The UPPAAL Model Checker

Julián Proenza

Systems, Robotics and Vision Group. UIB.
SPAIN

Universitat de les
Illes Balears

The aim of this presentation

- Introduce the **basic concepts of model checking** from a **practical perspective**
- Describe the **basic features of** the **UPPAAL** model checker
- Use **examples to illustrate the practical use** of UPPAAL for the formal verification of systems

Presentation Outline

Presentation Outline

The need for Design Validation

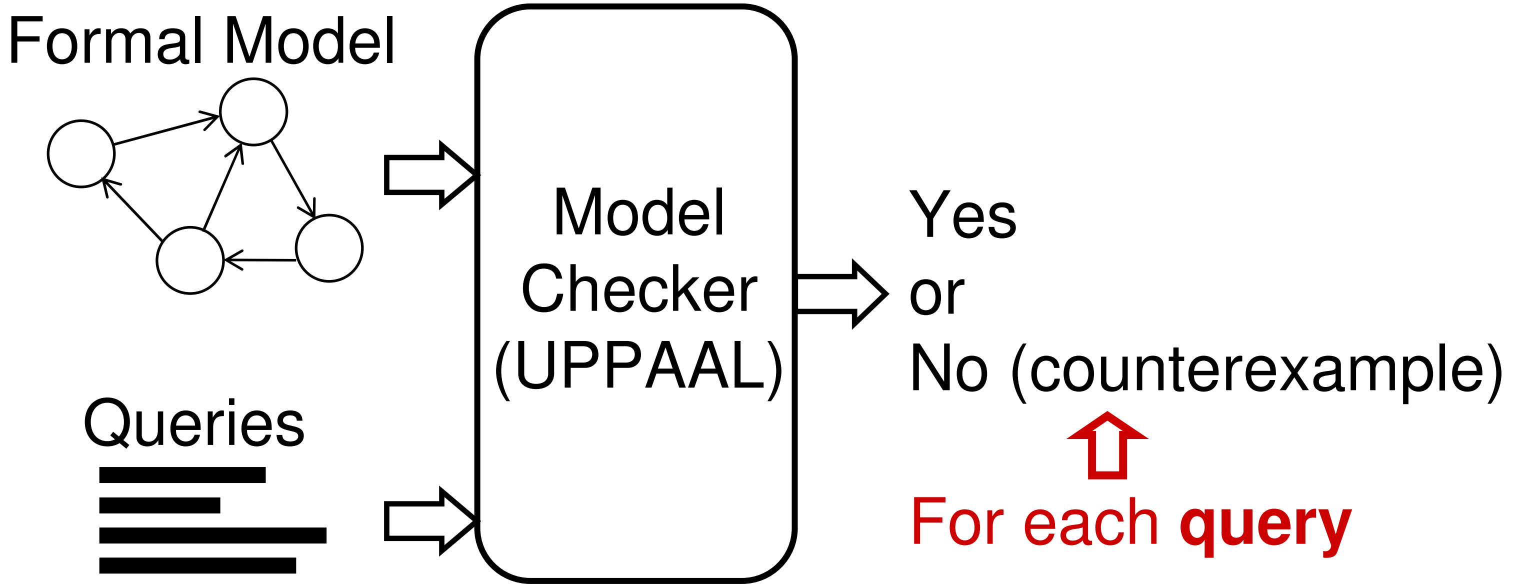
- Hardware and software are widely used in **applications where failure is unacceptable**
- Design Validation is necessary: **ensuring design correctness** at the earliest stage possible
- **Traditional Techniques** (rarely get exhaustive validation):
  - **Simulation** (on an abstraction or a model of the system)
  - **Testing** (on the actual product)
- **Formal Methods** (aimed at exhaustive validation)
  - **Deductive Verification** (costly, slow and only partially automatic)
  - **Model Checking** (for finite-state concurrent systems → automatic)

The Model Checking Technique

- Use of Formal Methods has been considered for a long time a **very desirable task** for ensuring the correct design of a system
- The complexity of these methods made them **only accessible to specialists** (mathematicians).
  - Thus they were actually only used for very critical systems
- **Model Checking** is the first technique that is **truly accessible for “normal” engineers**
  - Enabling the use of formal verification in a wider spectrum of applications (including VHDL systems)
  - Applicable to (finite-state concurrent systems) sequential circuits, communication protocols, software…

The 3 Steps of Model Checking

1. Build a **model** for the system, typically as a set of automata
2. Formalize the **properties** to be verified using expressions in a logic
3. Use the model checker (a **tool**) to generate the space of all possible states and to exhaustively check whether the properties hold in each and everyone of the possible DYNAMIC BEHAVIOURS of the model



Presentation Outline

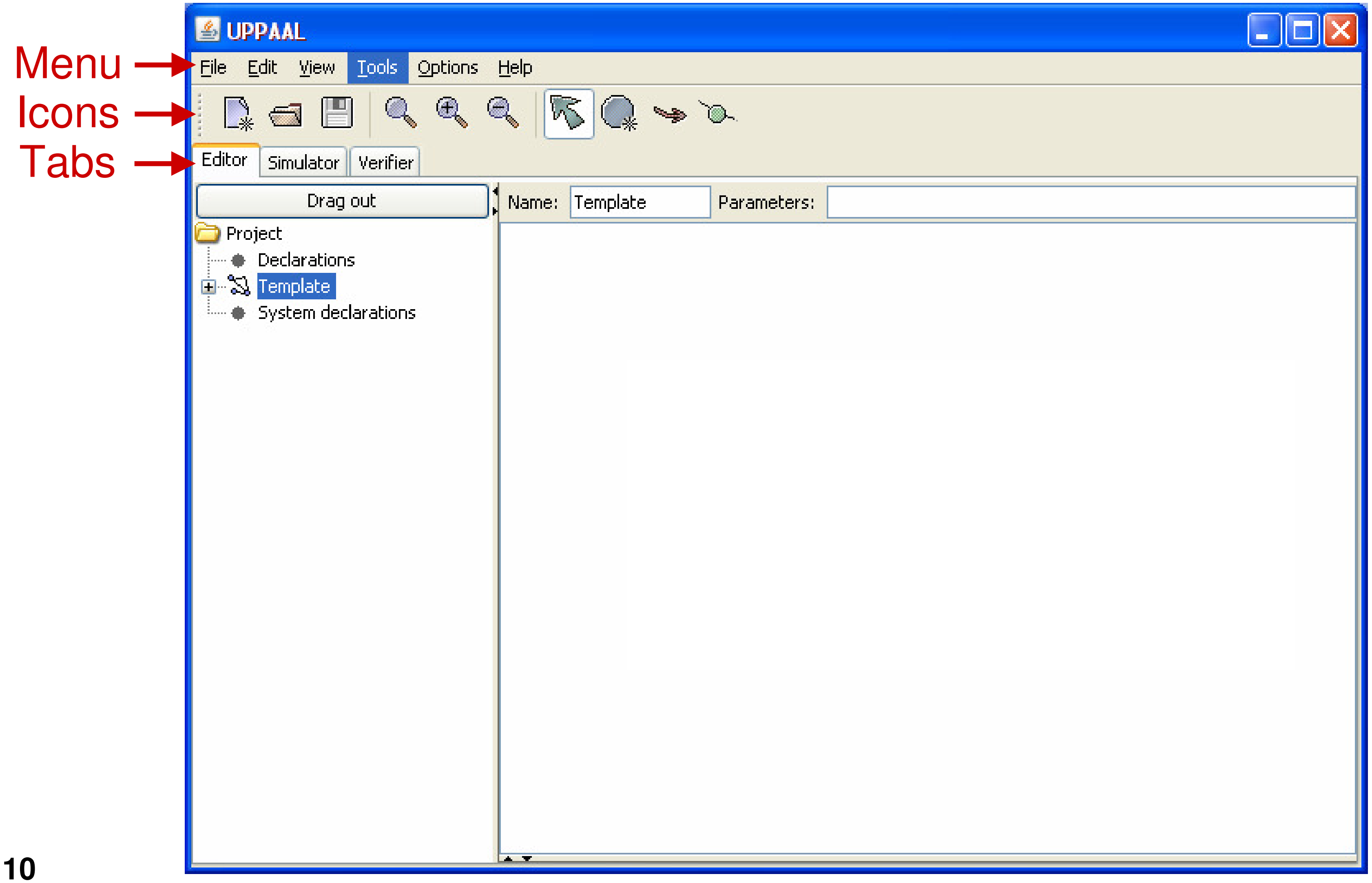
Introducing UPPAAL (v4.0.6)…

- **UPPAAL is a tool box for *validation*** (via graphical simulation) **and *verification*** (via automatic model-checking) of real-time systems.

