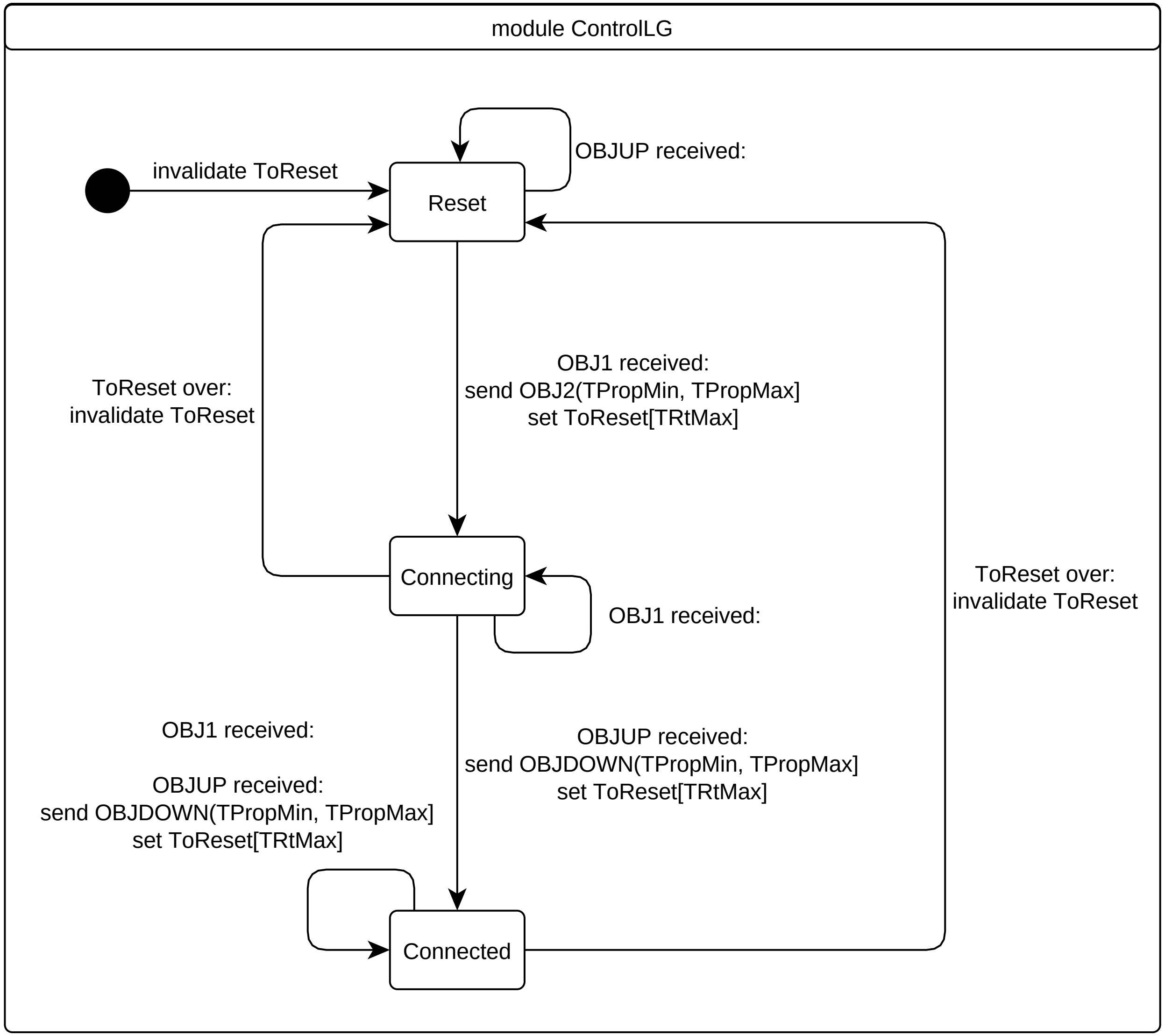
- Analysed function: connection handling
  - Establishing connection
  - Sending object state
- Checked property:
  - The connection remains established after a while
  - A liveness property



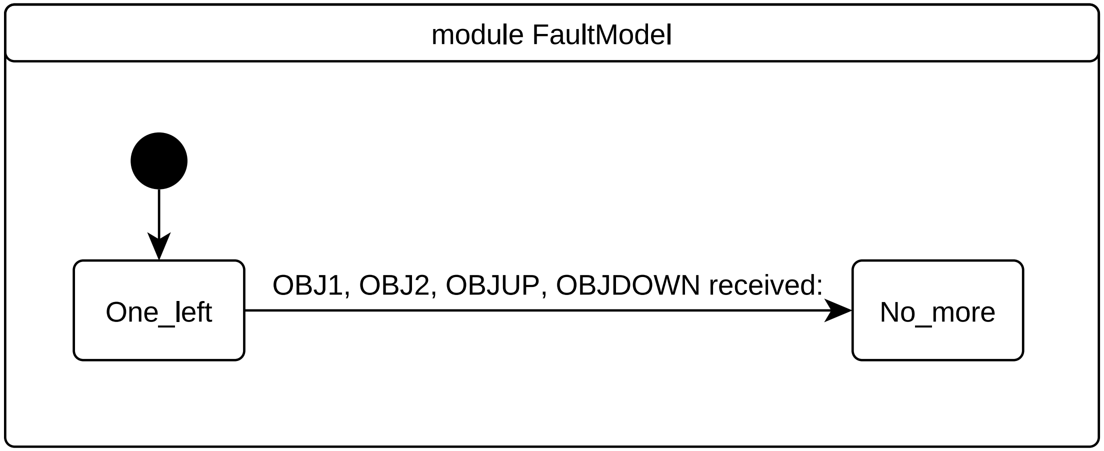
Case study: Field LG



Case study: Control LG

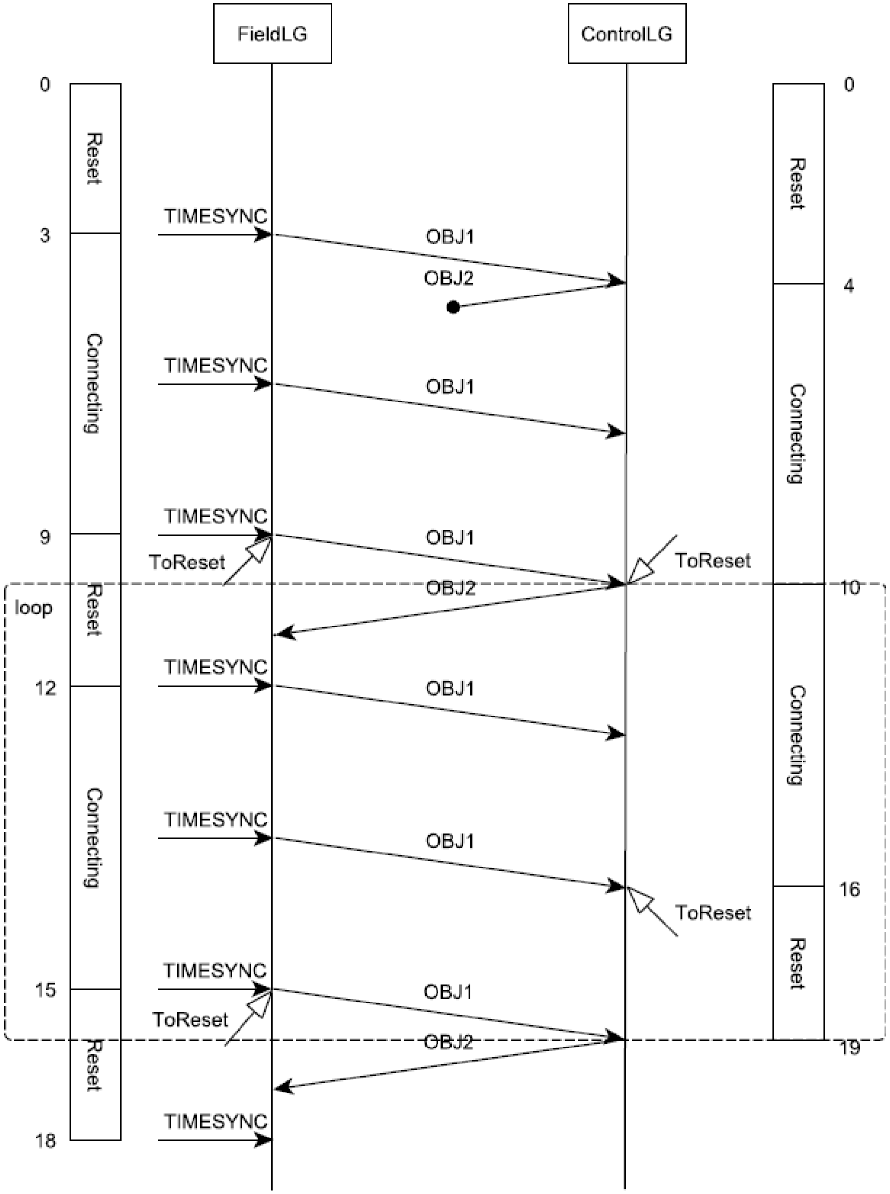


Case study: Fault model

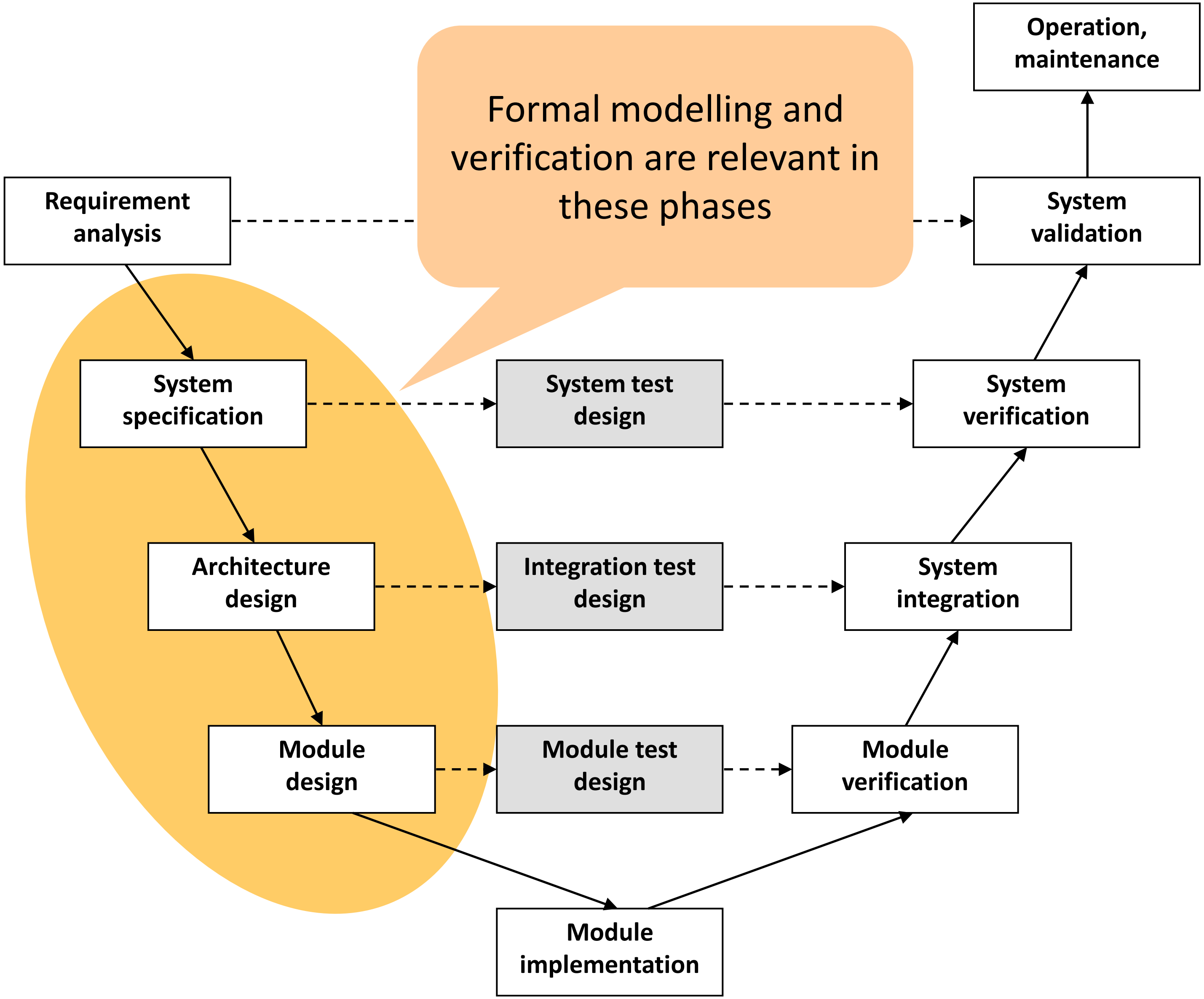


Case study: results

- An unexpected counterexample for the correct behavior
  - The loss of a single message can cause the protocol to be stuck in a bad state
- Suggestion to correct the specification
- Proving the correctness of the modified system



