Verification of the Detailed Design

Istvan Majzik
majzik@mit.bme.hu

Budapest University of Technology and Economics
Dept. of Measurement and Information Systems

Overview

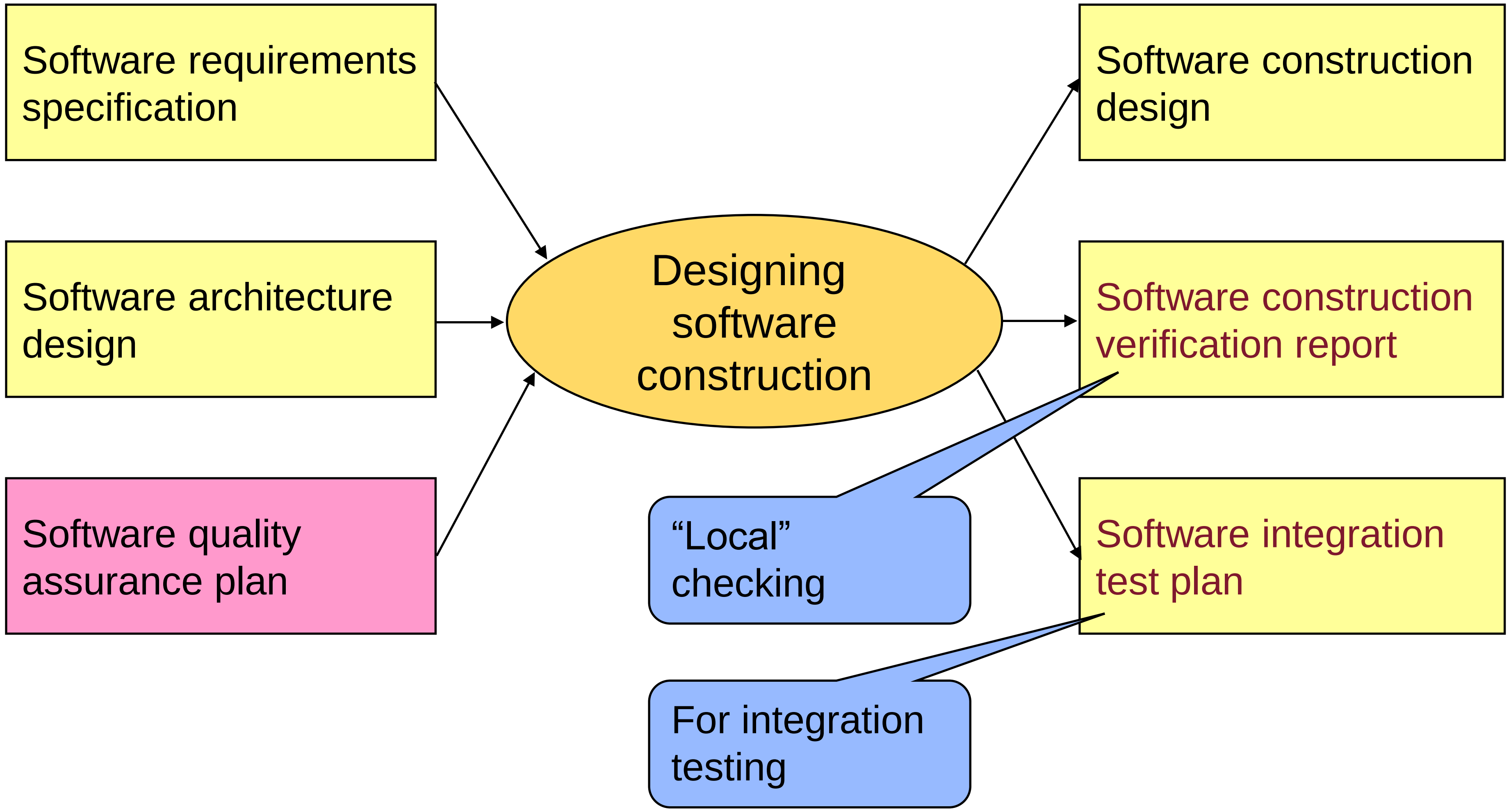
- Preparation of the detailed design
  - Software construction
  - Module (component) design
- Verification
  - Verification criteria
  - Techniques
- Formal verification
  - Basic formalisms for representing the design
  - Formalization of the requirements (to be continued)

Preparation of the detailed design

Software construction

Module (component) design

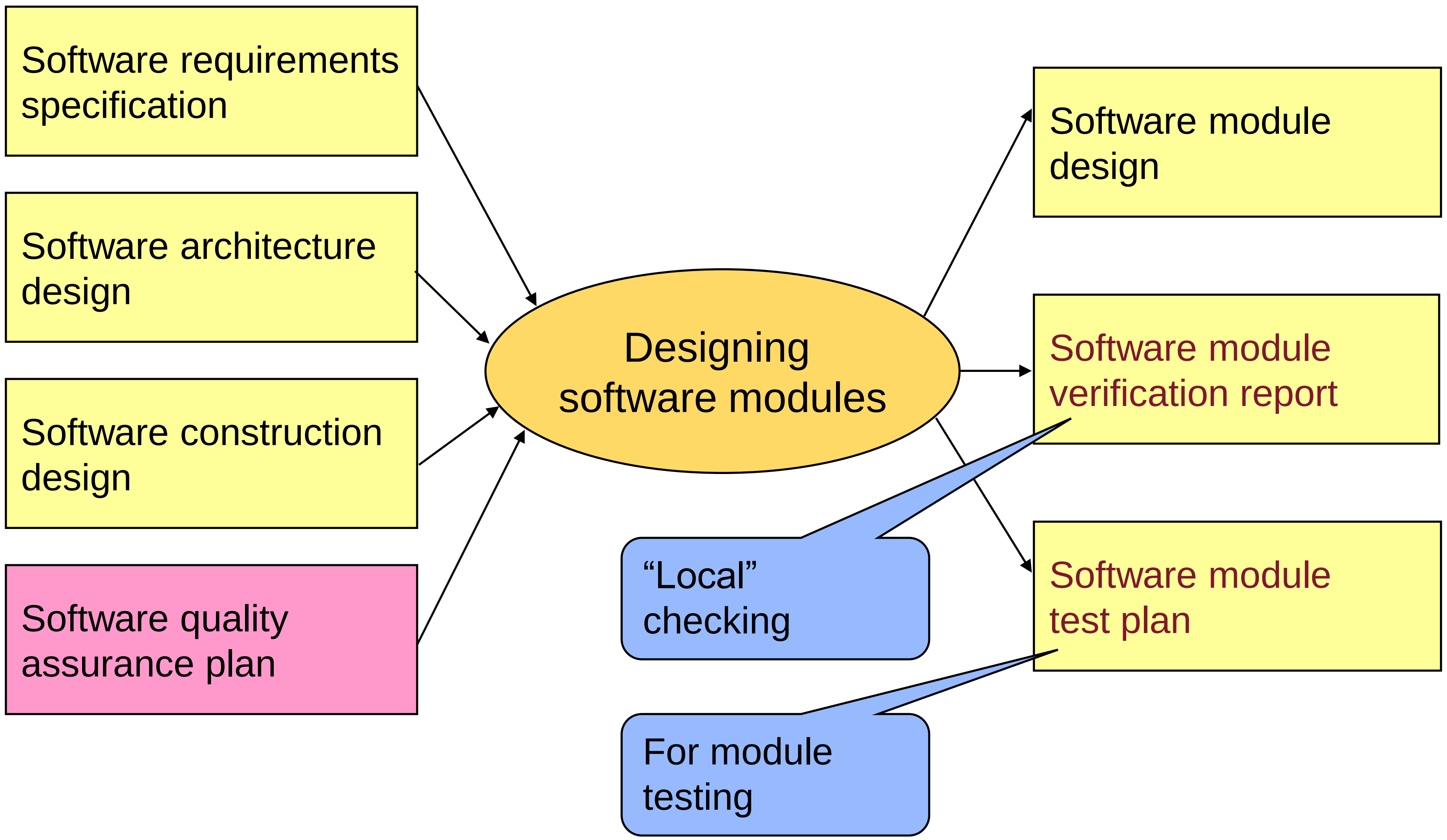
Software construction


Software requirements specification
Software architecture design
Software quality assurance plan
Designing software construction
Software construction design
Software construction verification report
Software integration test plan
"Local" checking
For integration testing

Software construction

- To be designed:
  - System level algorithms for the interaction of modules
  - Global data structures
- Design (description) language:
  - Capturing interactions and information exchange (ordering, timeliness)
  - Representing (abstract / concrete) data structures
  - Characterized by modularity, abstraction, precision
- Available methods:
  - Formal, semi-formal, structured methods
- Specific characteristics (in critical systems):
  - Fully defined interfaces
  - Module and parameter size / complexity limits
  - Information hiding