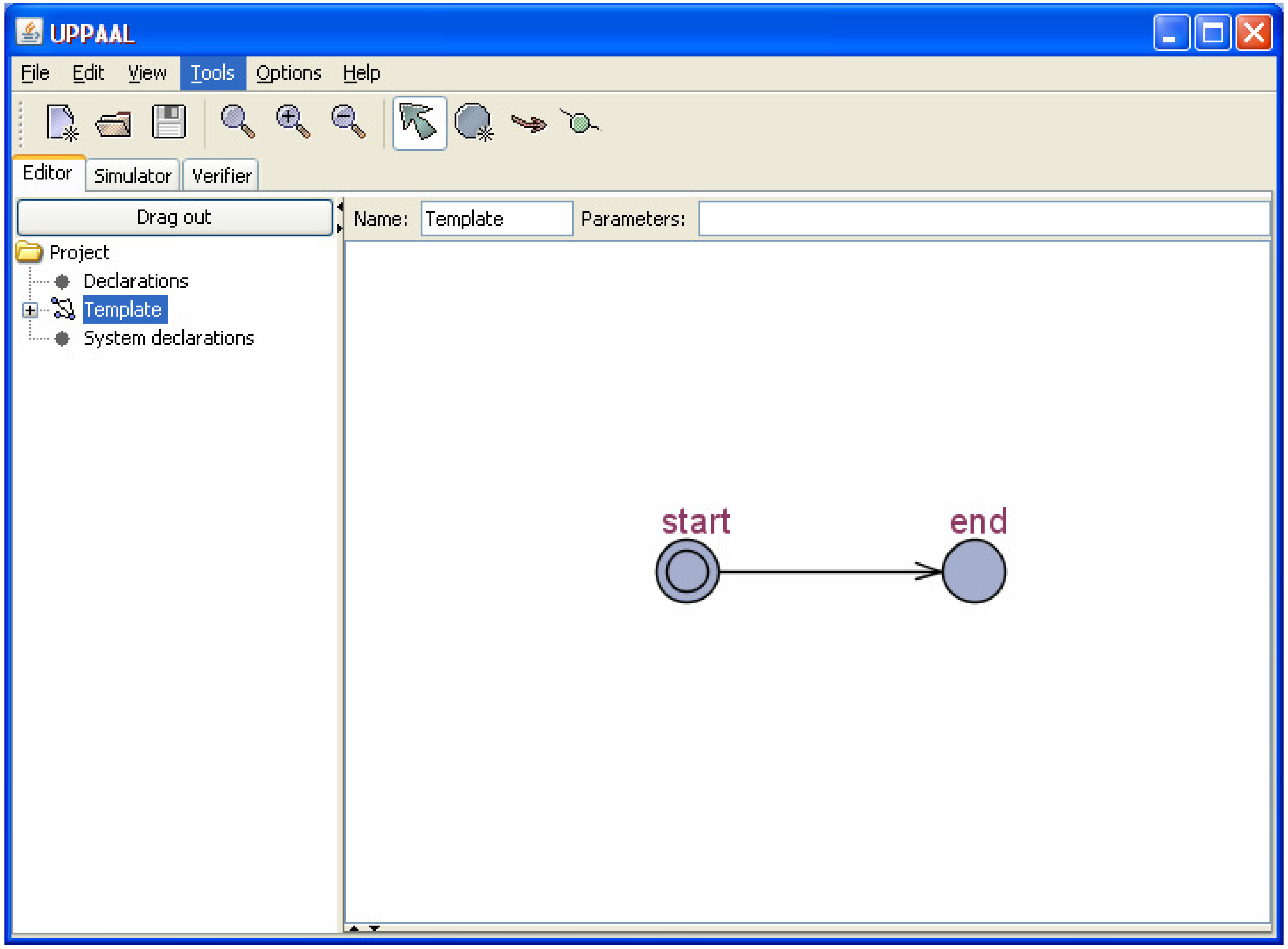
- It consists of **two main parts**:
  - a **Graphical User Interface** (GUI) (executed on the users work station) and
  - a **model-checker engine** (by default executed on the same computer as the user interface, but can also run on a more powerful server)

- It has been jointly developed by **Upp**sala University in Sweden and **Aal**borg University in Denmark