Example software lifecycle (V-model)



Formal modelling and verification are relevant in these phases

Techniques and measures in standards

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

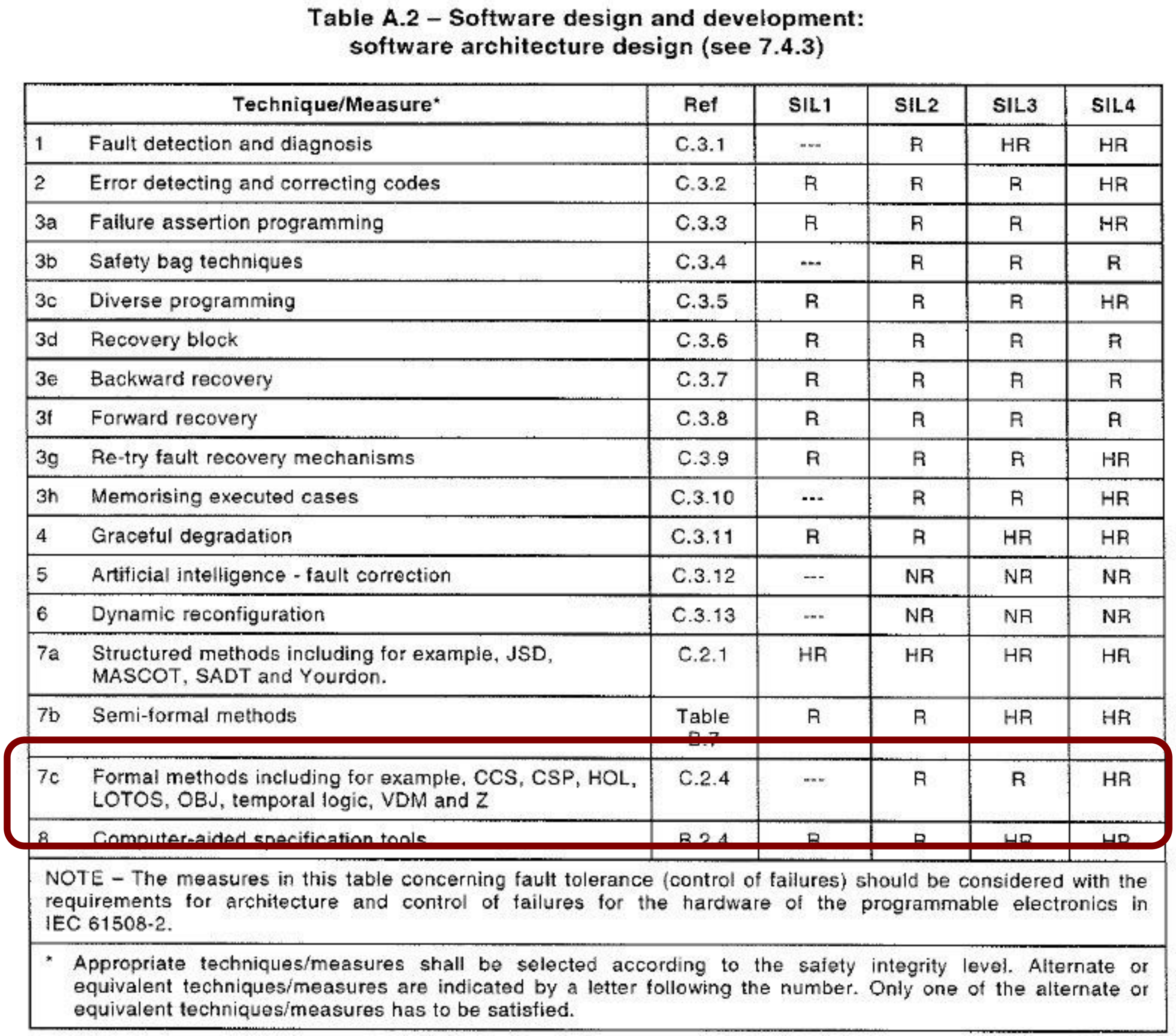


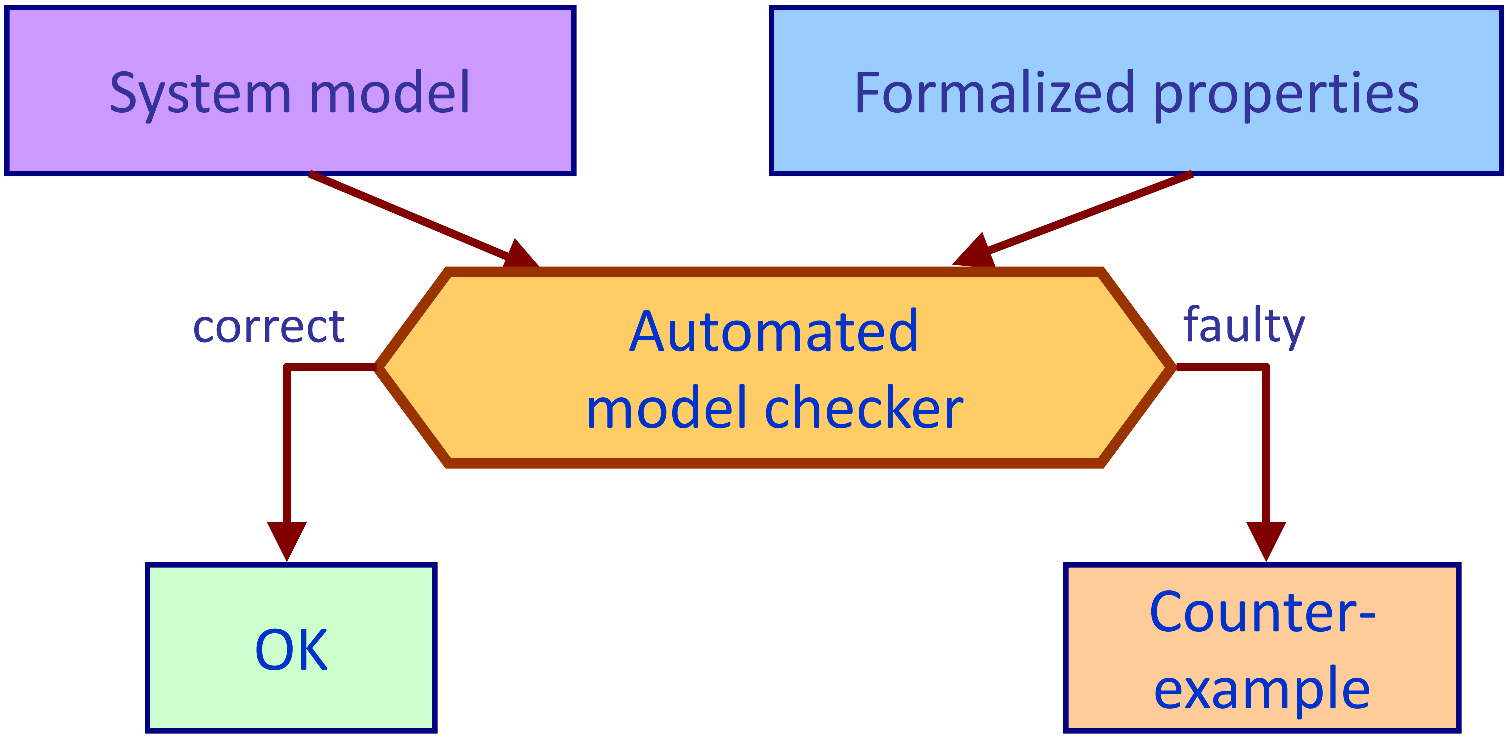
Table A.2 – Software design and development: software architecture design (see 7.4.3)

	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

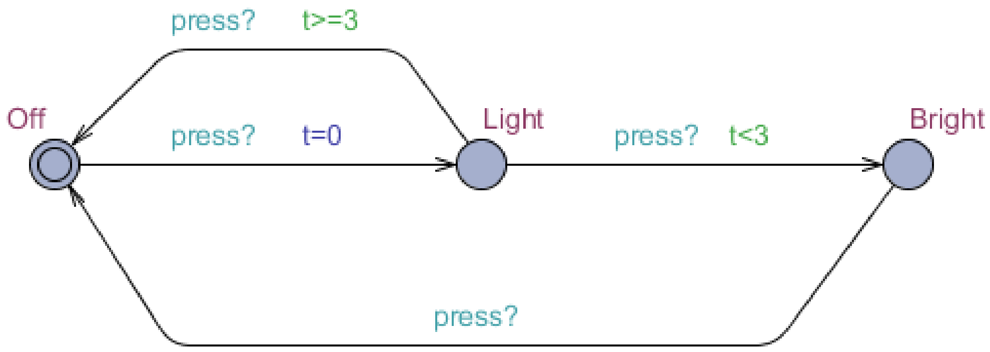
NOTE – The measures in this table concerning fault tolerance (control of failures) should be considered with the requirements for architecture and control of failures for the hardware of the programmable electronics in IEC 61508-2.

* 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

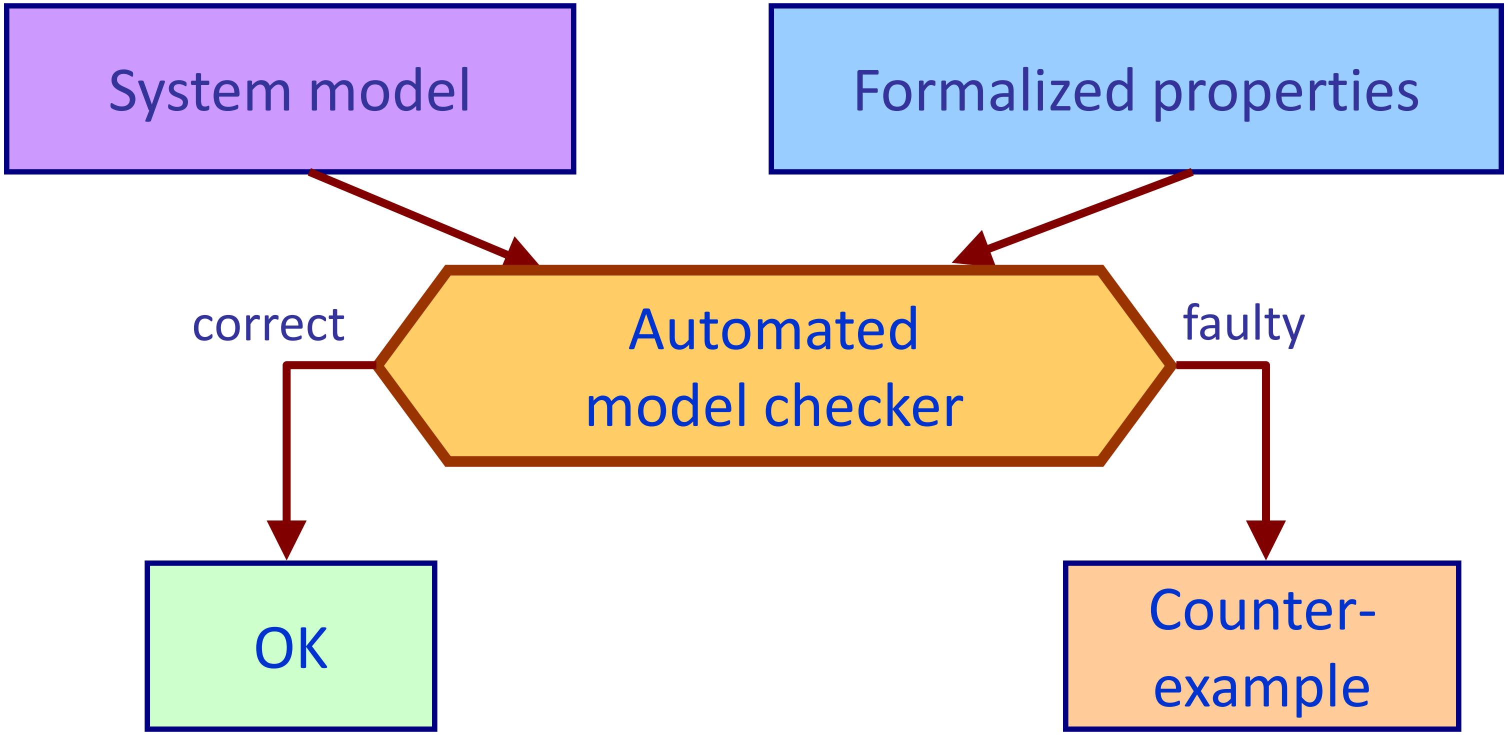


Modeling with timed automata



Goals of formal modeling and verification

- Modeling with timed automata
- Mapping to timed automata from higher-level models (e.g., from UML state machines)