Software module (component) design


Software requirements specification
Software architecture design
Software construction design
Software quality assurance plan
Designing software modules
Software module design
Software module verification report
Software module test plan
"Local" checking
For module testing

Software module design

- Internal design of software modules
  - Algorithms
  - Data structures
- Design (description) language
  - Languages closer to implementation
    - E.g., pseudo-codes can be used
  - Formal, semi-formal, structured languages
    - Description of the behavior is important: control flow automata, state machines, statecharts

Verification of the detailed design

Verification criteria

Techniques

Verification criteria for the detailed design

- Local characteristics of the design
  - Completeness, consistency, verifiability, feasibility
- Conformance (to the outputs of previous steps)
  - Behavioral properties specified earlier
    - Safety properties: “Something bad never happens”
    - Liveness properties: “Something good will eventually happen”
  - Conformance of abstract and refined behavior
    - Simulation, bisimulation, refinement relations
- Completeness of test plans

Verification techniques for the detailed design

- Static checking
  - Review: Checklist, error guessing
  - Structure based analysis
    - Control flow: complexity, structure, …
    - Data flow: initialization and use of variables, ordering of access, …
    - Border values: switching to different behavior
  - Analysis of unwanted behavior
    - Potential influences through reserving resources (CPU, memory), …
  - Symbolic execution
    - Checking inputs that cause parts of a program to execute
- Dynamic checking
  - Prototype implementation and animation
    - Detection of problematic cases requires particular care
  - Simulation
    - Can we simulate all possible executions?
  - Formal verification
    - For proving properties (“exhaustive” checking)

Formal verification

- Use of precise, mathematical techniques (esp. discrete mathematics, mathematical logic)
  - Formal language: Formal syntax and semantics
    - Behavior description (design, implementation)
    - Property description (property specification)
  - Mathematical algorithm for verification
    - Checking design properties (e.g., ambiguity)
    - Checking changes (e.g., refinement)
    - Conformance of behavior and property descriptions

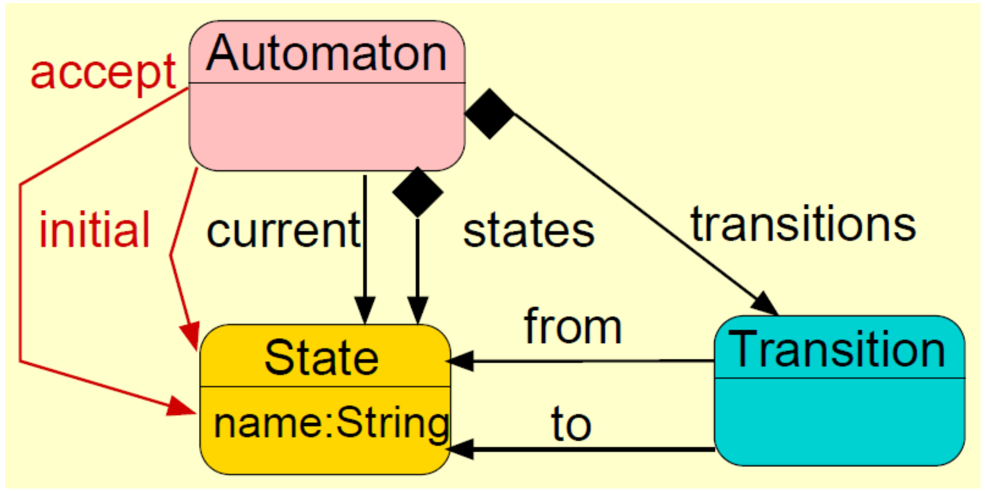
- Crucial aspect: Formalization of the real problem
  - Not automatized
  - Simplification, abstraction is needed (it has to be validated)

Formal syntax

- Mathematical description: $KS = (S, R, L)$ and AP, where

$$\text{AP}=\{\text{P,Q,R,...}\}$$
$$\text{S}=\{s_1, s_2, s_3, \ldots s_n\}$$
$$\text{R} \subseteq \text{S} \times \text{S}$$
$$\text{L: S} \rightarrow 2^{AP}$$

- BNF: BL ::= true | false | $p \wedge q$ | $p \vee q$
- Metamodel:
  - Abstract syntax: grammar rules
  - Concrete syntax: representation



Formal semantics (overview)

The meaning of the model following the syntax:

- Operational semantics: “for programmers”
  - Defines what happens during operation (computation)
  - Builds on simple notions of execution: states, events, actions, …
  - E.g., to describe the state space for verification
- Axiomatic semantics: “for correctness proofs”
  - Predicate language + set of axioms + inference rules
  - E.g., for automated theorem prover tools
- Denotational semantics: “for compilers”
  - Mapping to a known domain, driven by the syntax
    - Known mathematical domain, e.g., computation sequence, control-flow graph, state set, … and their operations (concatenation, union, etc.)
    - Analysis of the model: analysis of the underlying domain
  - E.g., for synthesis tasks

Models for formal verification

- Design models (with operational semantics)
  - Engineering (design) models:
    - E.g., DSL, UML with (semi-)formal semantics
  - Higher-level formal models:
    - Control-oriented: automata, Petri nets, …
    - Data processing-oriented: dataflow networks, …
    - Communication-oriented: process algebra, …
  - Basic mathematical models:
    - KS, KTS, LTS, finite state automata, Büchi automata
- Property descriptions
  - Higher level:
    - Time diagram, message sequence chart (MSC)
  - Base level:
    - First order logic, temporal logic, reference automaton