An overview of the tool

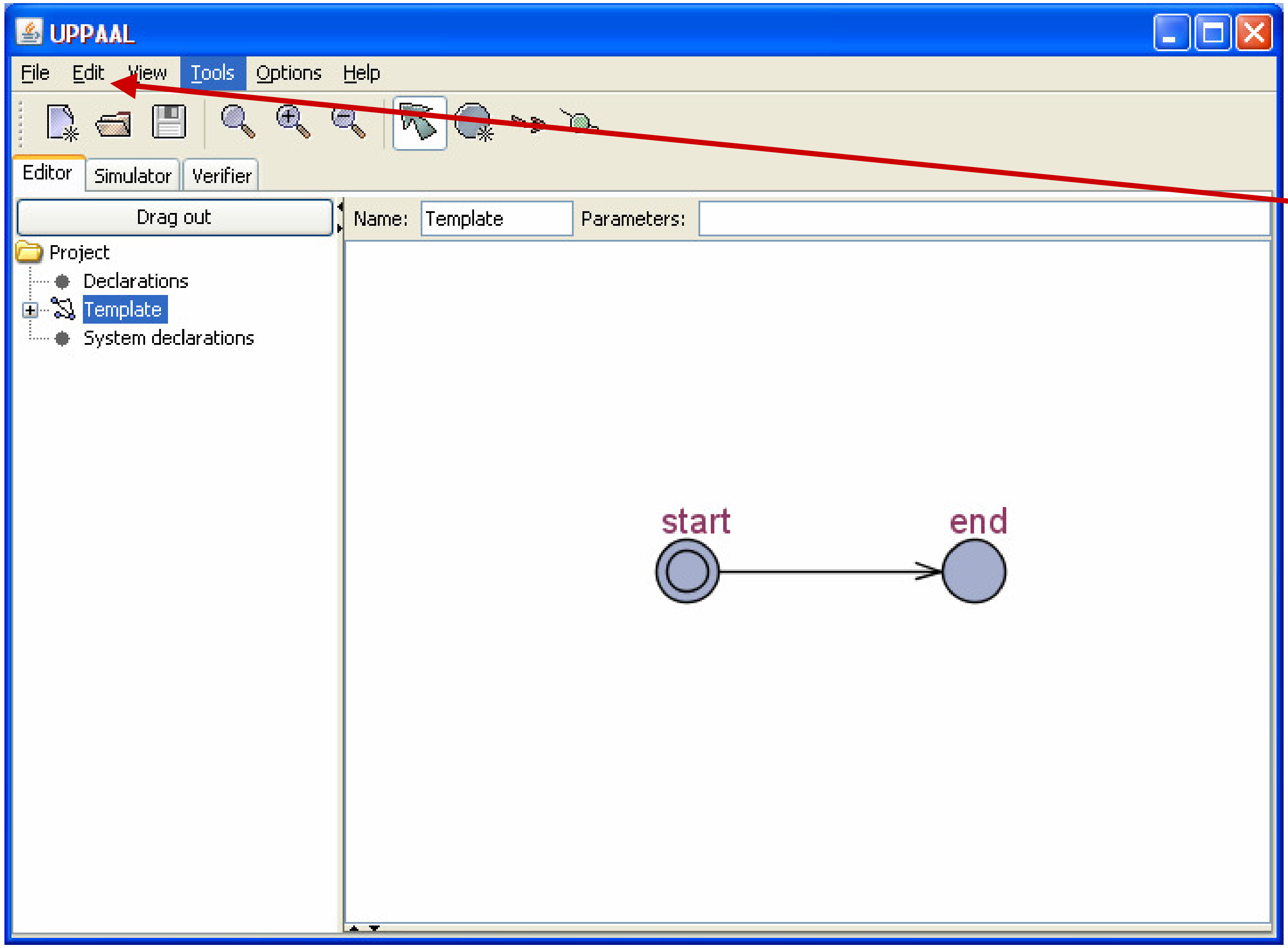


An overview of the tool
The Editor Window



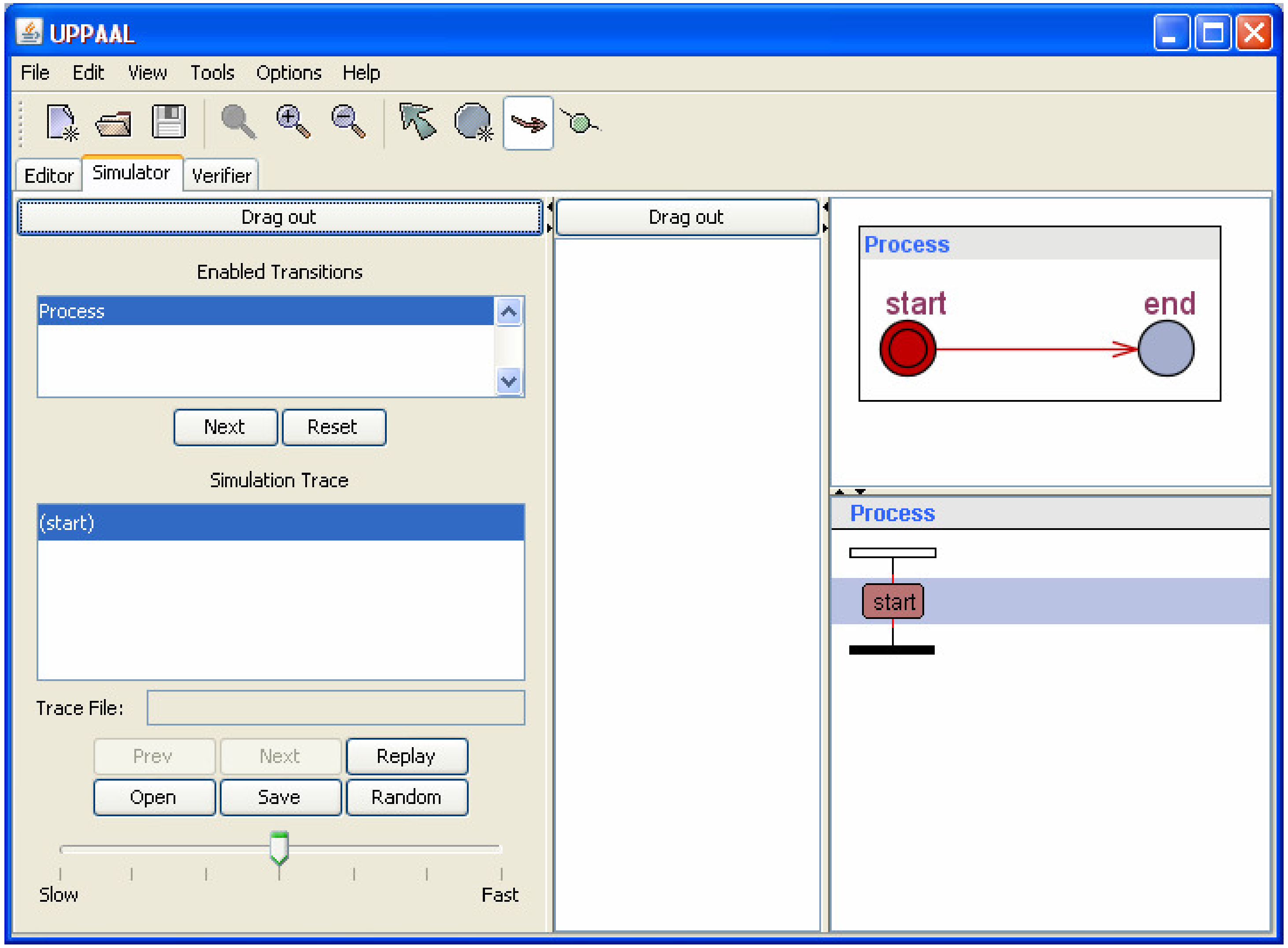
Name: Template
Parameters:

An overview of the tool
The Editor Window

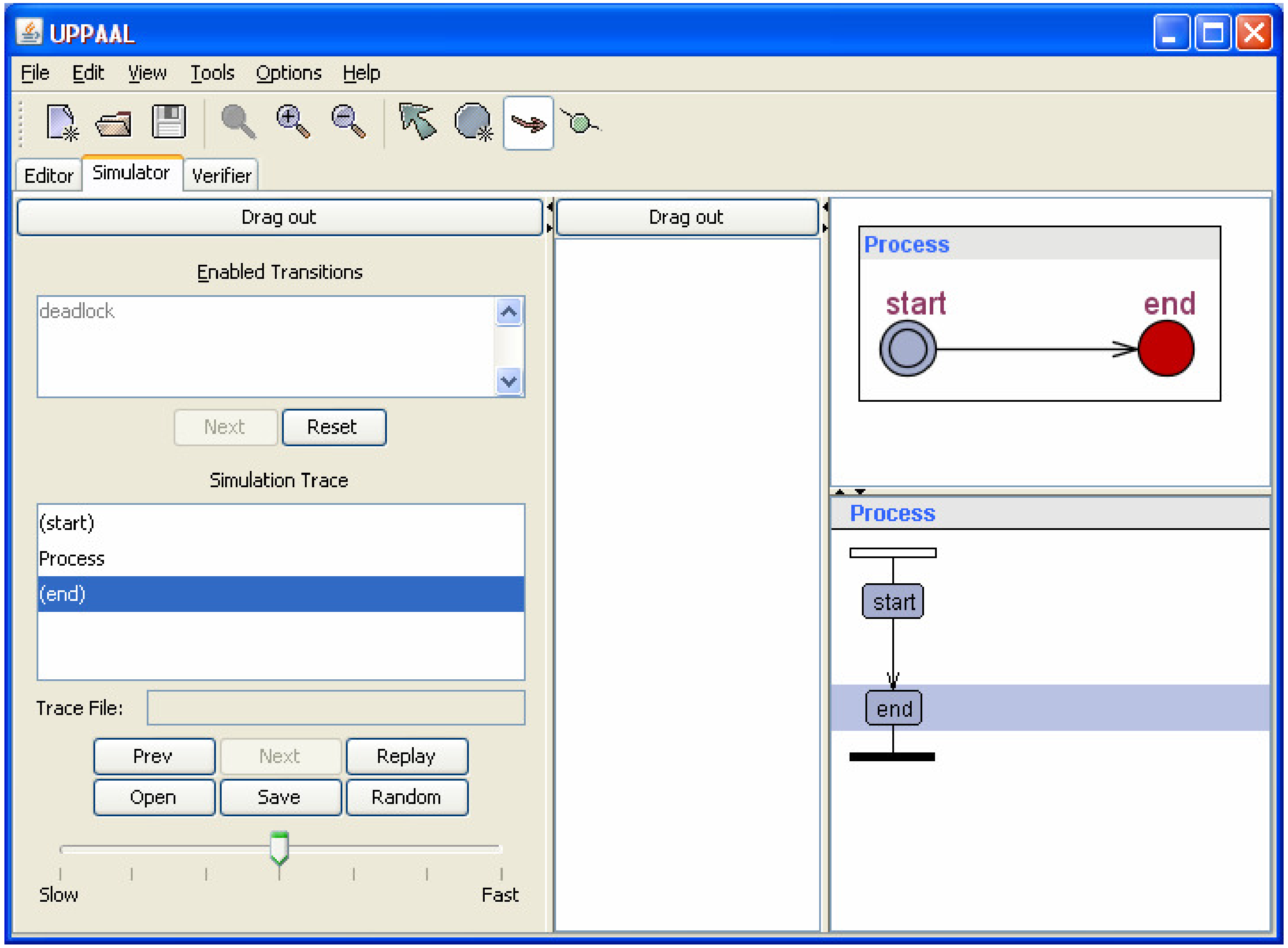


With "undo"!

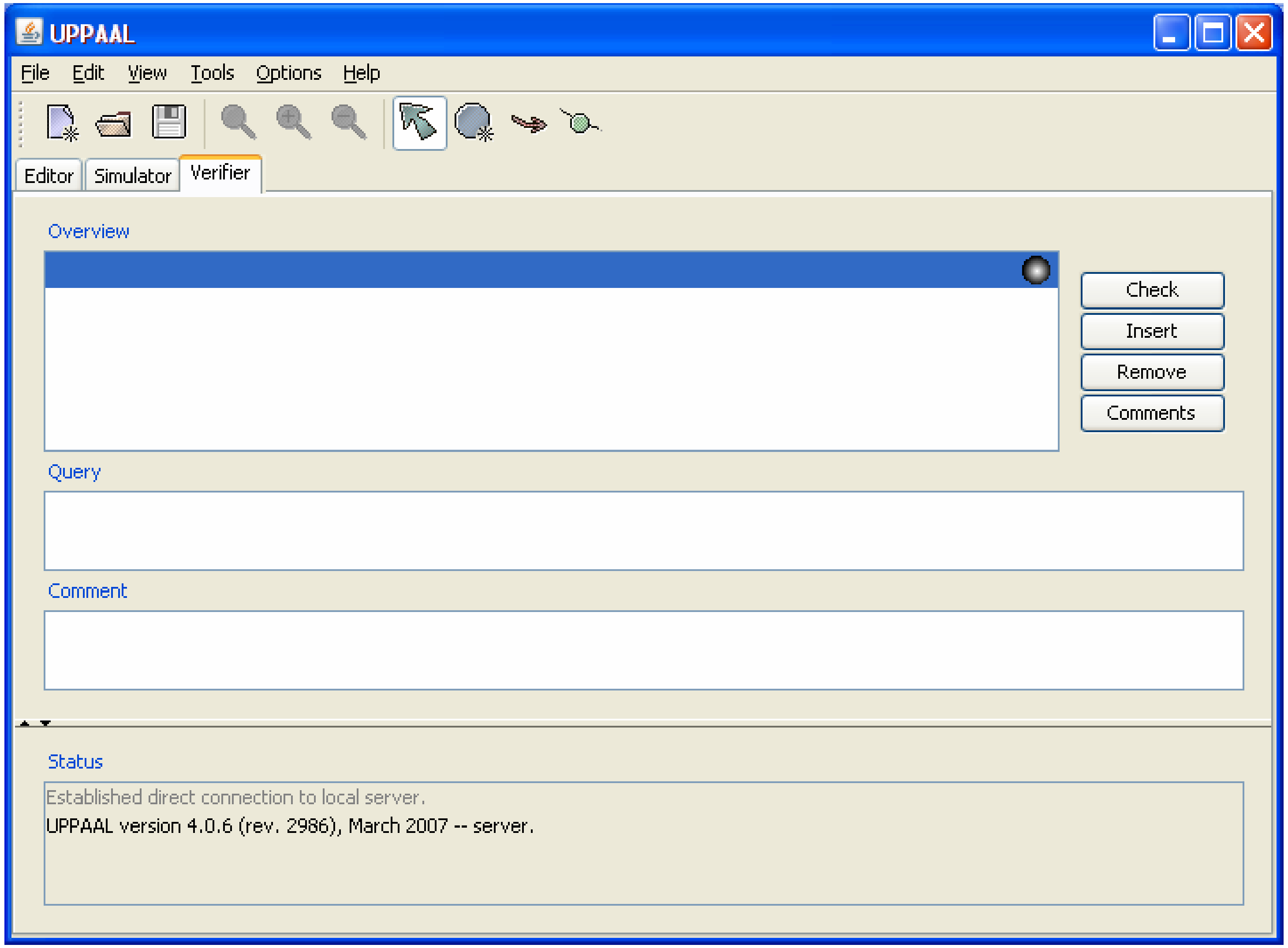
An overview of the tool
The Simulator Window (1)