System model

Formalized properties

Automated model checker

correct

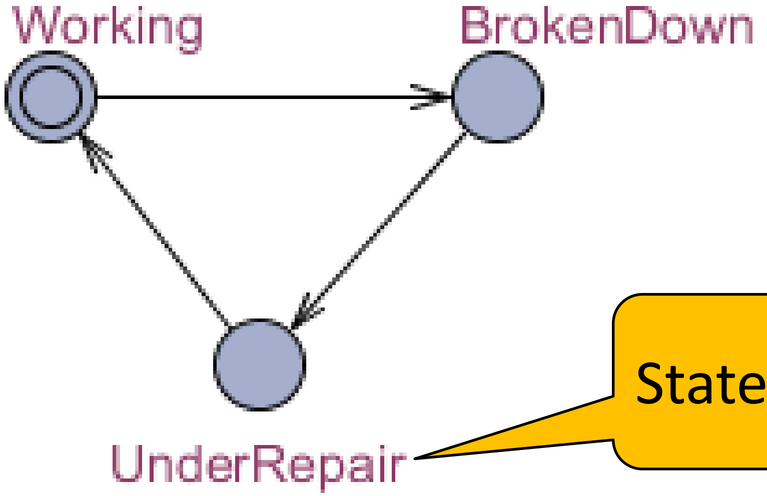
faulty

OK

Counter-example

Automata and variables

- Goal: Modeling event driven, state based behaviour
- Basic formalism: Finite state machine (FSM)
  - States (with state names)
  - State transitions



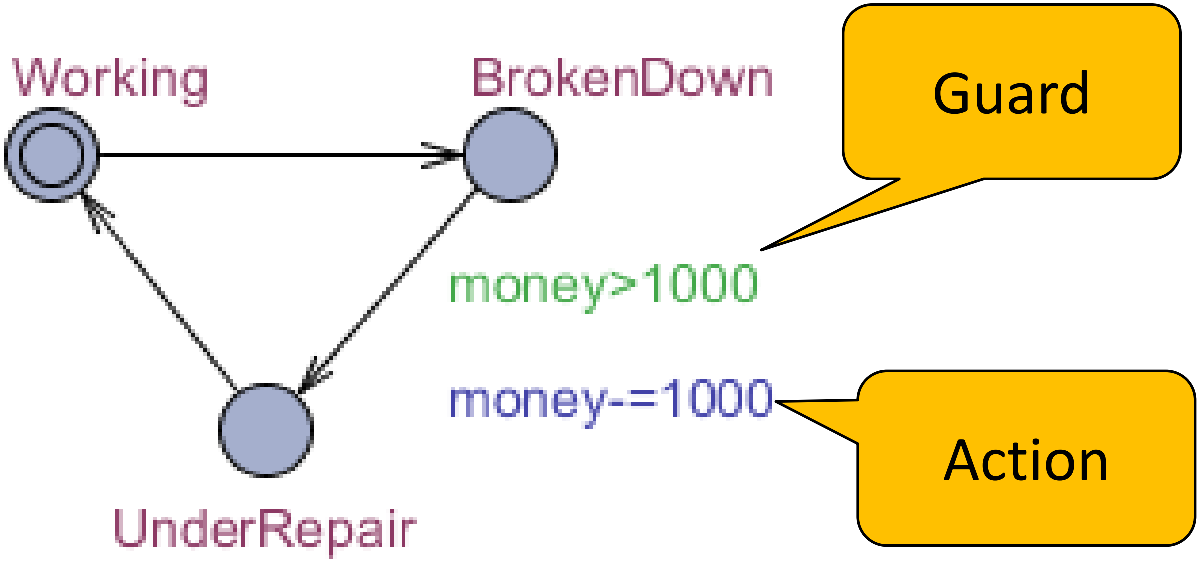
State name

Automata and variables

- Goal: Modeling event driven, state based behaviour
- Basic formalism: Finite state machine (FSM)
  - States (with state names)
  - State transitions
- Extension: Using integer variables
  - Range of potential values can be specified
  - Constants can be defined
  - Integer arithmetic can be used
- Extensions on state transitions:
  - Guards: Predicates on the variables
    - It shall be true in order to enable the state transition
  - Actions: Assignments to the variables

Automata and variables

- Goal: Modeling event driven, state based behaviour
- Basic formalism: Finite state machine (FSM)
- Extension: Using integer variables
- Extensions on state transitions

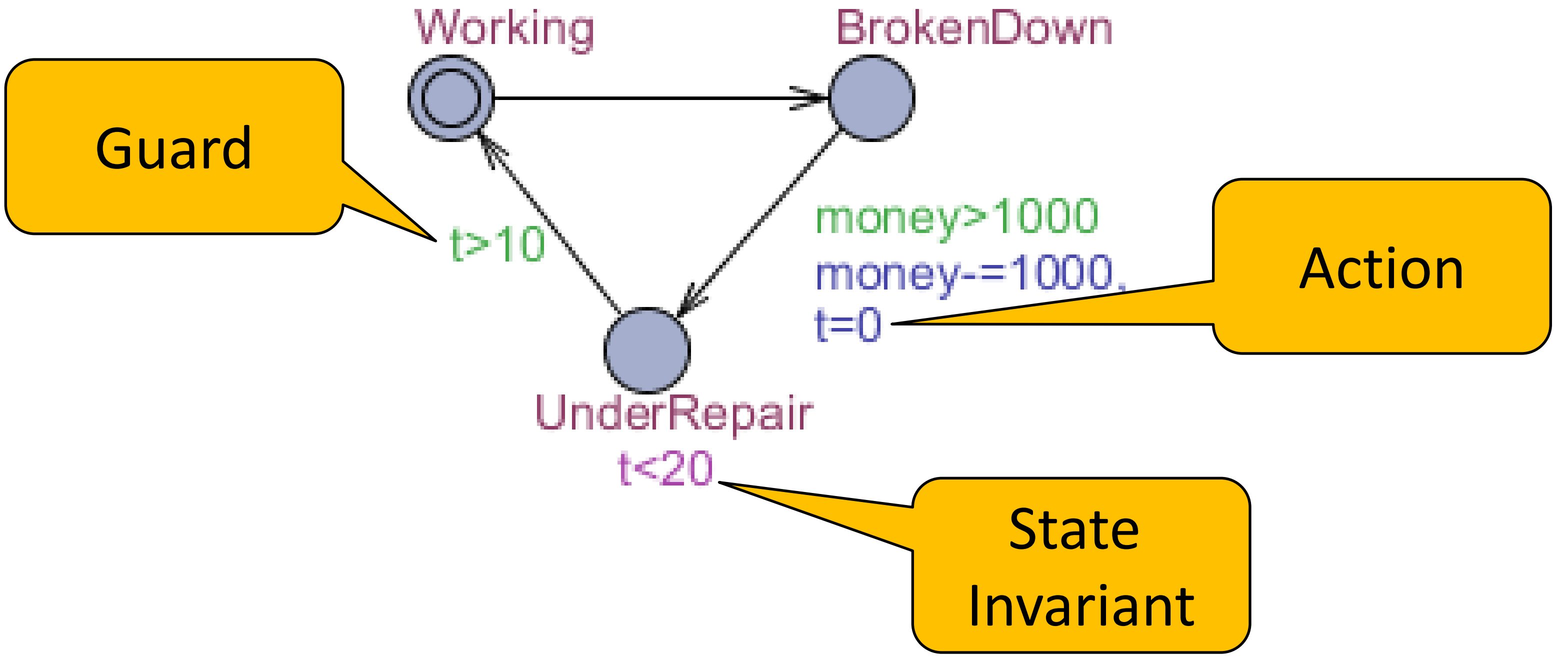


Extensions using clock variables

- Goal: Modelling time dependent behaviour
  - Time elapses in the states
  - Behaviour depends on the time spent in the state
  - To be verified: States that can be reached after/until a given time
- Modelling extension: Clock variables
  - Concurrent clocks (timers) having the same rate
  - Relative time measurements (e.g., time-out): Resetting and reading clock variables
- Usage in state transitions:
  - Actions: Resetting clock variables, independently
  - Guards: Referring to clock variables and constants
- Usage in states:
  - State invariants: The validity of the state is specified using predicates on clock variables and constants

Extensions using clock variables

- Goal: Modelling time dependent behaviour
- Modelling extension: Clock variables
- Usage in state transitions
- Usage in states