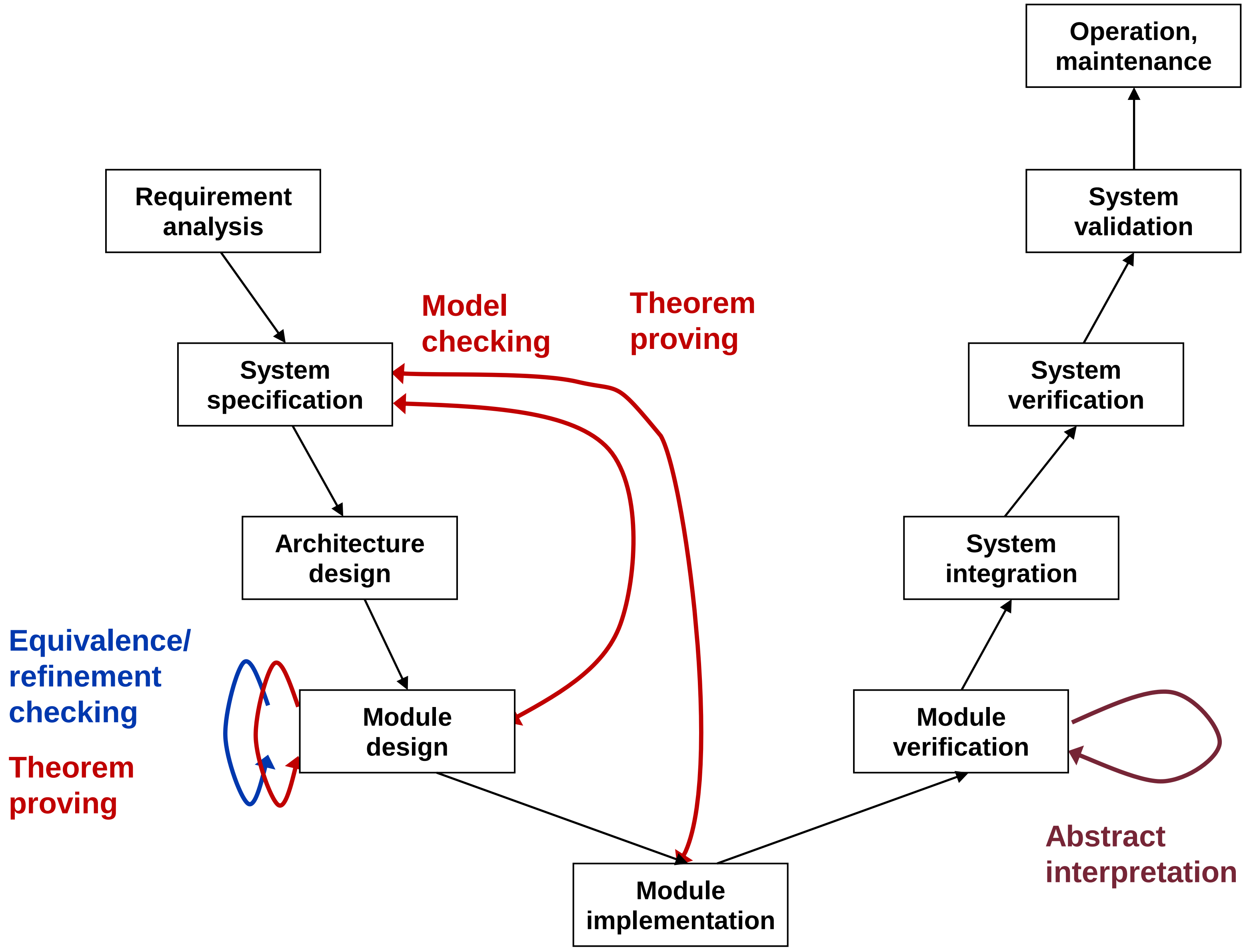
Typical formal verification techniques

Models / techniques	**Behavior description (basic model)**	**Property description (basic property)**
Model checking	Kripke structure (KS), Kripke transition system (KTS)	Temporal logics, first order logics
Equivalence / refinement checking	Labeled transition system (LTS), finite automata	LTS, automata (as reference behavior)
Theorem proving	Deduction system	Theorem to be proved (first order logic)
Static analysis (abstract interpretation)	Kripke transition system (extracted from the program)	Assertion (first order logic)

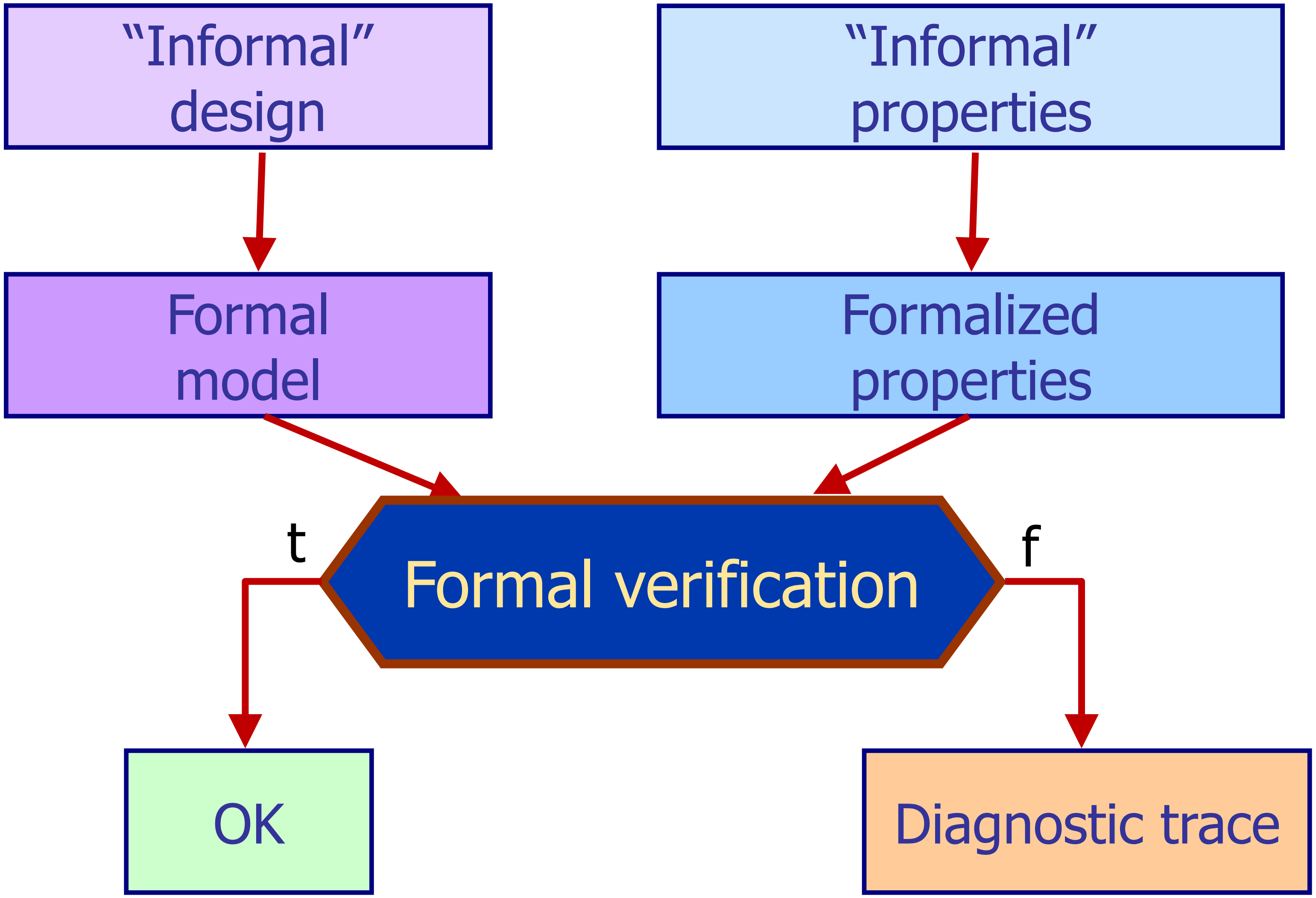
Advantages and limitations of the techniques

- Model checking, equivalence/refinement checking
  - ☺ Fully automated, exhaustive checking
  - ☺ Construction of diagnostic trace (for debugging)
  - ☹ State space exploration (handled partially)
- Theorem proving
  - ☺ Scalable for complex systems (e.g., by induction)
  - ☺ High expressive power
  - ☹ Interactive (need hints, e.g., to find a proof strategy)
  - ☹ There is no diagnostic trace (counter-example)
- Static analysis (abstract interpretation)
  - ☺ Handling state space explosion by abstraction
  - ☹ Abstraction is hard to automate

The role of formal verification techniques

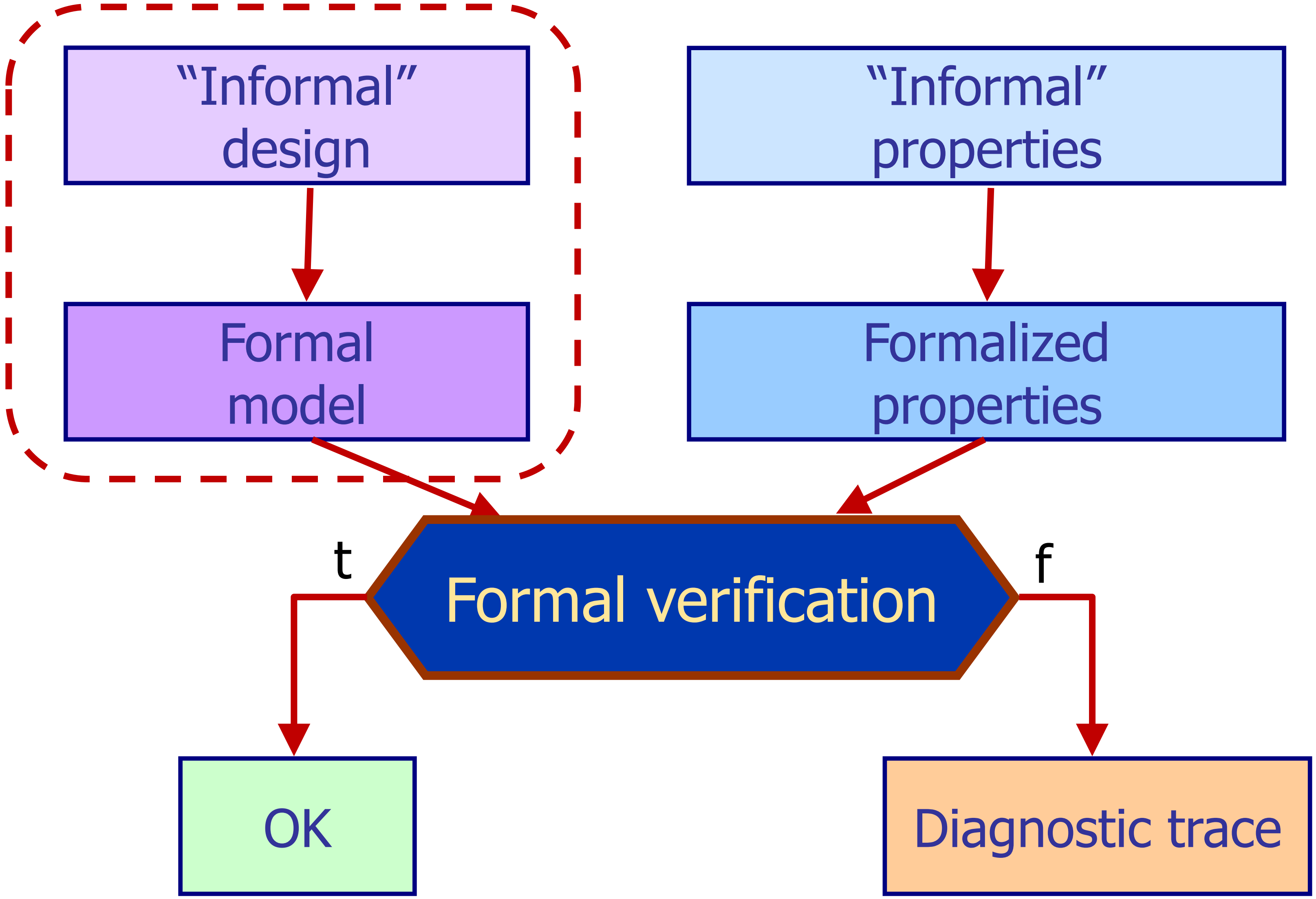


Our goal


"Informal" design
"Informal" properties
Formal model
Formalized properties
Formal verification
t
f
OK
Diagnostic trace