An overview of the tool
The Simulator Window (2)

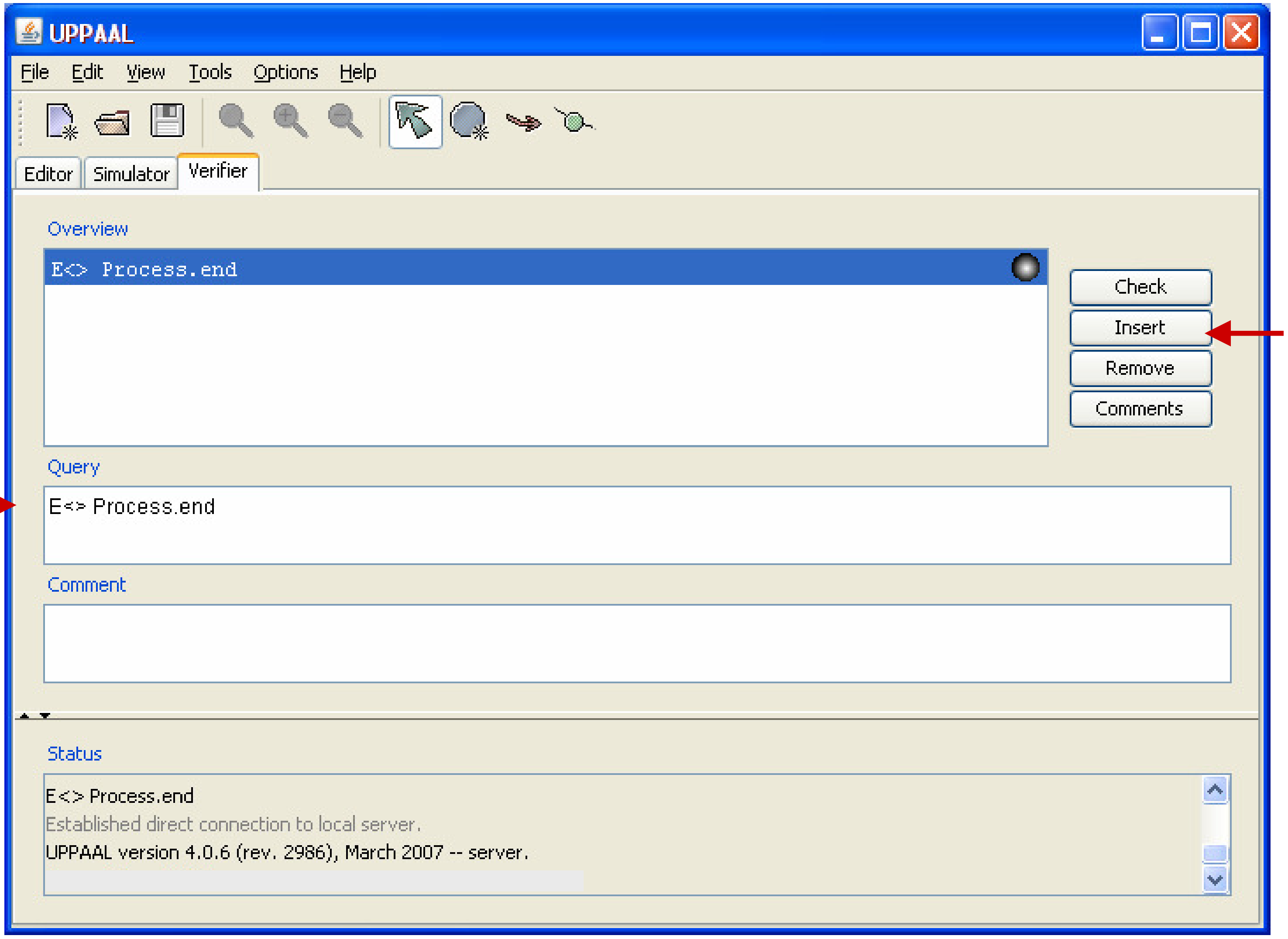


An overview of the tool
The Verifier Window (1)



Established direct connection to local server.
UPPAAL version 4.0.6 (rev. 2986), March 2007 -- server.

An overview of the tool
The Verifier Window (2)



```
E<> Process.end
```

Query:

```
E<> Process.end
```

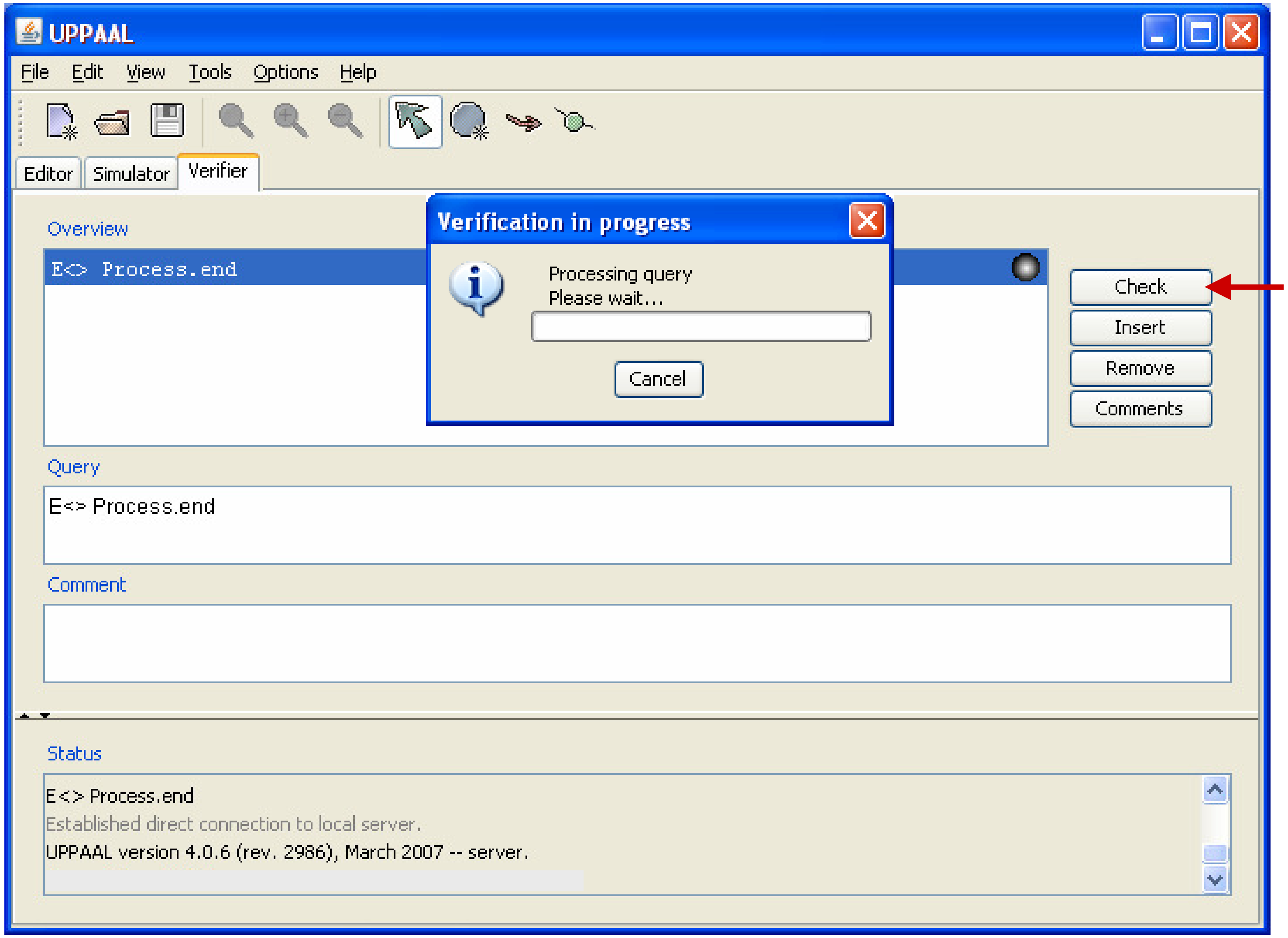
Status:

```
E<> Process.end
Established direct connection to local server.
UPPAAL version 4.0.6 (rev. 2986), March 2007 -- server.
```

1. Click

2. Edit

An overview of the tool
The Verifier Window (3)



```
E<> Process.end
```

Processing query
Please wait...