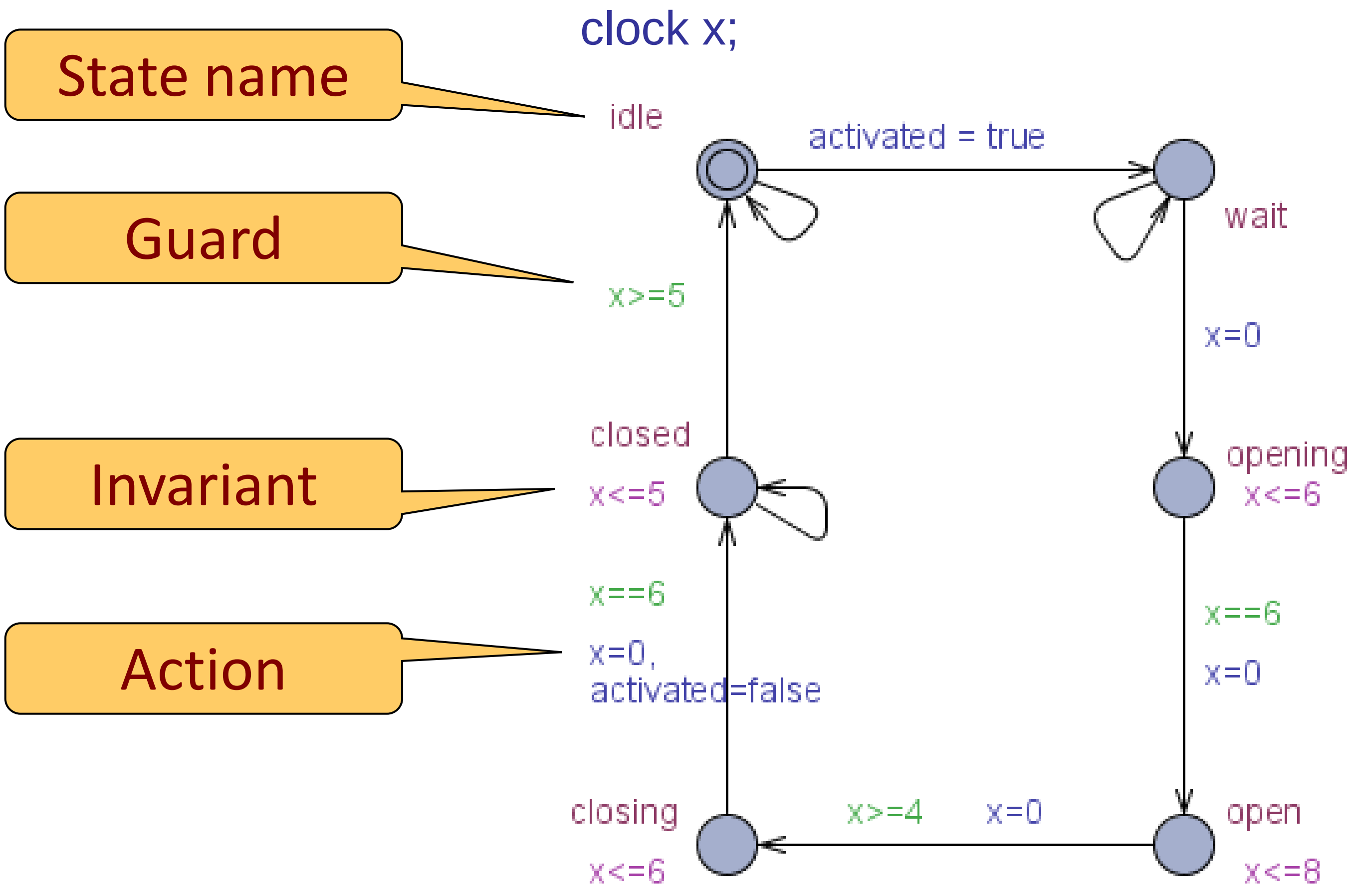


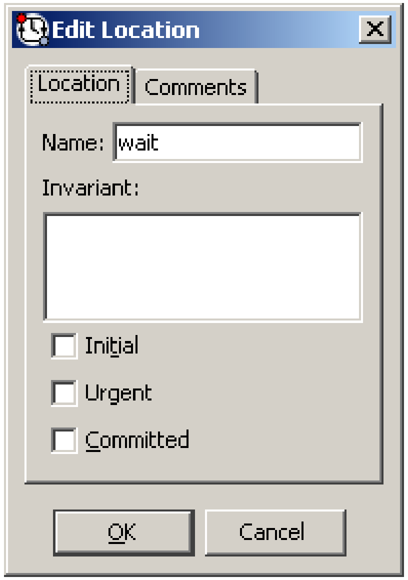
Guard

Action

State Invariant

Timed automata (in the UPPAAL tool)





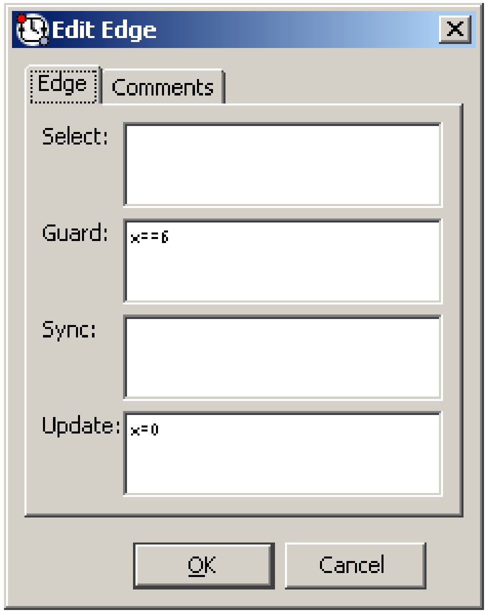
Name: wait

Invariant:

☐ Initial

☐ Urgent

☐ Committed



Select:

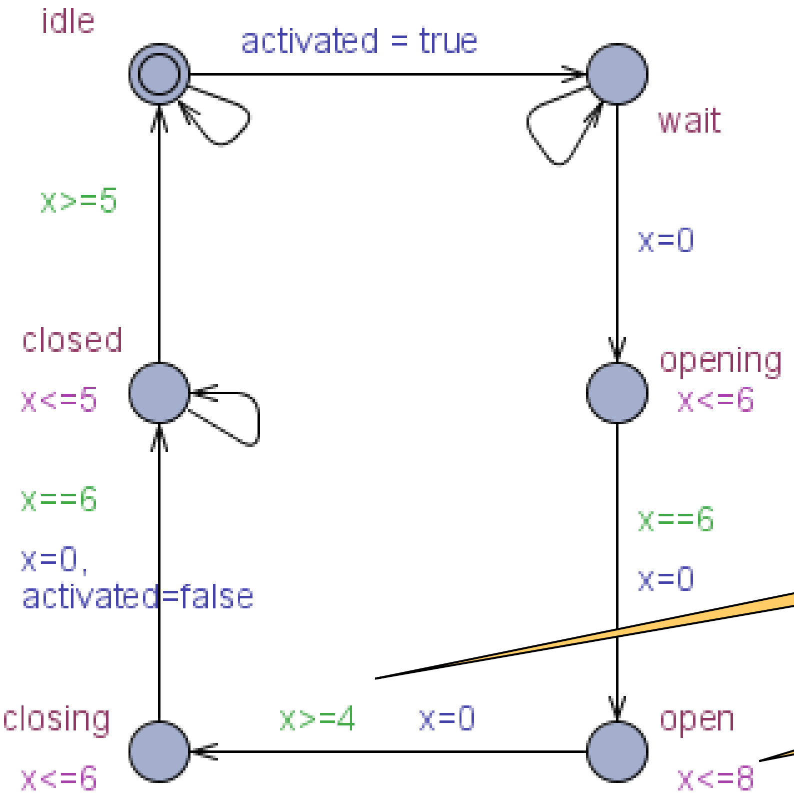
Guard: x==6

Sync:

Update: x=0

Role of state invariants and guards

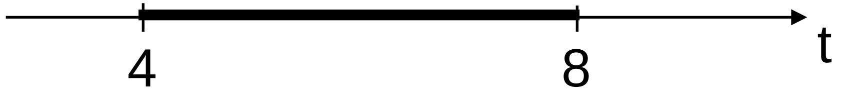
clock x;



Guard

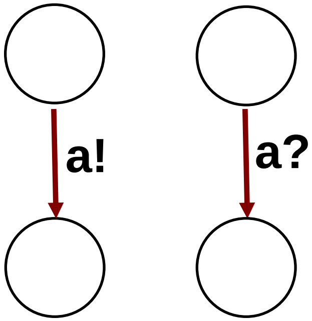
Invariant

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



Extensions for modeling distributed systems

- Goal: Modeling networks of interacting automata
  - Synchronization among automata
  - Synchronized state transitions (rendezvous): synchronous communication
    - Sending and receiving of messages at the same time
    - This primitive can be used also to model asynchronous communication
- Extension: Synchronized actions
  - Channels are defined (synchronous channels)
  - Message sending: ! operator on the channel
    Message receiving: ? operator on the channel
    - E.g., on the channel a the actions are a! and a?



chan a

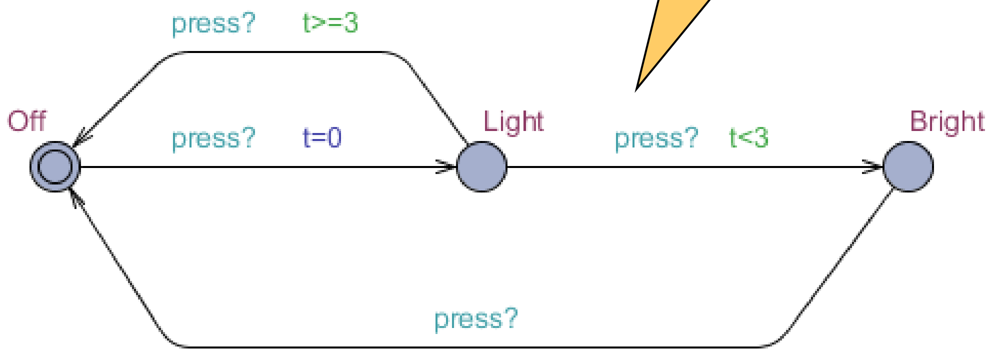
- Parameterization
  - Automata with parameters: Instantiation of templates
    - E.g., Door(bool &id) with id as a parameter
  - Channel arrays (indexed)
    - E.g., a[id] is a channel indexed by the value of variable id

Example: Using clock variables and synchronization

Declarations:

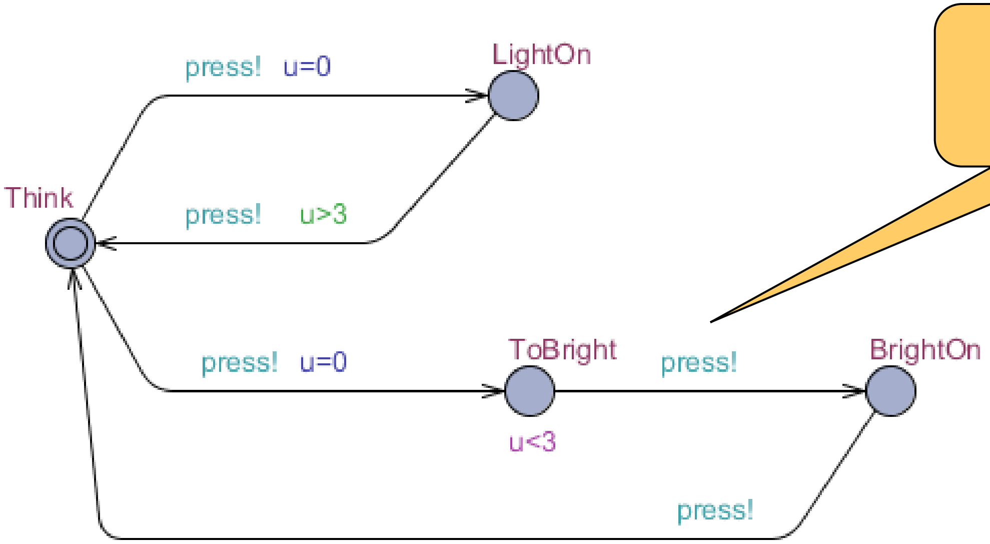
clock t, u;
chan press;

Switch:



"Receiving a message"

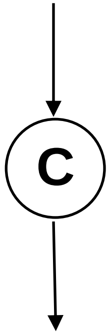
User:



"Sending a message"

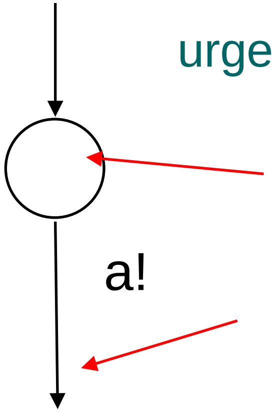
Further extensions

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



- Urgent channel: delay is not allowed
  - Synchronization shall be executed immediately, without delay (but interleaving is possible)
  - No time related guard is allowed on the state transition with an action referring to an urgent channel
  - No state invariant is allowed in a state where there is an outgoing transition with an action referring to an urgent channel

urgent chan a;