Formalization of the design:
Basic formalisms

Our goal


"Informal" design
Formal model
"Informal" properties
Formalized properties
Formal verification
t
f
OK
Diagnostic trace

Basic formalisms (overview)

- Kripke structure (KS)
  - States, transitions
  - Local properties of states as labels
- Labeled transition system (LTS)
  - States, transitions, actions
  - Local properties of transitions as labels
- Kripke transition system (KTS)
  - States, transitions
  - Local properties of states and transitions as labels
- Finite state automata (FSA)
  - Accepting and rejecting runs on finite input sequences
  - Büchi acceptance criteria on infinite input sequences
- Timed automata (TA)
  - Extensions: variables, clocks, synchronization

1. Kripke structure

Basic characteristics:

- Expresses properties of states: labeling by atomic propositions
- Possibly more than one labels per state
- Application: description of behavior or algorithm

Definition:

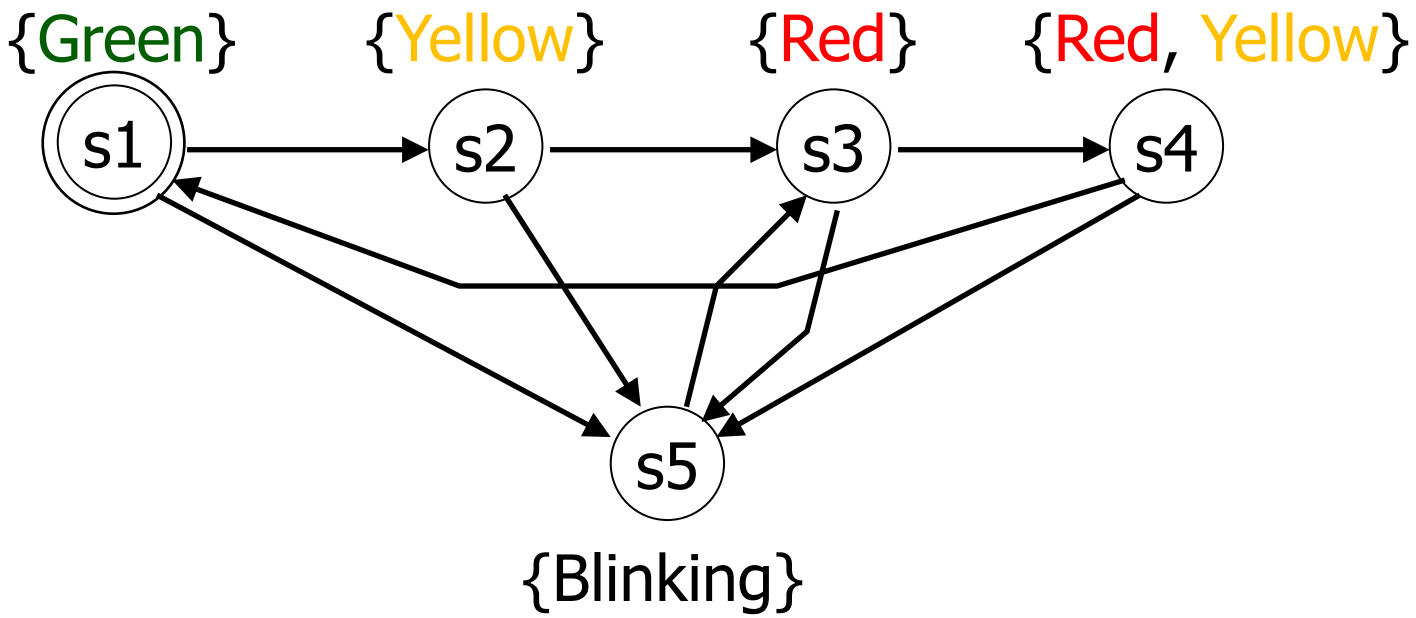
A Kripke structure KS over a set of atomic propositions $AP = \{P, Q, R, \dots\}$ is a tuple (S, R, L) where

- $S = \{s_1, s_2, \dots, s_n\}$ is a finite set of states,
  $I \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is the set of transitions and
- $L : S \to 2^{AP}$ is the labeling of states by atomic propositions

Example: Kripke structure

Traffic light controller

- $\mathrm{AP} = \{\mathrm{Green}, \mathrm{Yellow}, \mathrm{Red}, \mathrm{Blinking}\}$
- $S = \{s_1, s_2, s_3, s_4, s_5\}$



2. Labeled transition system

Basic characteristics:

- Expresses properties of transitions: labeling by actions
- Exactly one action per transition
- Application: modeling of communication and protocols

Definition:

A labeled transition system LTS over a set of actions $Act = \{a, b, c, \dots\}$ is a triple $(S, Act, \rightarrow)$ where

- $S = \{s_1, s_2, \dots, s_n\}$ is a finite set of states,

  $I \subseteq S$ is the set of initial states,
- $\rightarrow : S \times Act \times S$ is the set of transitions

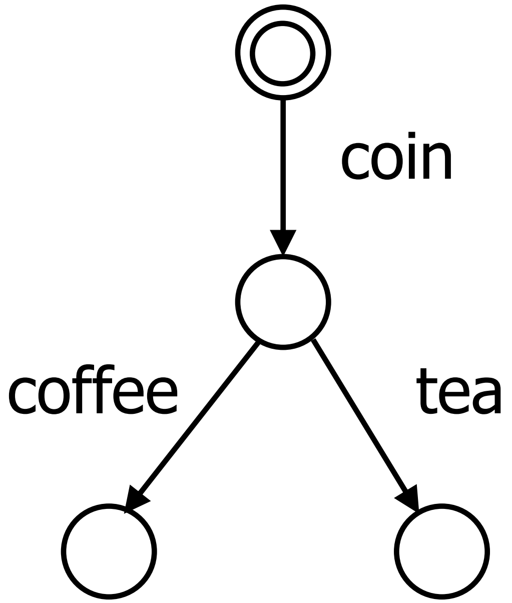
We denote by $s \overset{a}{\rightarrow} s'$ iff $(s, a, s') \in \rightarrow$.

Example: Labeled transition system

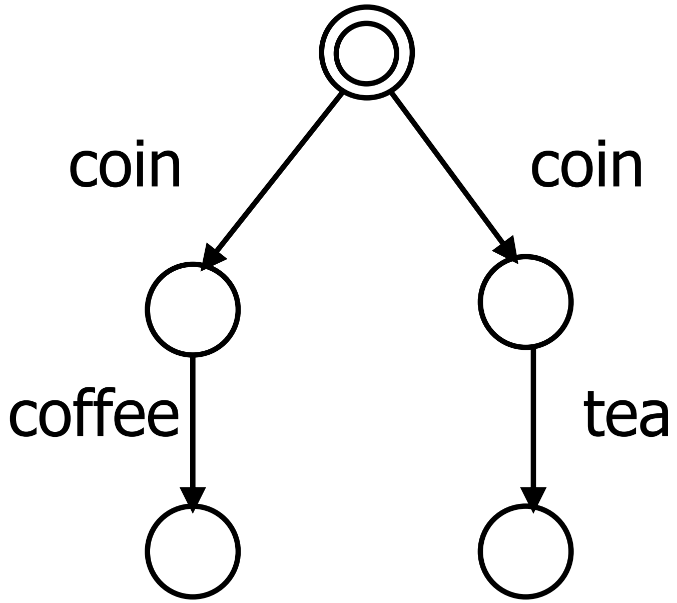
Vending machine

- $Act = \{coin, coffe, tea\}$

T1



T2



3. Kripke transition system

Basic characteristics:

- Expresses properties of both states and transitions: labeling by atomic propositions and actions
- Possibly more than one labels per state, exactly one action per transition

Definition:

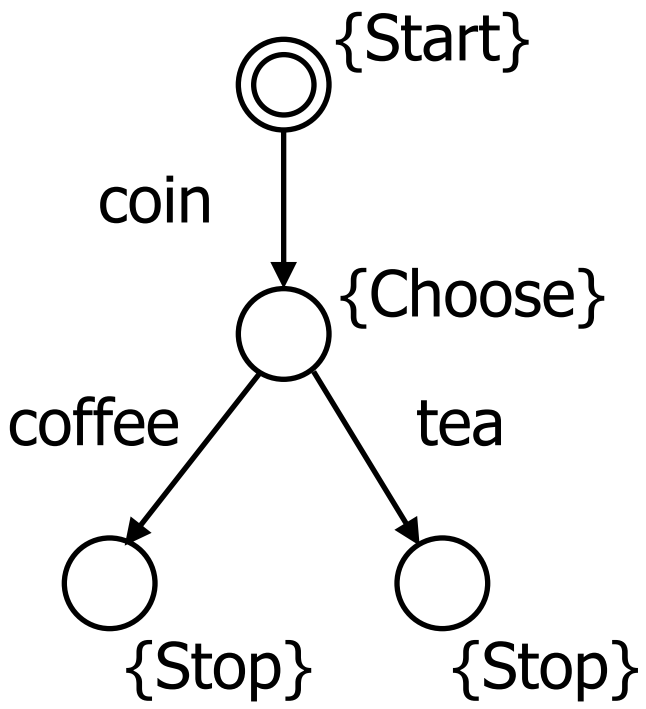
A Kripke transition system KTS over a set of atomic propositions AP and set of actions Act is a tuple $(S, \rightarrow, L)$ where

- $(S, Act, \rightarrow)$ is an LTS
- $L : S \rightarrow 2^{AP}$ is the labeling of states by atomic propositions

Example: Kripke transition system

Vending machine with state labeling

- $Act = \{\mathrm{coin}, \mathrm{coffee}, \mathrm{tea}\}$
- $AP = \{\mathrm{Start}, \mathrm{Choose}, \mathrm{Stop}\}$



4. Automata on finite words

- $A=(\Sigma, S, S_0, \rho, F)$ where
  - Σ alphabet, S states, S_0 initial states
  - ρ state transition relation, $\rho: S \times \Sigma \to 2^S$
  - F set of accepting states
- Run of an automaton
  - State sequence $r=(s_0, s_1, s_2, \ldots s_n)$ on the incoming word $w=(a_0, a_1, a_2, \ldots a_n)$
  - r is an accepting run if $s_n \in F$
  - A word w is accepted by the automaton, if there is an accepting run over w
- Language L accepted by the automaton A:

$$L(A)=\{ w \in \Sigma^* \mid w \text{ accepted}\}$$

Automata on infinite words

- The accepting state (at the end of an input word) cannot be checked
- Büchi acceptance criterion:
  - On the incoming infinite word $w=(a_0, a_1, a_2, \dots)$ there is an $r=(s_0, s_1, s_2, \dots)$ infinite state sequence
  - $\lim(r)=\{s \mid s$ occurs infinitely often,
    i.e., there is no j, such that $\forall k>j: s \neq s_k\}$
  - Accepting run: $\lim(r) \cap F \neq 0$
  - A word w is accepted by the automaton,
    if there is an accepting run over w
    (i.e., accepting state occurs infinitely often)
- Language L accepted by the automaton A:

  $L(A)=\{ w \in \Sigma^* \mid w \text{ accepted}\}$

Timed Automata:
Finite State Automata with Time

Timed Automata in the UPPAAL model checker

Timed Automata: Extension with variables

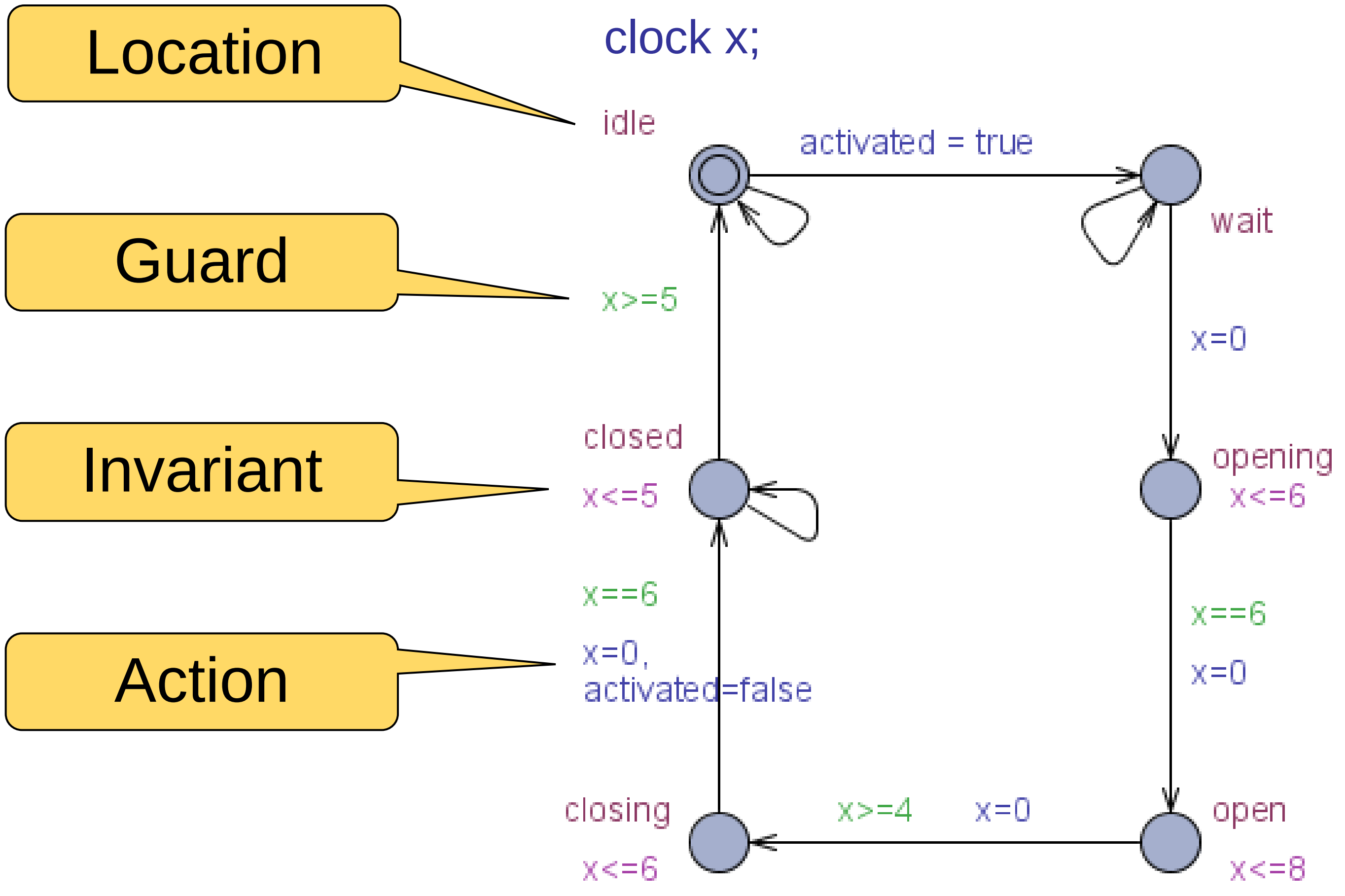
- Basic formalism: Finite state automaton (FSA)
  - Control locations (named)
  - Edges
- Language extension: integer variables
  - Variables with restricted domain (e.g. int[0, 1] id)
  - Constants
  - Integer arithmetic
- Use of variables: on transitions
  - Guard: predicate over variables
    - The transition can fire only if predicate holds
  - Action: variable assignment

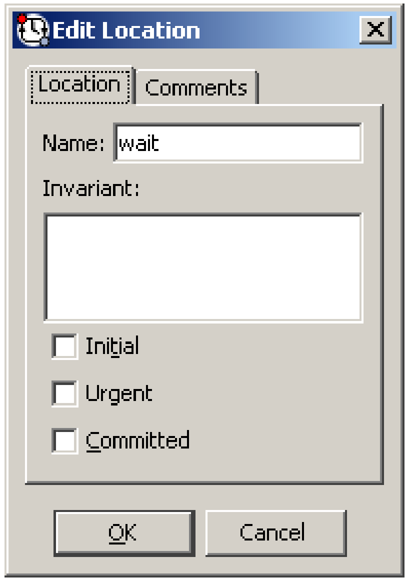
Timed automata: Extension with clock variables