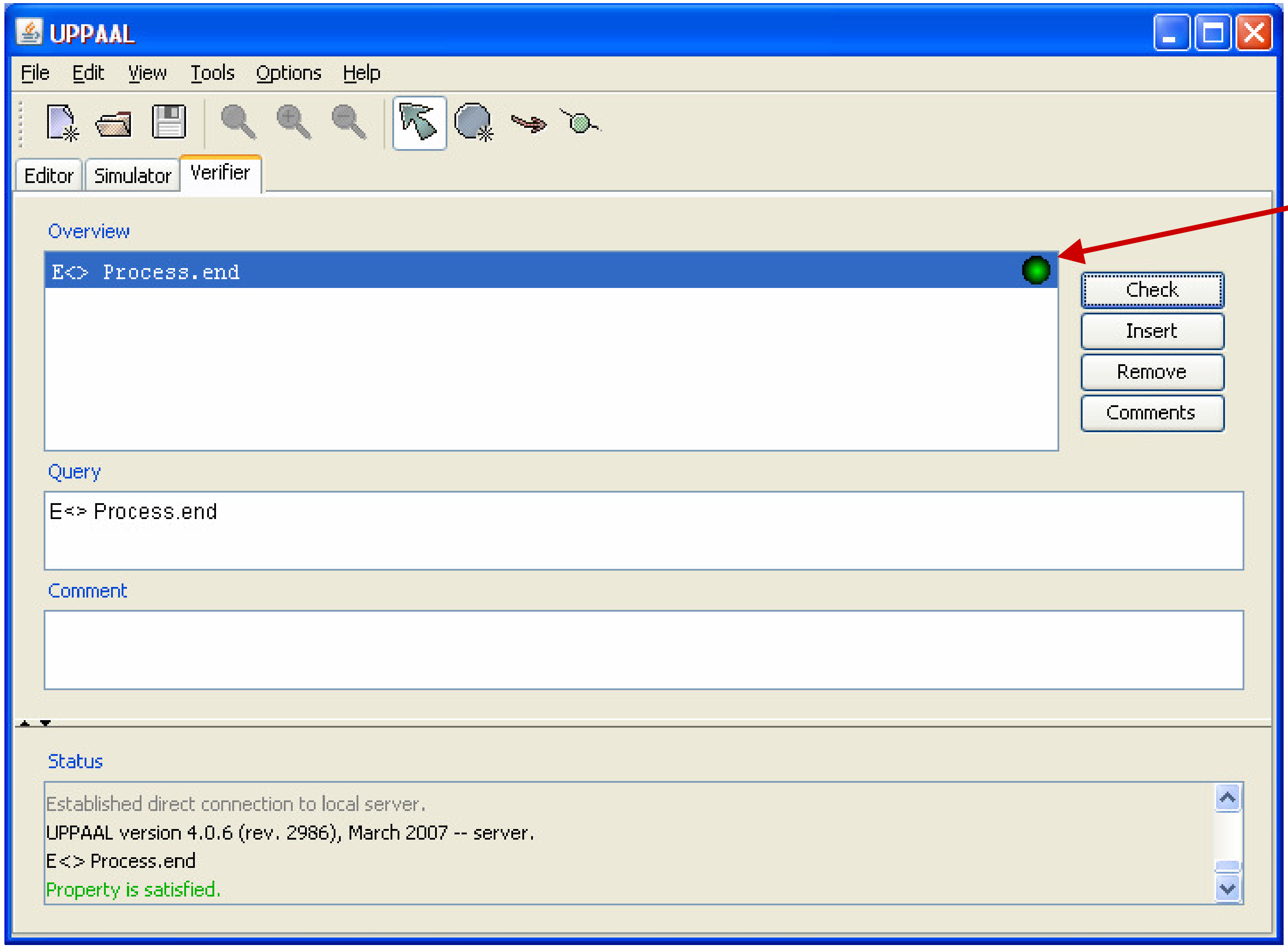
Cancel

E<> Process.end

E<> Process.end
Established direct connection to local server.
UPPAAL version 4.0.6 (rev. 2986), March 2007 -- server.

Click

An overview of the tool
The Verifier Window (4)



```
E<> Process.end
```

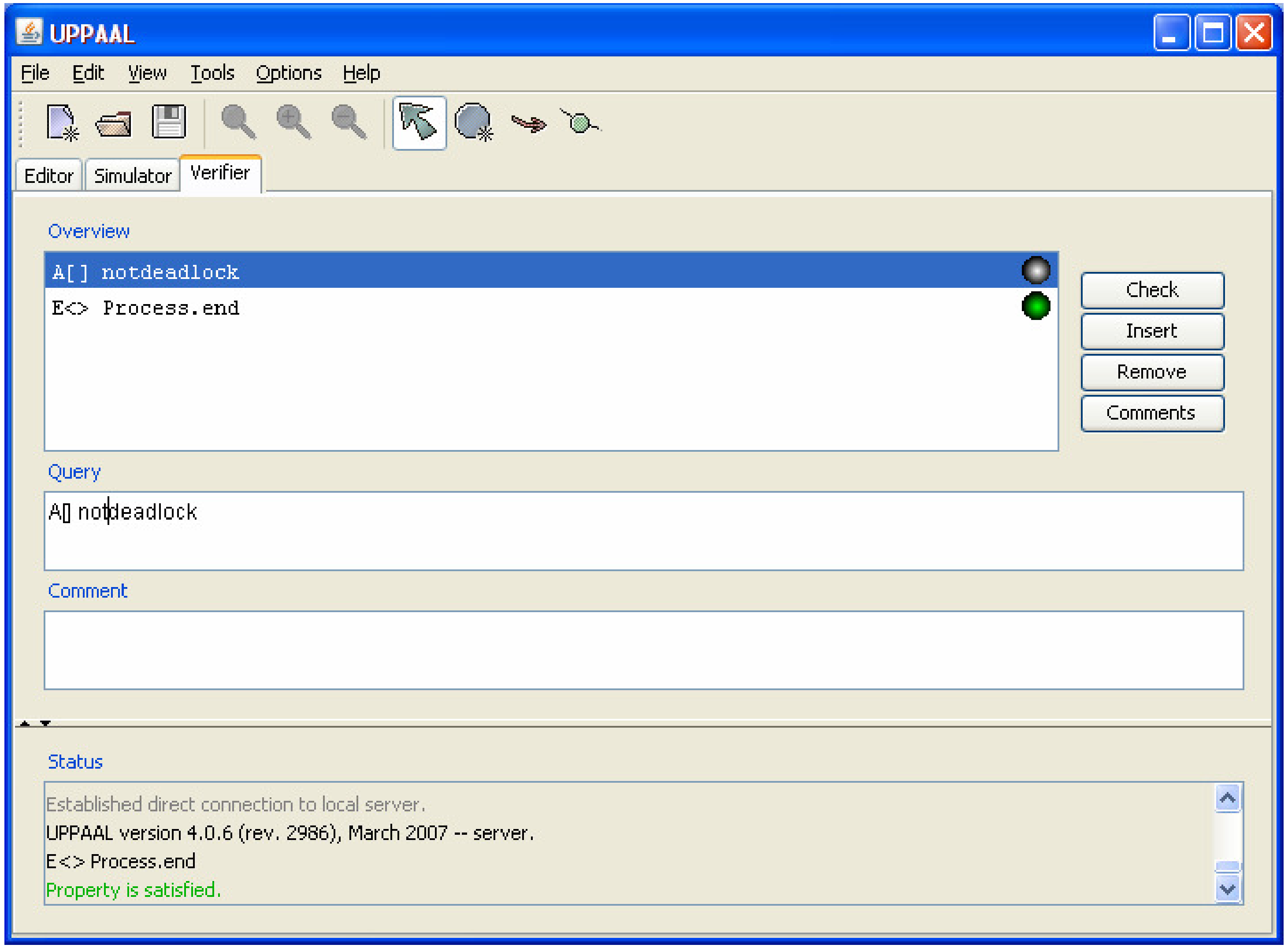
Query: E<> Process.end

Status:

```
Established direct connection to local server.
UPPAAL version 4.0.6 (rev. 2986), March 2007 -- server.
E<> Process.end
Property is satisfied.
```

Green light

An overview of the tool
The Verifier Window (5)