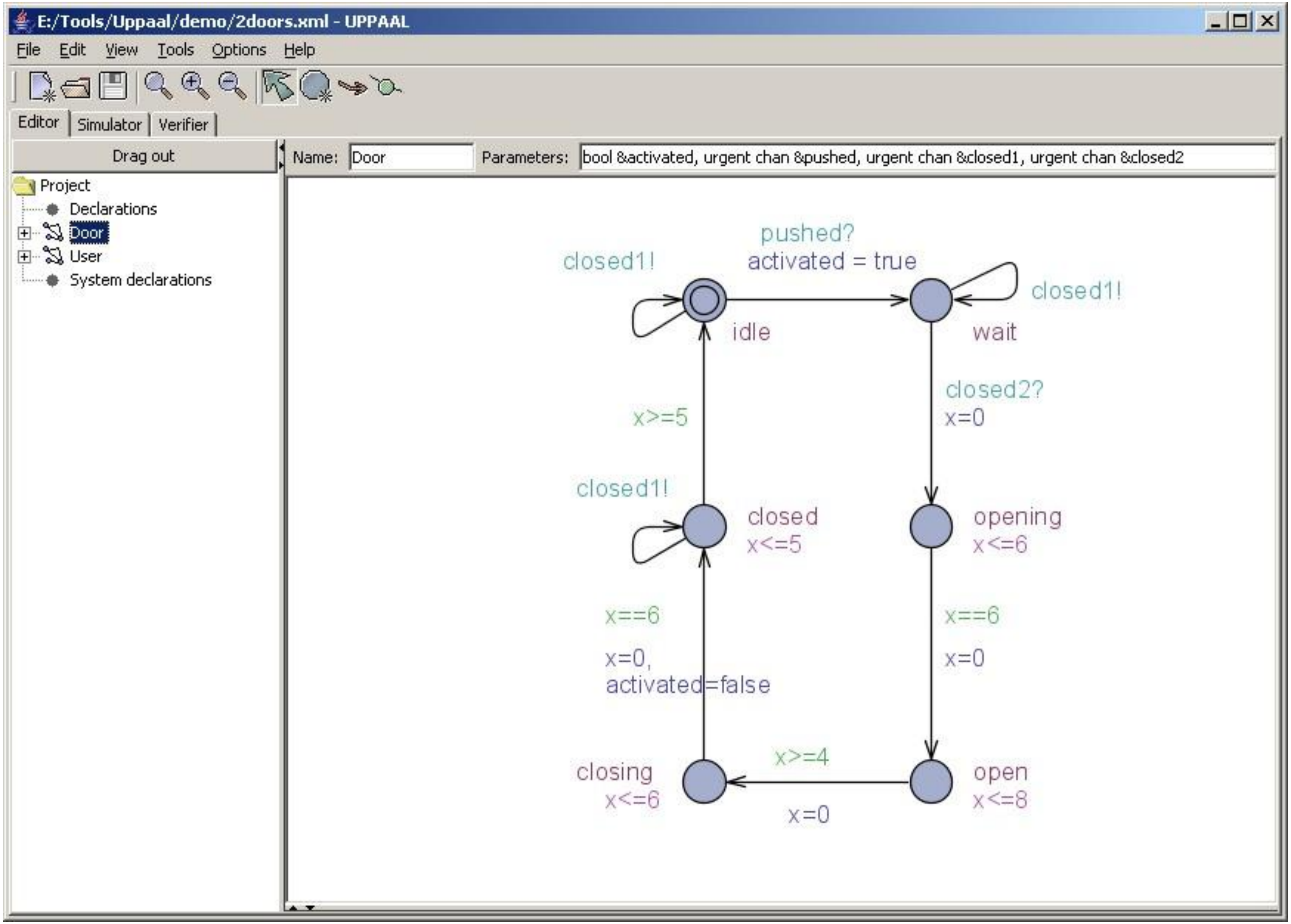
No state invariant is allowed here

No time related guard is allowed here

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/

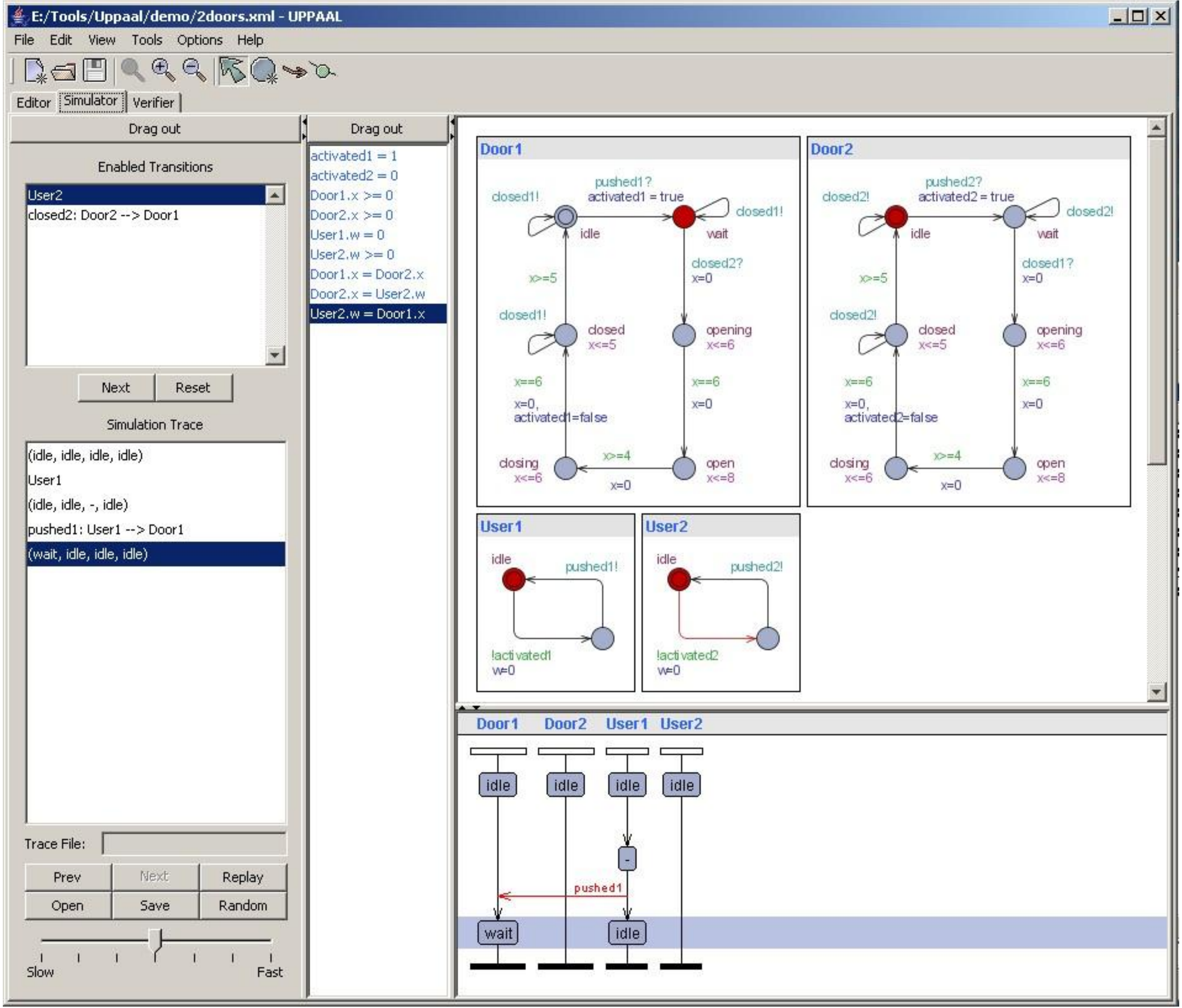
Automaton model



Name: Door

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

Simulator



Enabled Transitions

User2
closed2: Door2 --> Door1

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

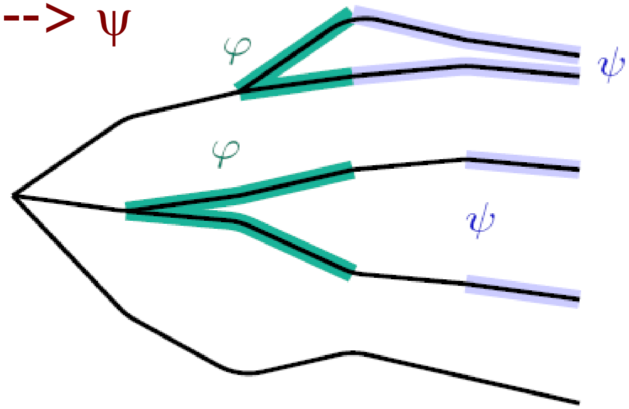
Simulation Trace

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

Trace File:

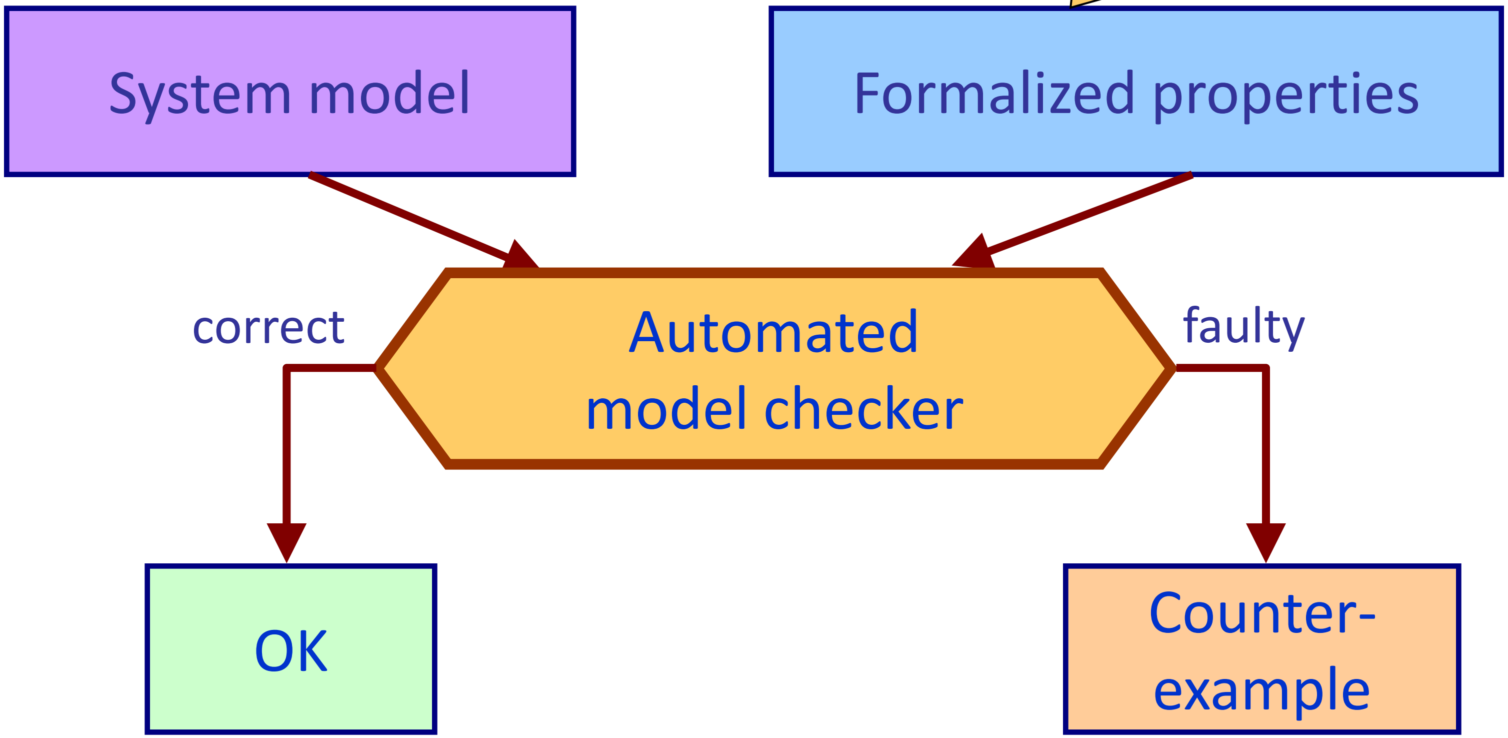
Formalizing requirements with temporal logics

φ --> ψ



Goals of formal modeling and verification

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



System model

Formalized properties

Automated model checker

correct

faulty

OK

Counter-example

What are the formalized properties?