- Goal: modeling 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)
- Language extension: clock variables
  - Measuring time elapse by a constant rate
- Use of clock variables on transitions
  - Guard: predicate over clock variables
  - Action: resetting clocks to zero
- Use of clock variables on locations
  - Location invariant: predicate over clock variables, being in a location is valid until its invariant holds

Timed automata in UPPAAL

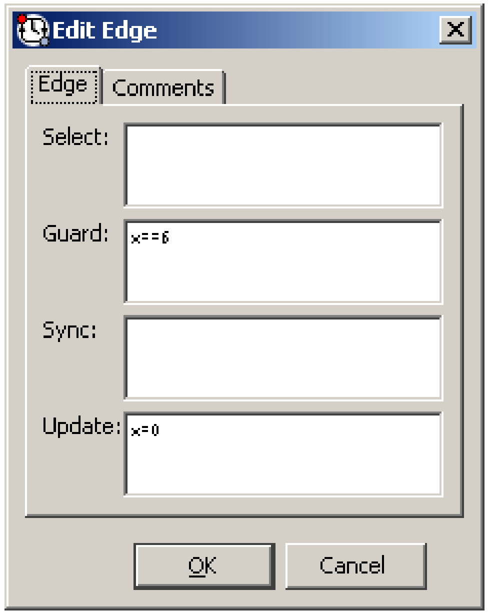
Example: revolving door





Name: wait

Invariant:



Select:

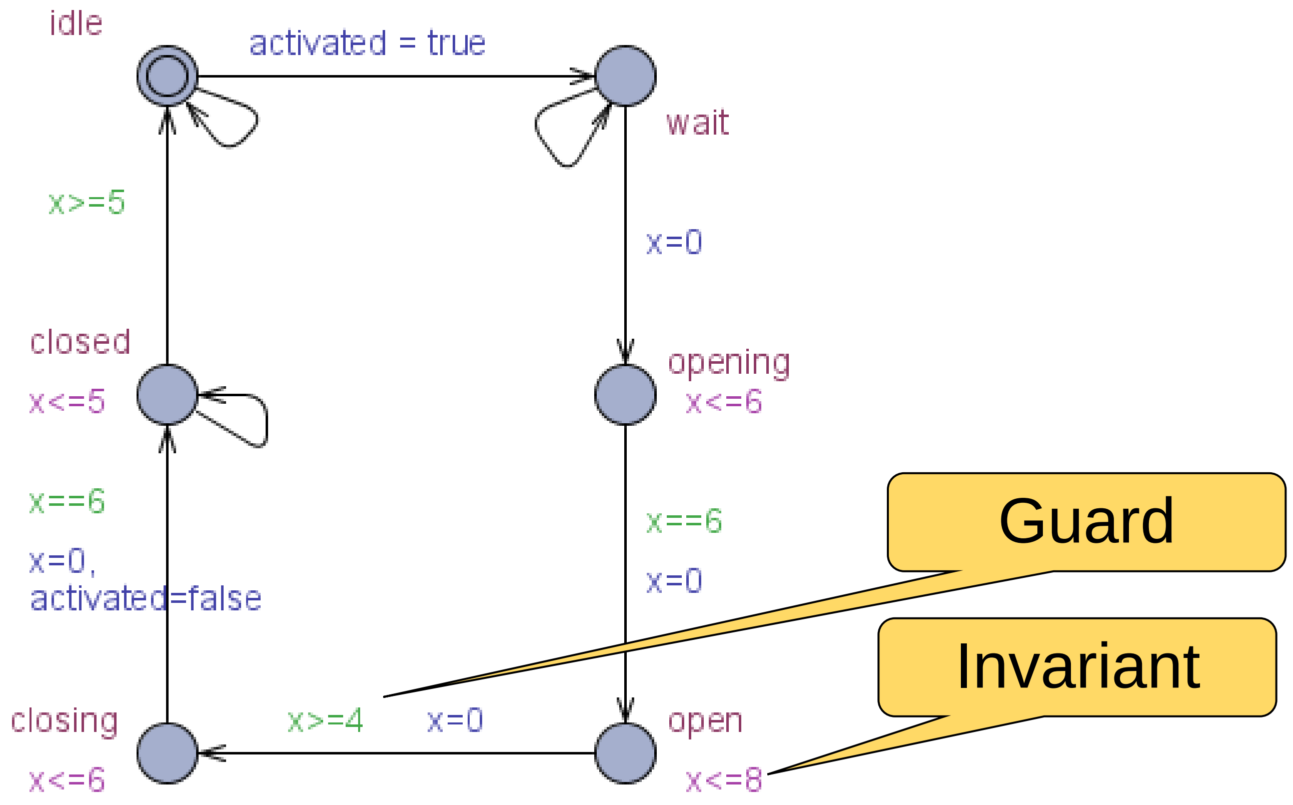
Guard: x==6

Sync:

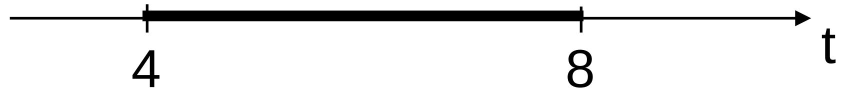
Update: x=0

Role of guards and invariants

clock x;

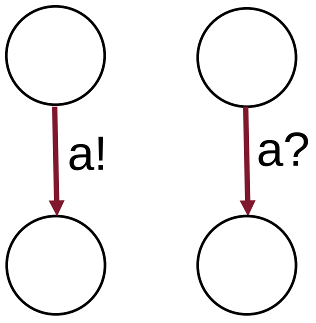


Upon exiting location open, the value of clock is in interval [4, 8]



Extensions for concurrency

- Goal: modeling networks of automata
  - Interaction: Synchronization between automata transitions
  - Synchronous communication (handshake)
    - Sending and receiving a message occurs at the same time
    - Modeling of asynchronous behavior: by modeling channels
- Language extension: synchronized actions
  - Channels for sending messages
  - Sending a message: ! operator
    Receiving a message: ? operator
  - E.g.: synchronization labels a! and a? for channel a
- Parameterization
  - Arrays of channels: E.g. channel a[id] for a variable id



chan a

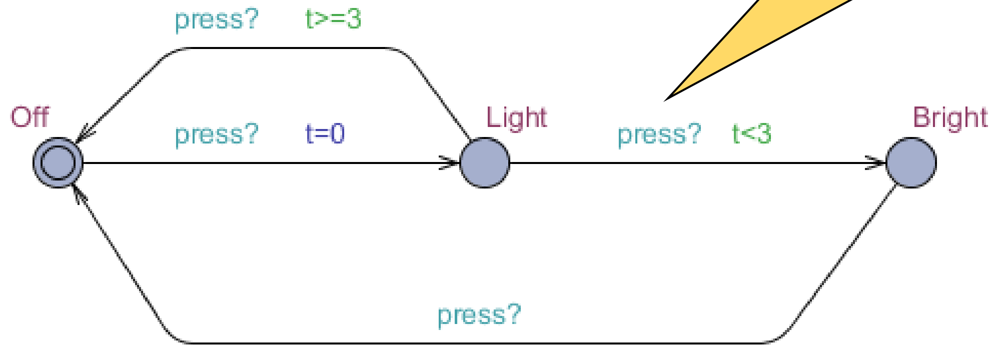
Example for clocks and synchronization

Declarations:

clock t, u;

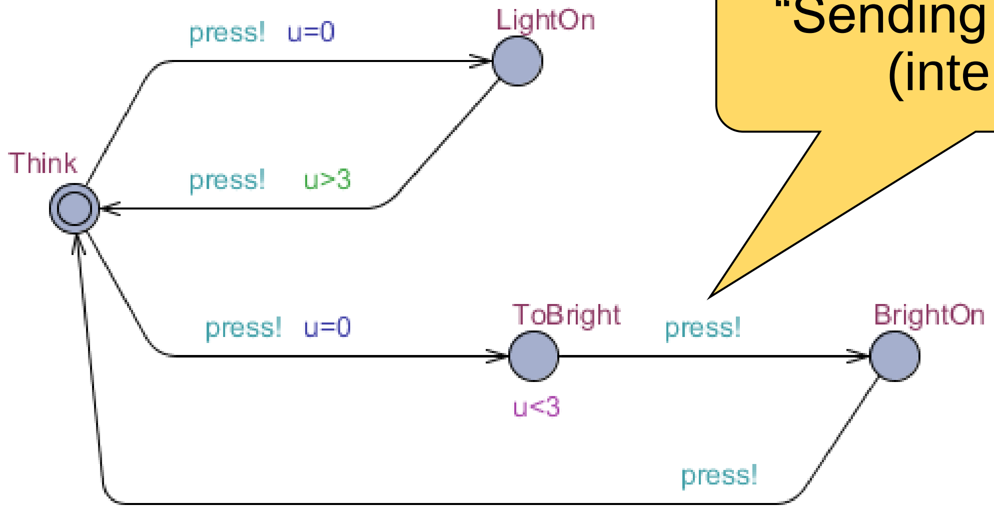
chan press;