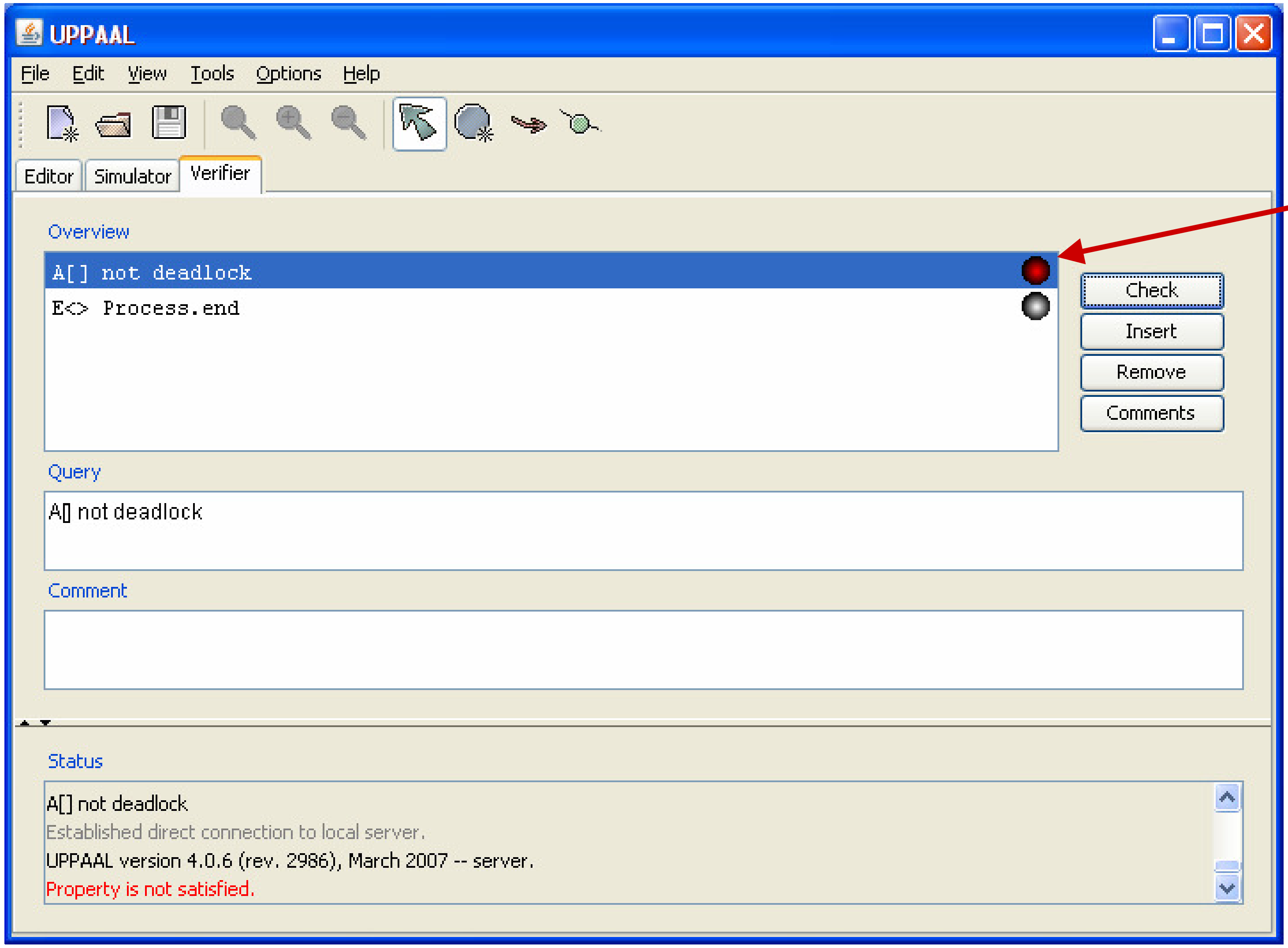


```
A[] notdeadlock
E<> Process.end
```

Query: A[] notdeadlock

```
Established direct connection to local server.
UPPAAL version 4.0.6 (rev. 2986), March 2007 -- server.
E<> Process.end
Property is satisfied.
```

An overview of the tool
The Verifier Window (6)



```
A[] not deadlock
E<> Process.end
```

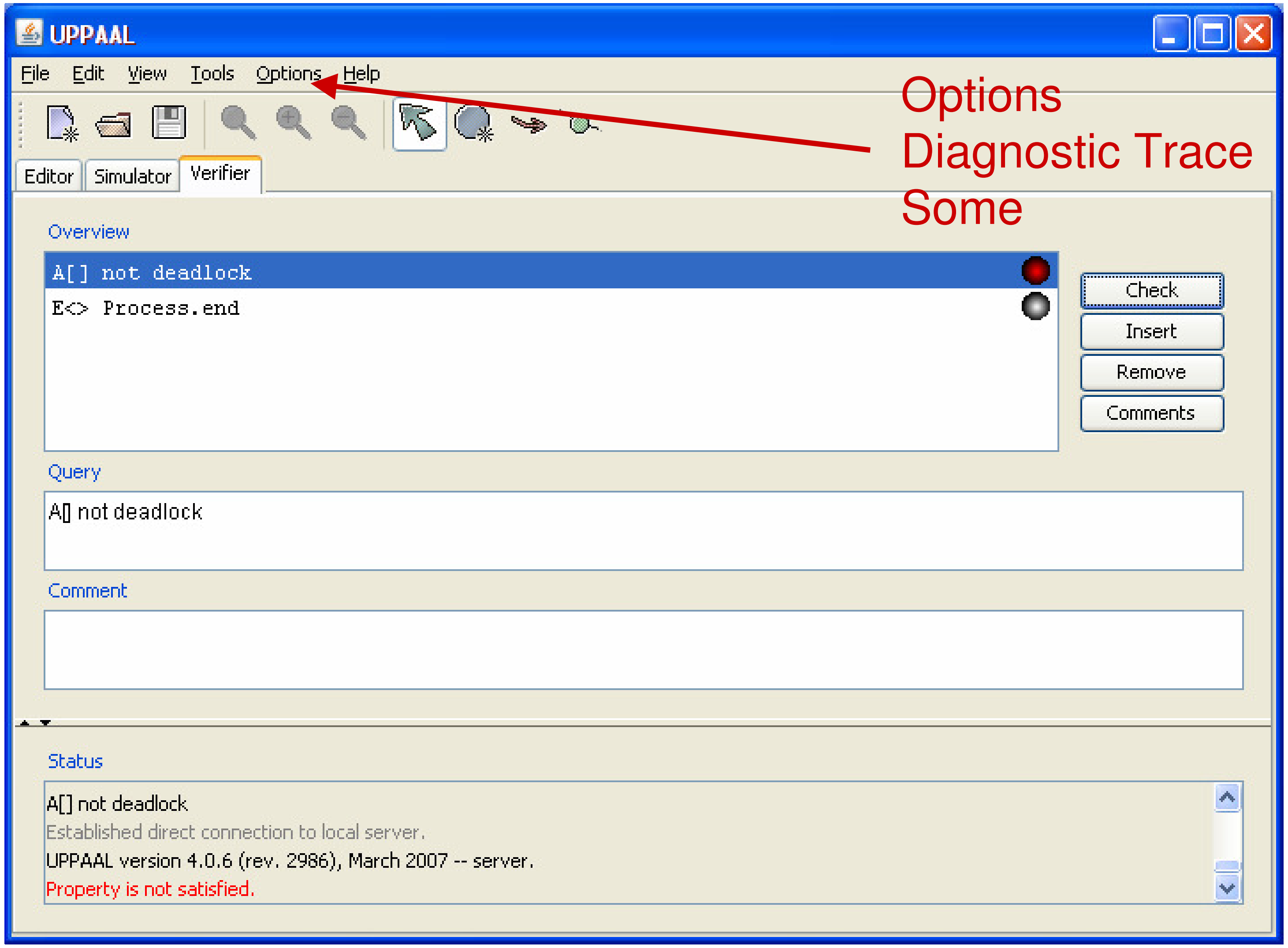
Query: A[] not deadlock

Status:

A[] not deadlock
Established direct connection to local server.
UPPAAL version 4.0.6 (rev. 2986), March 2007 -- server.
Property is not satisfied.