An example to illustrate the properties to be formalized:

- The states 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 of states
  - Evaluation is possible on the state space of the system
  - Example: „Heating shall be followed by ventilating”
  - It can be applied in continuously working systems
  - Typical categories of reachability properties:
    - „Safety” of the system
    - „Liveness” of the system

Safety properties

- Typical use: Specification 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.”
- Invariant properties are specified:
  - „In each reachable state it shall be true that …”
- Examples of IT related safety properties:
  - Mutual exclusion: In each reachable state, only one process shall stay in the critical section
  - Security: In each reachable state only authorized information access is possible

Liveness properties

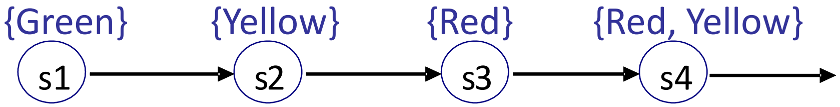
- Typical use: Specification that a desired state is eventually reachable, i.e., something good shall happen
  - „After switch-on, the request should eventually be responded.”
  - „The process shall eventually reach its goal.”
- Existence (reachability) of given state(s) is specified:
  - „A state is eventually reached, in which …”
- Examples of IT related liveness properties:
  - After sending a request the reply shall eventually be received
  - The message that is sent shall eventually be delivered
  - The process shall compute the required result

Language to formalize reachability properties

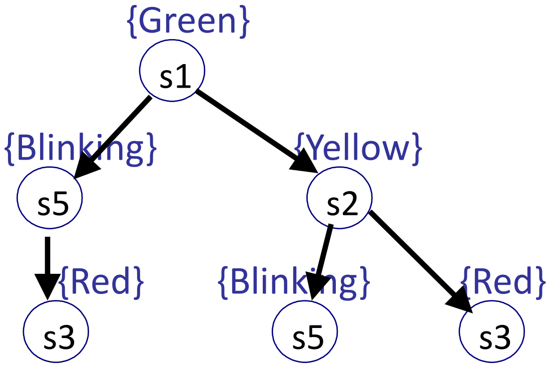
- Reachability: Refers to states that occur each after the other (following each other)
  - The sequence of states in considered as logic time:
    - The present: The current state
    - The next time points: The subsequent states
  - Temporal (ordering in logic time) operators can be defined to express the reachability properties
- Temporal logic:
  - Formal language to express propositions qualified in terms of time
  - Typical temporal operators: „always”, „eventually”, „before”, „until”, „after”, …

Temporal logics

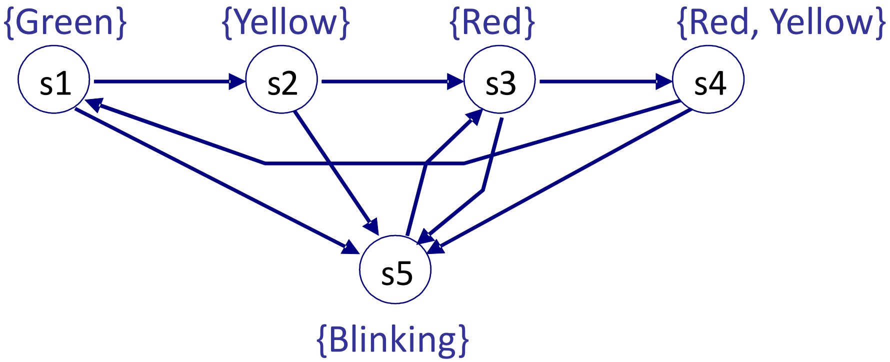
- Linear time:
  The subsequent states form a linear sequence (each state has only one successor)
  → logic time forms a linear timeline