Switch:



“Receiving a message” (interaction)

User:

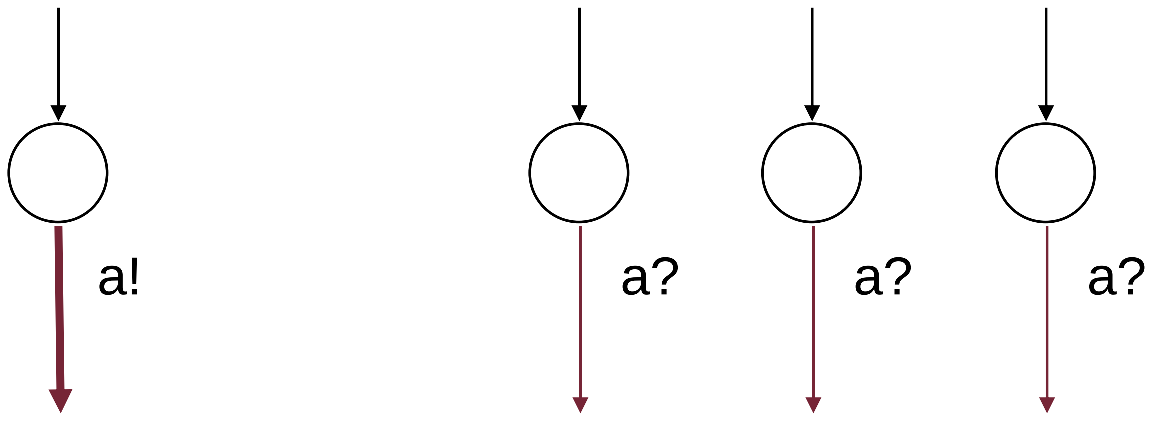


“Sending a message” (interaction)

Further extensions: broadcast channel

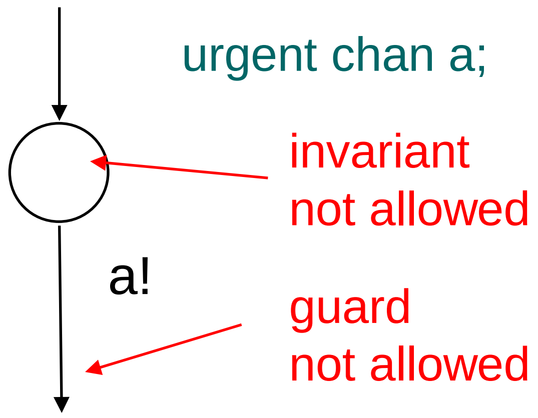
- Broadcast channel: one-to-many communication
  - Sending a message unconditionally
    - No handshake needed
  - All processes that are ready to receive the message will synchronize
    - Receiving edge can only be taken upon receiving message
  - Restriction: no guard on receiving edge

broadcast chan a;



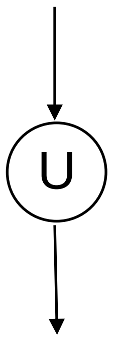
Further extensions: Urgent channel

- Urgent channel: prohibit time delay (waiting for synchronization)
  - The synchronization is executed without delay,
    (other edges might be traversed before, but only instantly)
  - Restrictions:
    - No guard is allowed on an edge labeled with the name of an urgent channel
    - No invariant is allowed on a location that is the source of an edge labeled with the name of an urgent channel

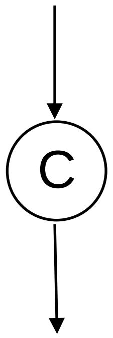


Further extensions: special locations

- Urgent location: prohibit time delay (waiting in location)
  - Time is not allowed to progress in the location
  - Equivalent model:
    - Introduce a clock variable: clock x
    - Reset clock on all incoming edges: x:=0
    - Add invariant: x<=0



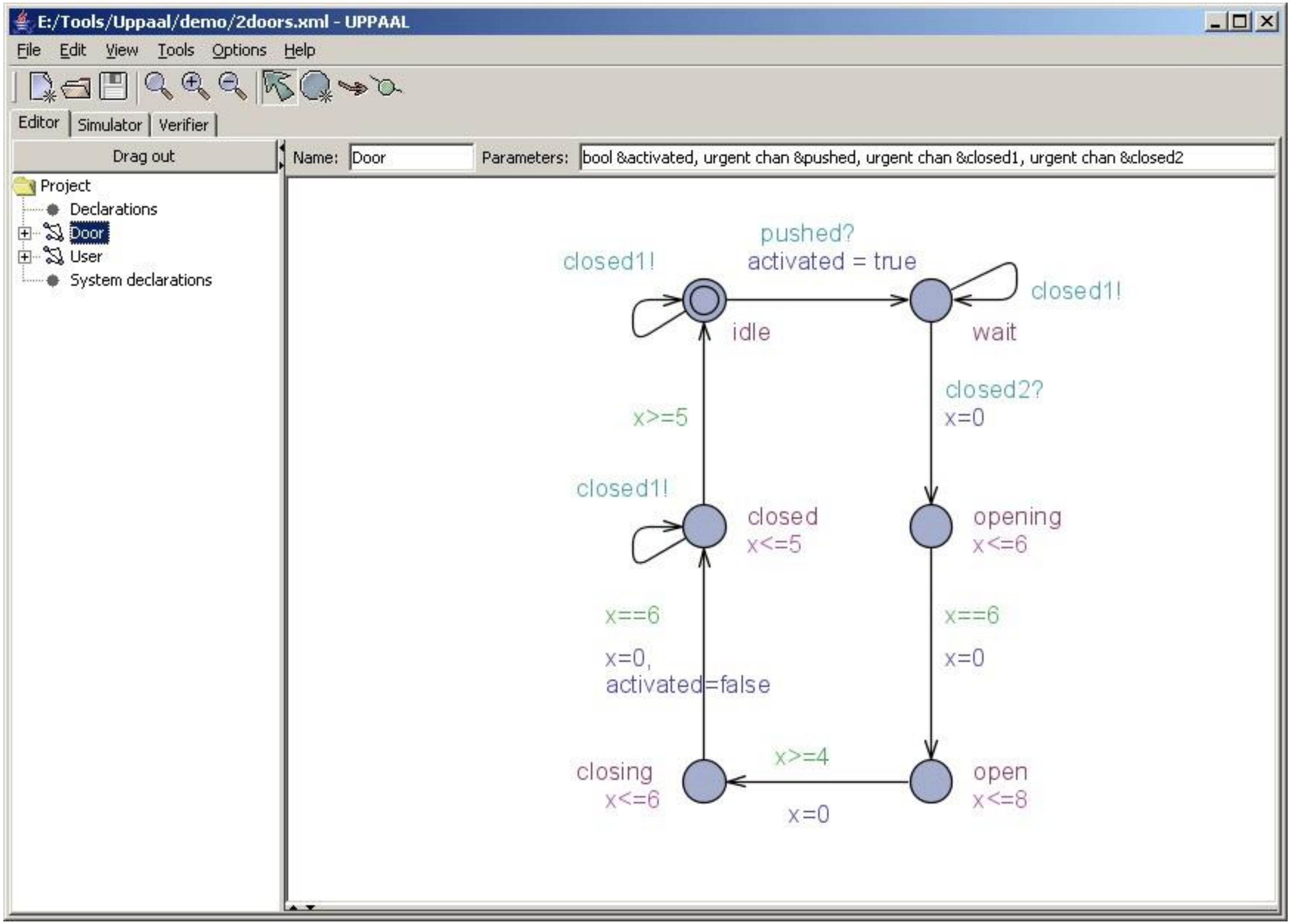
- Committed location: even more restrictive
  - A committed location is urgent
  - Committed state: at least one committed location is active
  - The next transition from a committed state must involve at least one out-edge of an active committed location