Red light

An overview of the tool
The Verifier Window (7)



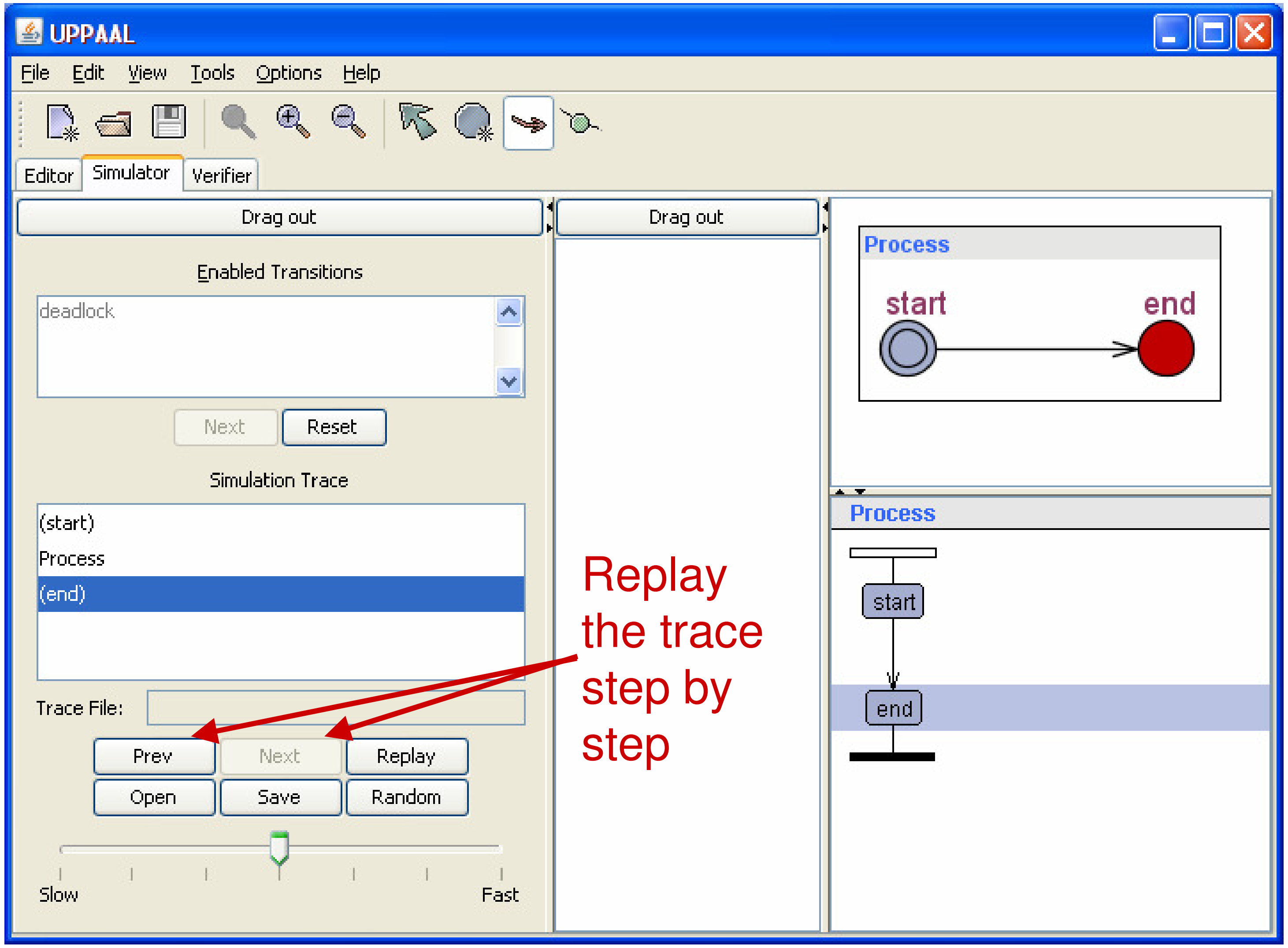
```
A[] not deadlock
E<> Process.end
```

Query: A[] not deadlock

Status:

```
A[] not deadlock
Established direct connection to local server.
UPPAAL version 4.0.6 (rev. 2986), March 2007 -- server.
Property is not satisfied.
```

An overview of the tool
The Stored Trace (counterexample)



Replay the trace step by step

Presentation Outline

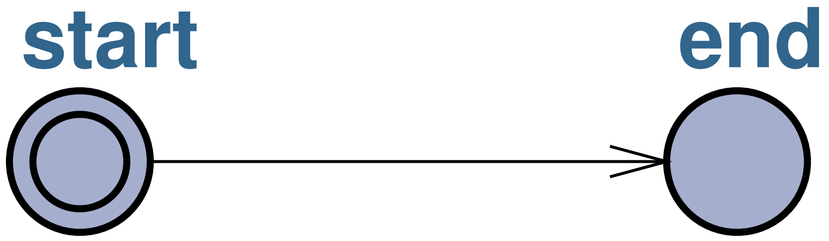
Modeling with UPPAAL

- In UPPAAL, **systems are modeled using *timed-automata***, which are finite state machines with *clocks*.
  – Clocks are variables which can evaluate to a **real number** and which can be defined in each automaton in order to measure the time progress.
  – All clocks evolve at the **same pace** in order to represent the global progress of time.
  – The actual value of a clock can be either tested or reset (**not assigned**).

- Given that UPPAAL is specially designed for the **verification of real-time systems**, clocks are a fundamental modeling and verification feature.

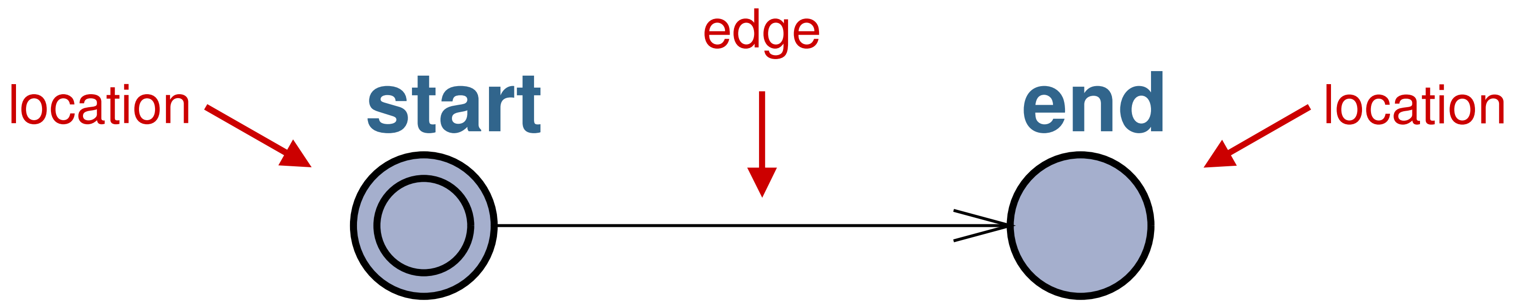
Structure of an UPPAAL Model

- An UPPAAL model is built as **a set of concurrent *processes***.
- Each process is graphically designed as a ***timed-automaton*.**
- Since instantiations of the same automaton are frequently needed ***templates*** are used
- A timed-automaton is represented as a graph which has ***locations*** as nodes and ***edges*** as arcs between locations.



Structure of an UPPAAL Model

- An UPPAAL model is built as **a set of concurrent *processes***.
- Each process is graphically designed as a ***timed-automaton***.
- Since instantiations of the same automaton are frequently needed ***templates*** are used
- A timed-automaton is represented as a graph which has ***locations*** as nodes and ***edges*** as arcs between locations.



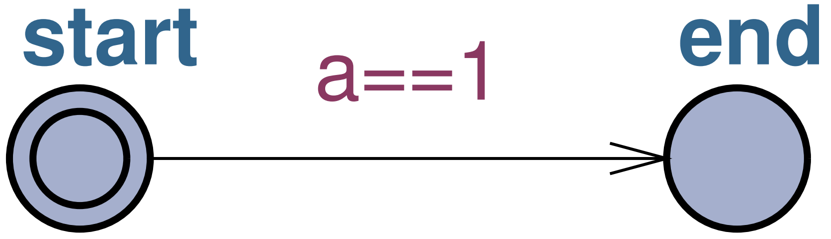
Labels in Edges

- Edges are annotated with ***guards***, ***updates***, ***synchronisations*** and ***selections***

Labels in Edges
Guards

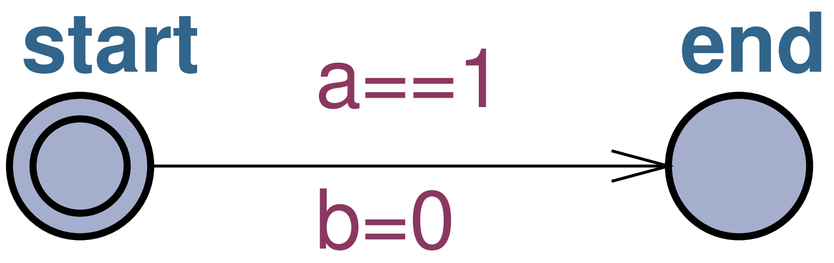
- Edges are annotated with ***guards***, ***updates***, ***synchronisations*** and ***selections***

- **A guard is** an expression which uses the variables and clocks of the model in order to indicate when the transition is enabled, i.e. may be fired.
  - Note that several edges may be enabled at an specific time but only one of them will be fired → leading to different potential **interleavings**



Labels in Edges
Updates

- **An update is** an expression that is evaluated as soon as the corresponding edge is fired. This evaluation changes the state of the system.