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

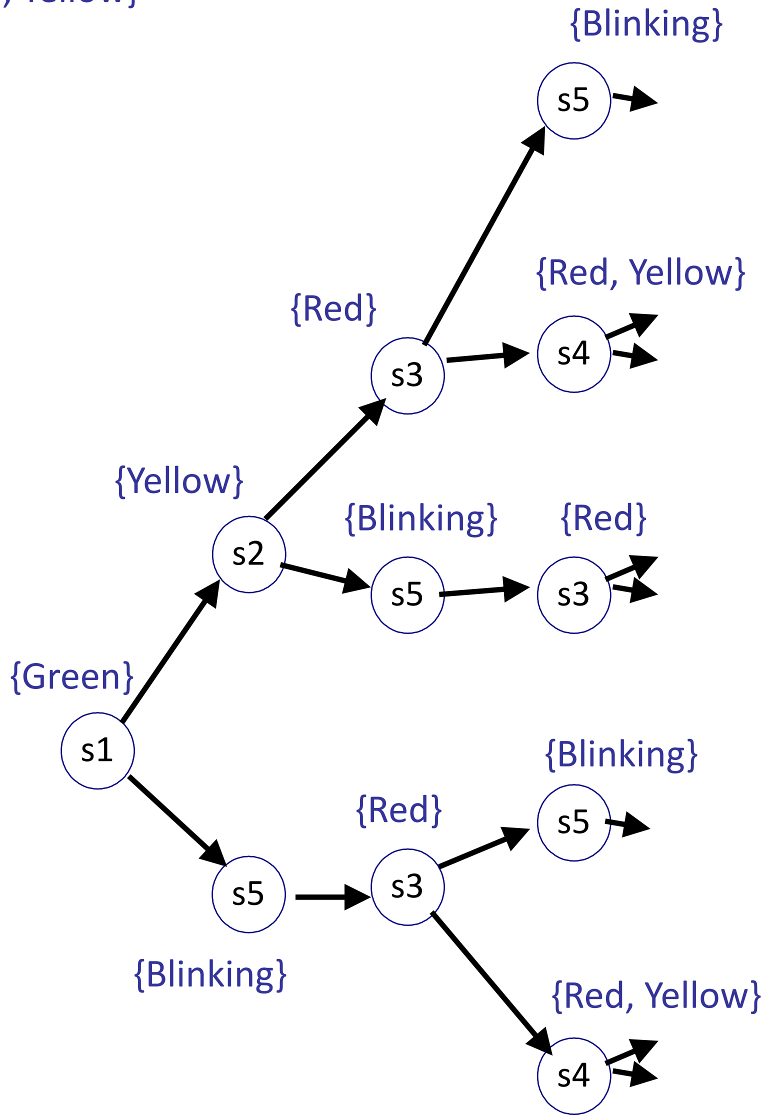


The computational tree

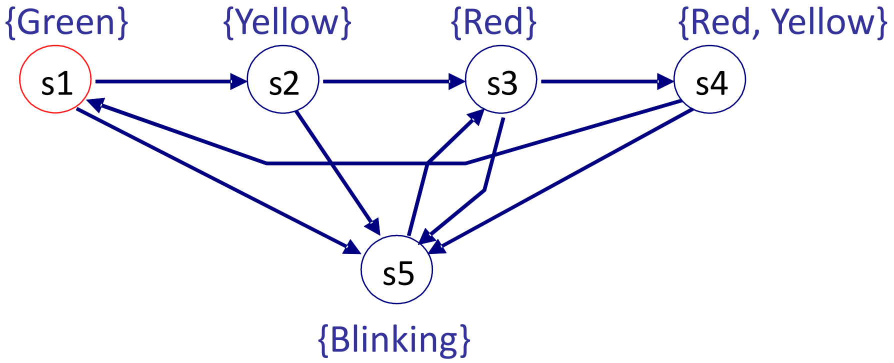


Automaton (FSM) with labelled states ↑

Computational tree: Structure of the potential successor states

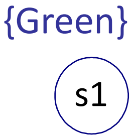


The computational tree

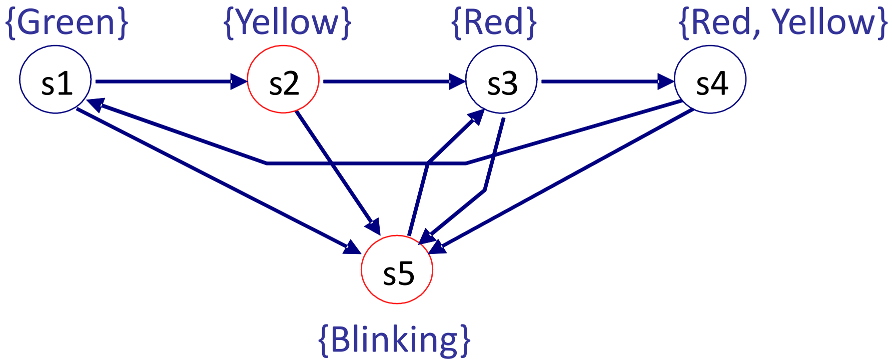


Automaton (FSM) with labelled states ↑

Computational tree: Structure of the potential successor states

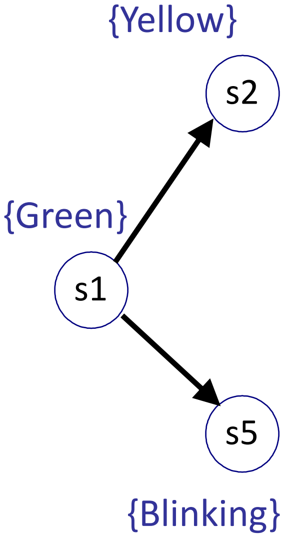


The computational tree

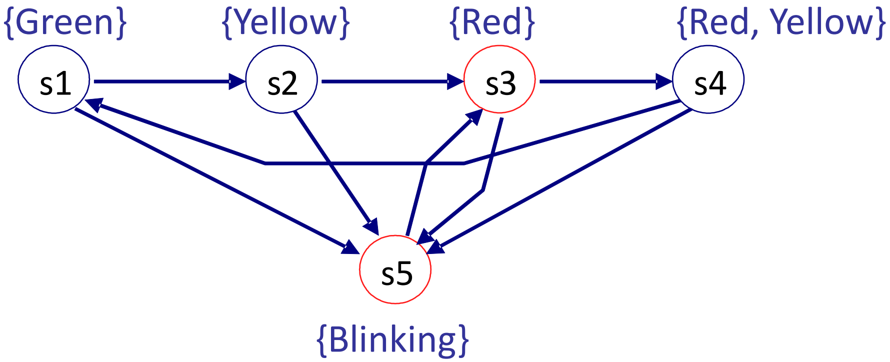


Automaton (FSM) with labelled states ↑

Computational tree: Structure of the potential successor states

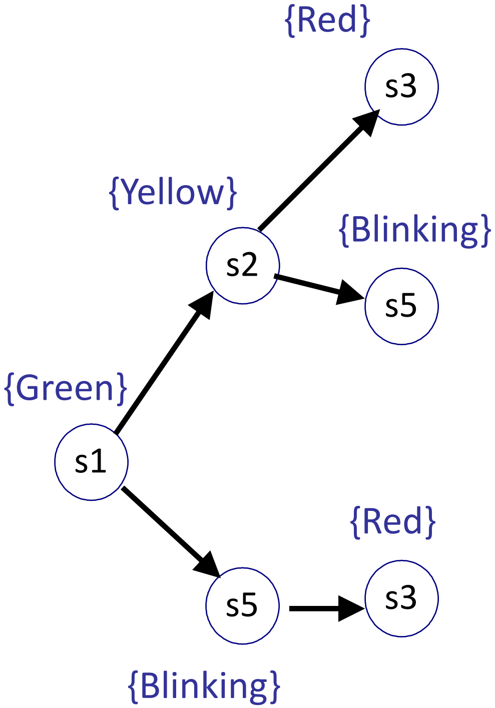


The computational tree

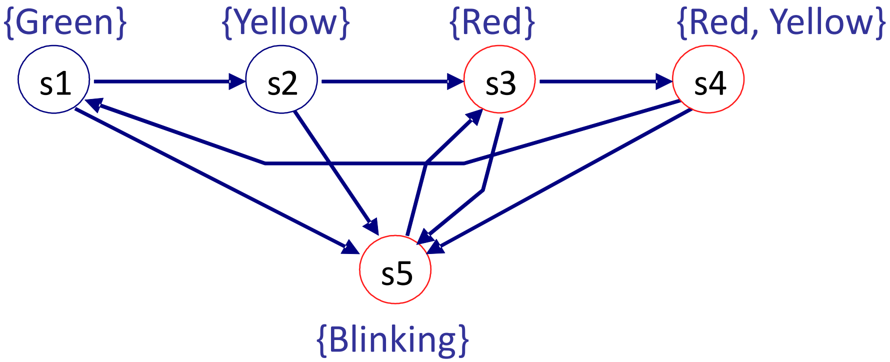


Automaton (FSM) with labelled states ↑

Computational tree: Structure of the potential successor states

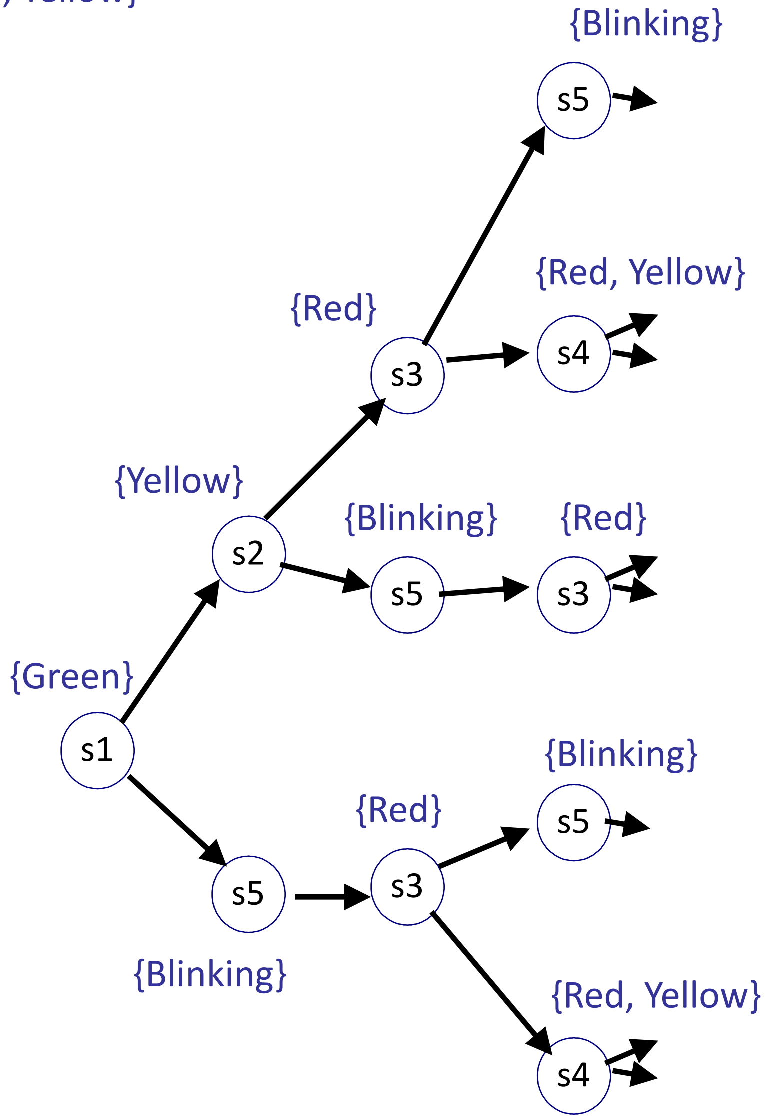


The computational tree



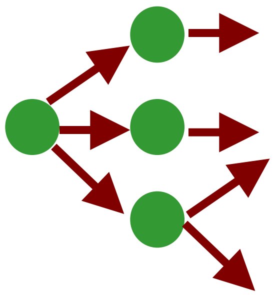
Automaton (FSM) with labelled states ↑

Computational tree: Structure of the potential successor states

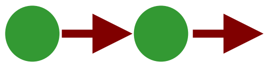


Quantifying paths and characterizing states

- Operators that quantify the paths starting from a given state:
  - A: for all paths from the given state
  - E: for an existing path from the given state



- Operators that characterize states along a given path:



  - F: for a state along the path ("future")
  - G: for all states along the path ("globally")
  - X: for the next state from the initial state of the path ("next")
  - U: for states until reaching a specified state ("until")
    - E.g., Yellow U Red means states 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 an existing 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): There shall exist a path where states are labeled with Yellow until a state with label Red is reached
- Restricted version of CTL is used in UPPAAL
  - AF, AG, EF, EG operators are used

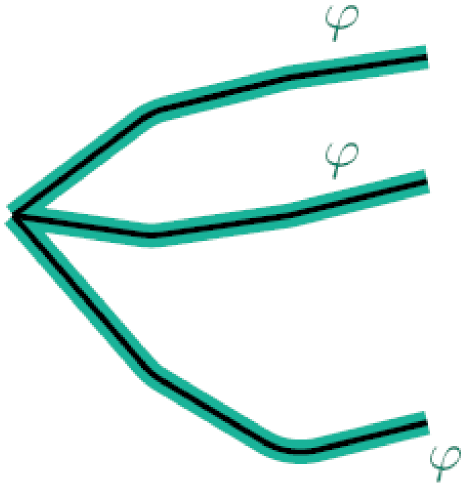
Summary of temporal operators in UPPAAL

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 an existing path, for all states φ	E[] φ
EF φ	For an existing 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

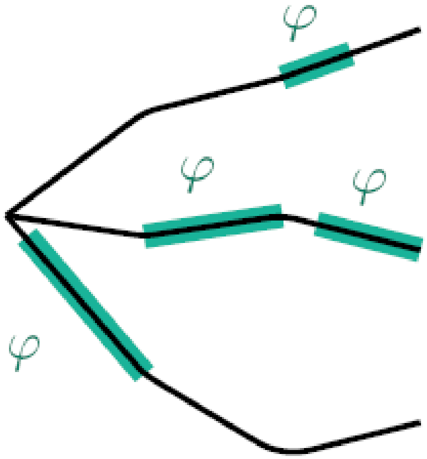
UPPAAL: φ and ψ are Boolean expressions on clocks, variables and state names

Composite operators for all paths

AG φ



AF φ

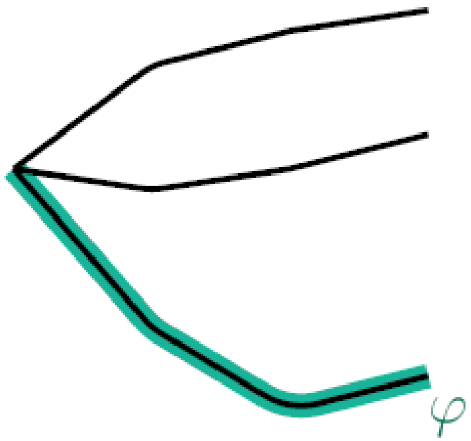


AG φ: For all paths,
for all states φ is true

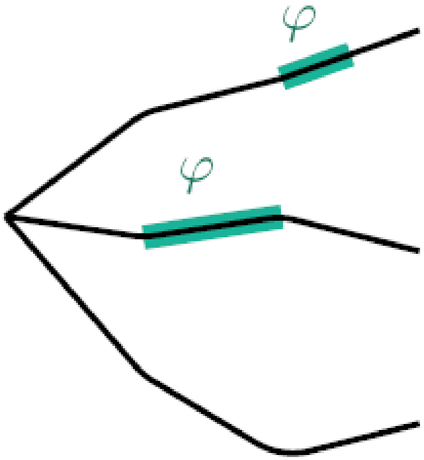
AF φ: For all paths,
for a state eventually φ
becomes true

Composite operators for an existing path

EG φ



EF φ



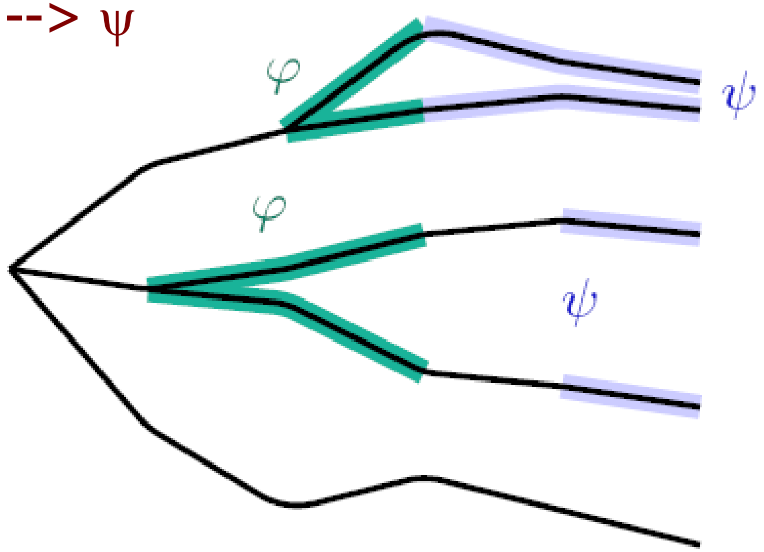
EG φ: There exists a path, where for all states φ is true

EF φ: There exists a path, where for a state 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

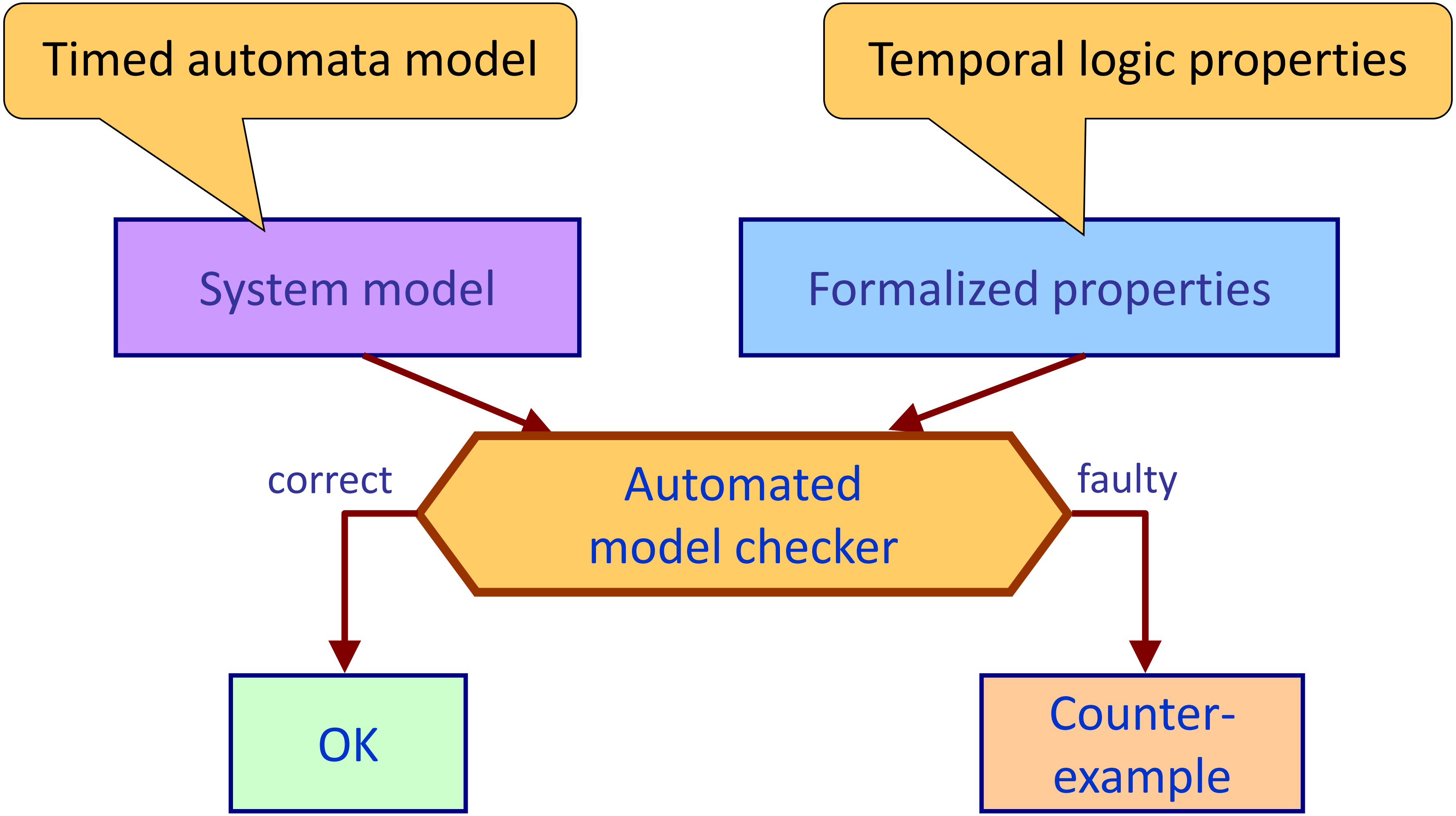
Let us consider an air-conditioner with states labelled by the following propositions:
{Switched-off, Switched-on, Faulty, LightCooling, StrongCooling, Heating, Ventilating}

- These atomic propositions can be used in the formalized properties
- The reachability properties refer to the initial state of the system
- The behaviour of the air-conditioner may not be known when the properties are formalized (the behavioural model shall be verified using these properties)

Examples for formalized properties:

- If the air-conditioner is faulty then it shall be eventually repaired:
  AG(Faulty => AF (¬Faulty)) or Faulty --> (¬Faulty)
- If the air-conditioner is faulty then it shall not heat:
  AG (¬(Faulty ∧ Heating))
- It shall be possible to eventually switch off the air-conditioner:
  AF (Switched-off)
- The air-conditioner will eventually become faulty (Murphy's law) :
  AF (Faulty)