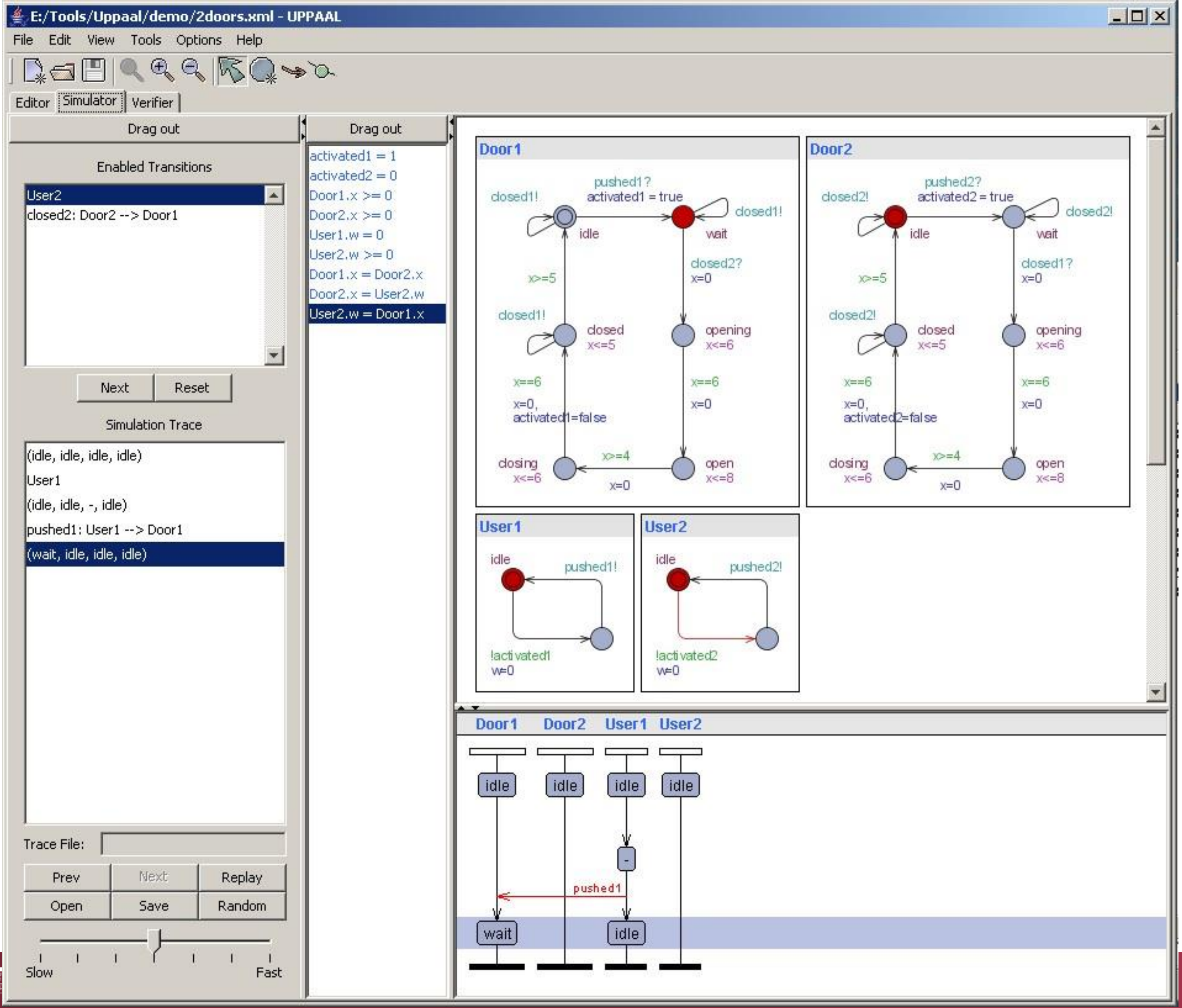
The UPPAAL model checker

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

Automaton model



Simulator



Enabled Transitions:

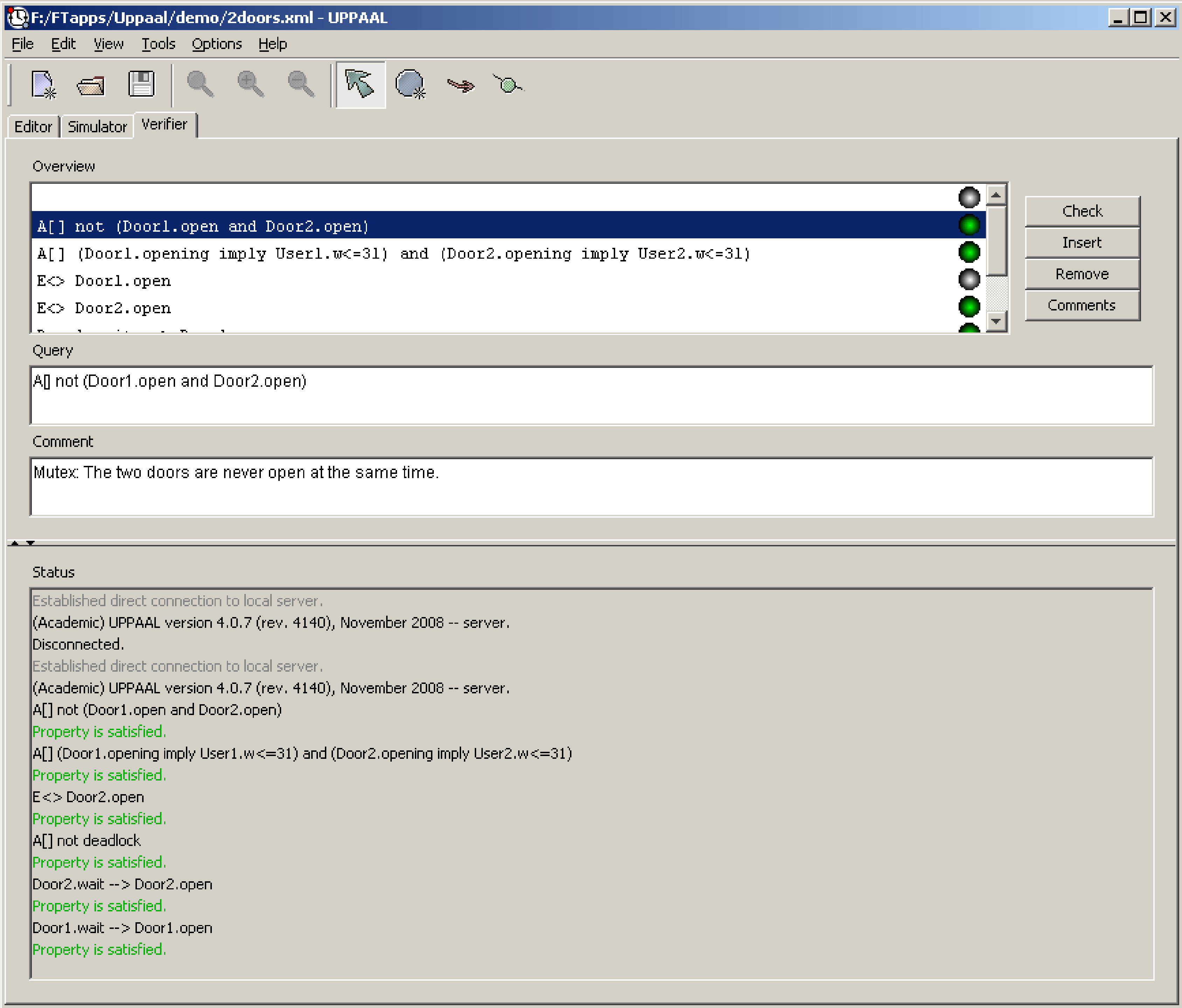
User2
closed2: Door2 --> Door1

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)

Verification


F:/FTapps/Uppaal/demo/2doors.xml - UPPAAL
File Edit View Tools Options Help
Editor Simulator Verifier
Overview
Check
Insert
Remove
Comments
Query
Comment
Status

```
A[] not (Door1.open and Door2.open)
A[] (Door1.opening imply User1.w<=31) and (Door2.opening imply User2.w<=31)
E<> Door1.open
E<> Door2.open
```

Query: A[] not (Door1.open and Door2.open)

Comment: Mutex: The two doors are never open at the same time.

```
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.
A[] not (Door1.open and Door2.open)
Property is satisfied.
A[] (Door1.opening imply User1.w<=31) and (Door2.opening imply User2.w<=31)
Property is satisfied.
E<> Door2.open
Property is satisfied.
A[] not deadlock
Property is satisfied.
Door2.wait --> Door2.open
Property is satisfied.
Door1.wait --> Door1.open
Property is satisfied.
```