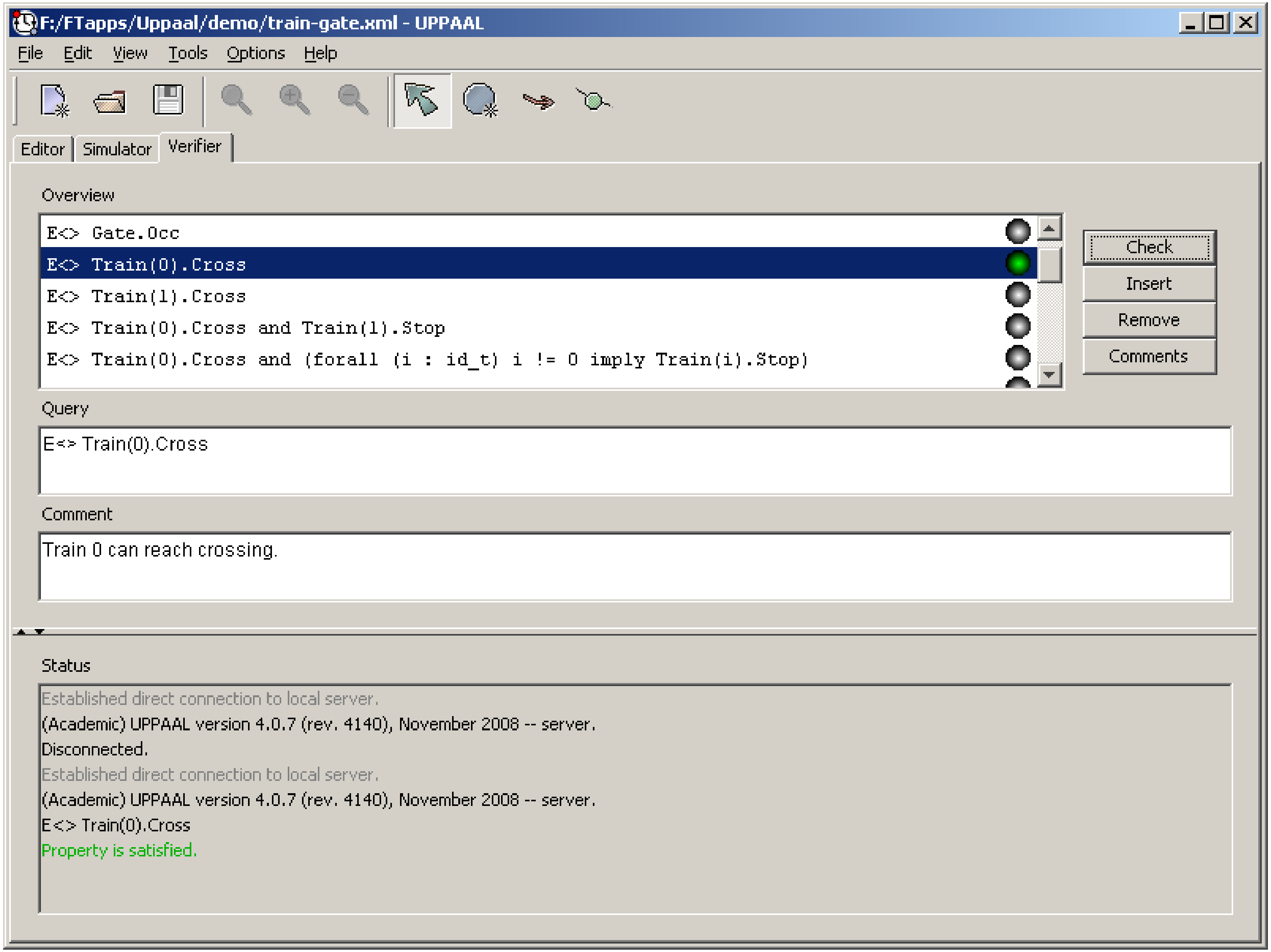
Model checking



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



```
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

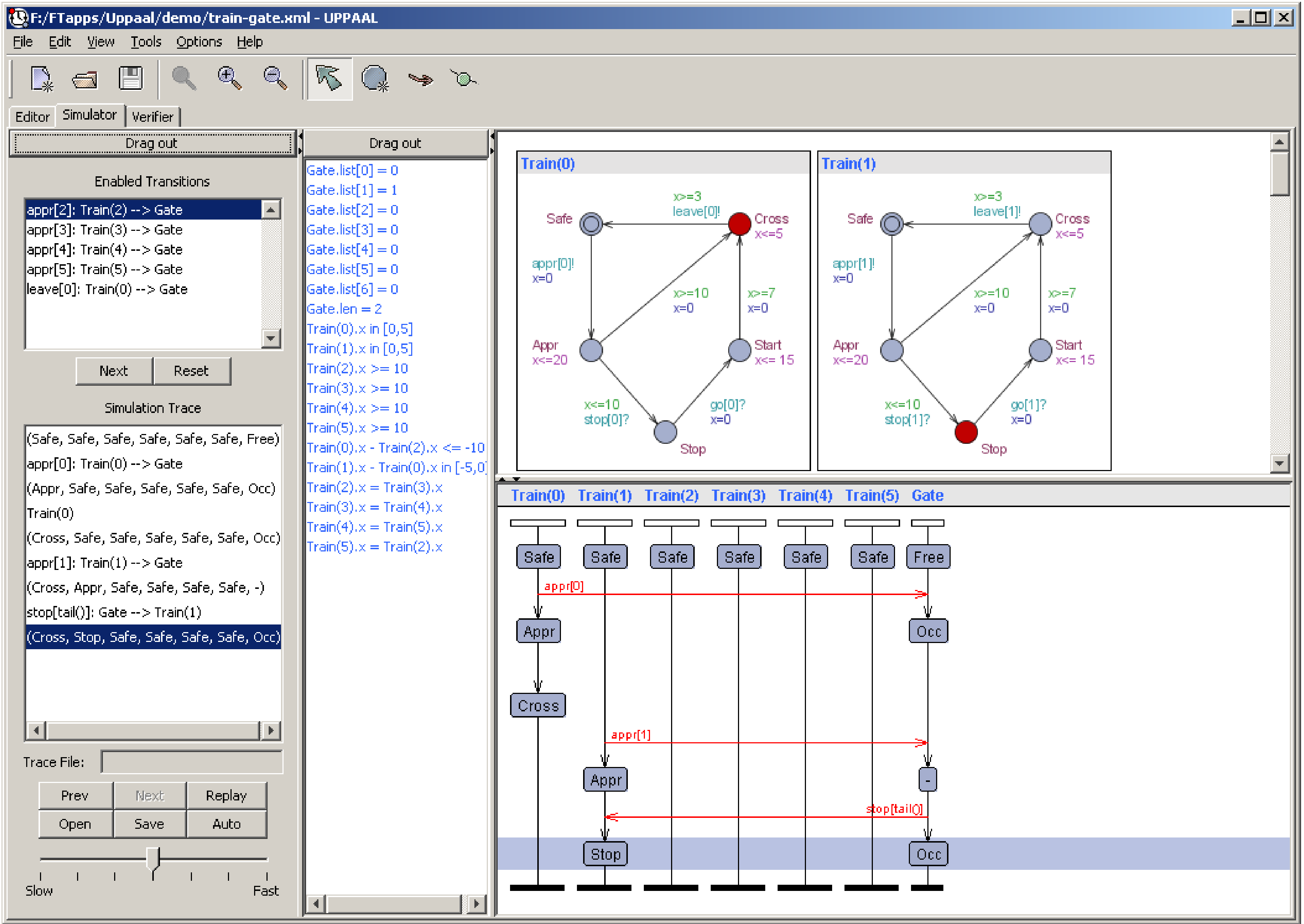
Comment

Train 0 can reach crossing.

Status

```
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

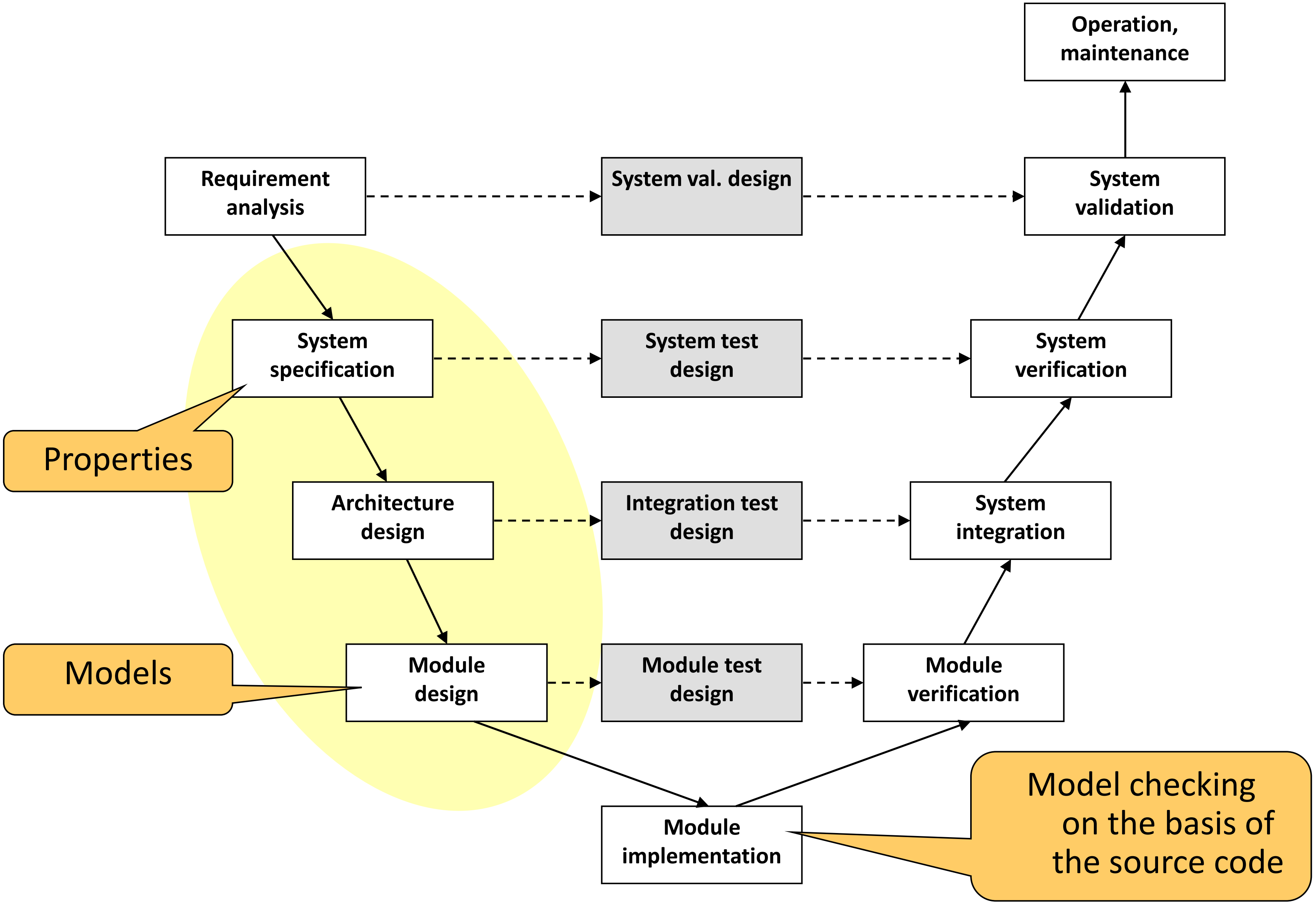
- (Safe, Safe, Safe, Safe, Safe, Safe, Free)
- appr[0]: Train(0) --> Gate
- (Appr, Safe, Safe, Safe, Safe, Safe, Occ)
- Train(0)
- (Cross, Safe, Safe, Safe, Safe, Safe, Occ)
- appr[1]: Train(1) --> Gate
- (Cross, Appr, Safe, Safe, Safe, Safe, -)
- stop[tail()]: Gate --> Train(1)
- (Cross, Stop, Safe,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

Trace File:

```
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
```

Demo

Summary: Model checking in the lifecycle



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 (in specific cases)
    - 10^{20}, or even 10^{100} states can be checked automatically
  - 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 to 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”

Summary of model based design and verification

- Formal modeling:
  - Timed automata models
- Formalization of properties:
  - Temporal logic
- Formal verification:
  - Model checking
- Source code synthesis:
  - Template based code generation from timed automata
- Monitor code synthesis:
  - Runtime verification of the control flow