Labels in Edges
Synchronisations

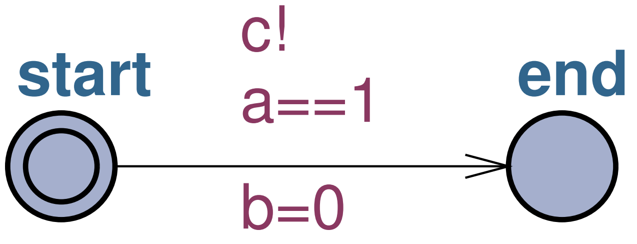
- **The *synchronization* is** the basic mechanism used to coordinate the action of two or more processes. Models for instance the effect of messages

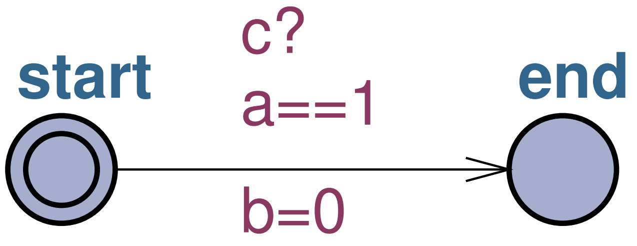
- It causes two (or more) processes to take a transition at the same time. A **channel** (*c*) is declared, then one process will have an edged annotated with *c!* and the other(s) process(es) another edge annotated with *c?*

- **Three different kinds of synchronizations**:
  - **Regular channel** (leading to Binary Synchronization)
  - **Urgent channel**
  - **Broadcast channel**

Labels in Edges
Synchronisations: Regular Channel

- A **regular channel** is declared as, e.g., `chan c`.
  - When a process is in a location from which there is a transition labelled with *c!* the only way for the transition to be enabled is that another process is in a location from which there is a transition labelled with *c?* and vice versa.
  - If at a specific instant there are several possible ways to have a pair *c!* and *c?*, one of them is non-deterministically chosen during model checking.

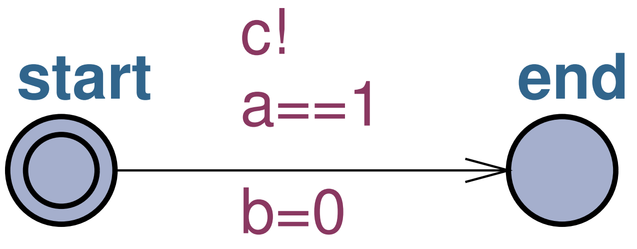


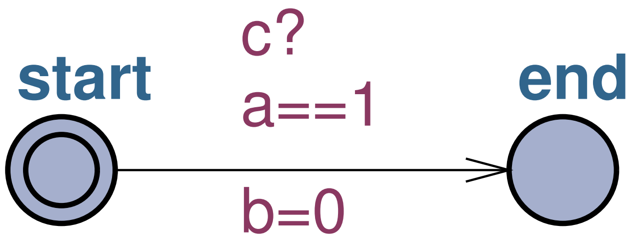


  - The update expression on an edge synchronizing on *c!* is executed **before** the update expression on an edge synchronizing on *c?*

Labels in Edges
Synchronisations: Urgent Channel

- An **urgent channel** is declared as `urgent chan c`.
  - Urgent channels are similar to regular channels, except that it is not possible to delay in the source state if it is possible to trigger a synchronisation over an urgent channel.
  - This means no time can pass but they can interleave with other transitions that require no time to pass.
  - Graphically they look like regular channels

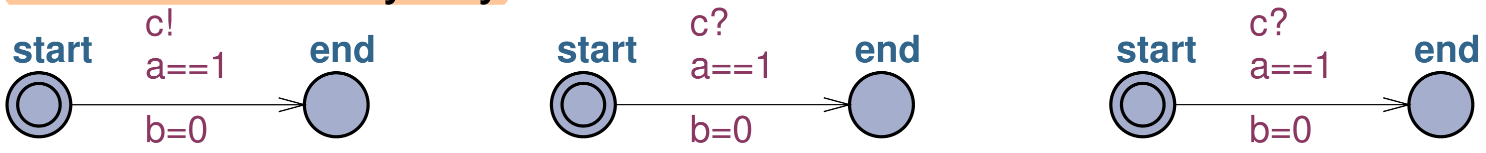




  - **Notice** that clock guards are not allowed on edges synchronising over urgent channels

Labels in Edges
Synchronisations: Broadcast Channel

- For a **broadcast channel**: `broadcast chan c.`
  - When one process is in a location from which there is a transition labelled with *c!* and one **or more** processes are in locations from which there is a transition labelled with *c? all* these transitions are enabled.
  - However, if there are no processes in locations from which there is a transition labelled with *c?,* the transition labelled with *c!* is enabled anyway.



  - **Notice** that clock guards are not allowed on edges **receiving** on a broadcast channel.
  - The update on the emitting edge is executed first. The update on the receiving edges are executed left-to-right in the order the processes are given in the **system definition**.

Locations

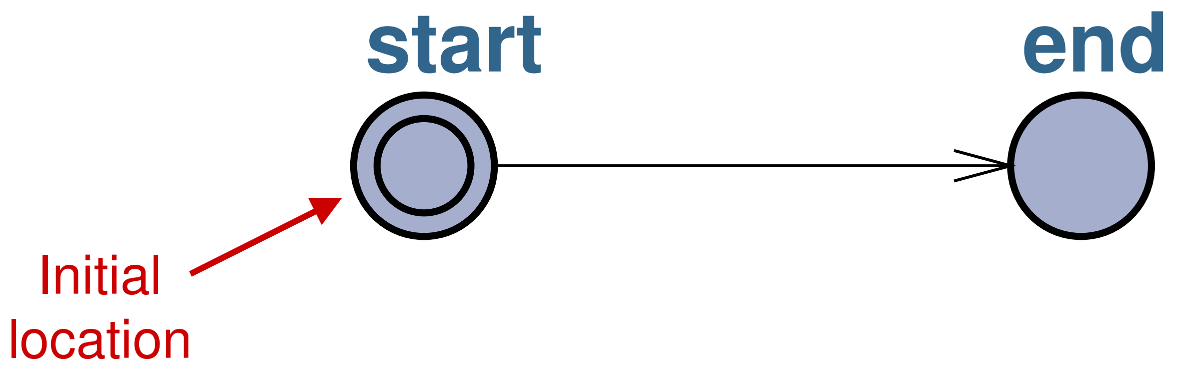
- **Locations** can have an optional **name** (as seen in previous slides)
  - Names are useful to refer to the location during model checking and when documenting the model

- **Locations can be of three different types** (that can be assigned by double-clicking on the location):
  - **Initial**
  - **Urgent**
  - **Committed**
  - **Normal** (all the rest)

Location Types
Initial

- Each template must be properly initialized, meaning it must start in a specific location. Therefore, each template must have exactly one location marked as ***initial***.

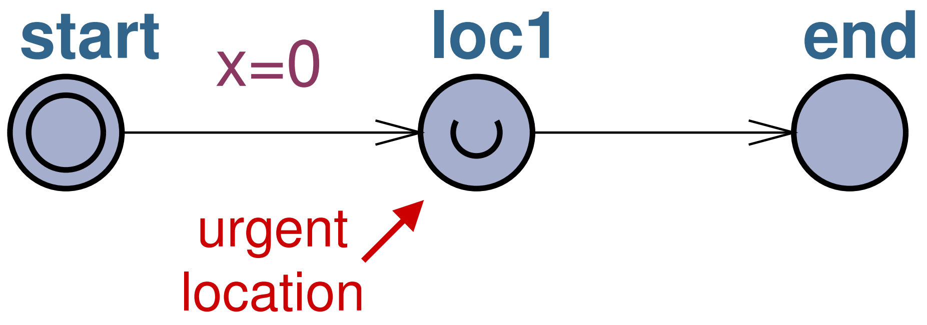
- Initial locations are identified with a double circle



Location Types
Urgent

- **Urgent locations** freeze time; i.e. time is not allowed to pass when a process is in an urgent location.
  - The location must be leaved before time could pass
  - Other transitions may happen before, as long as they do not require time to pass

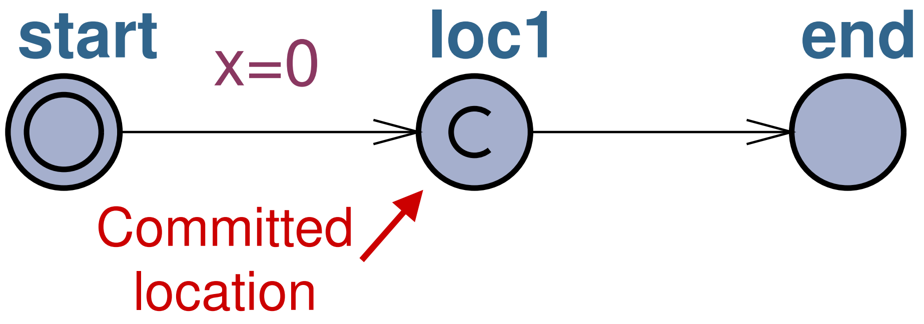
- Urgent locations are identified with a "U"



- Clock x will not increase as long as in loc1

Location Types
Committed

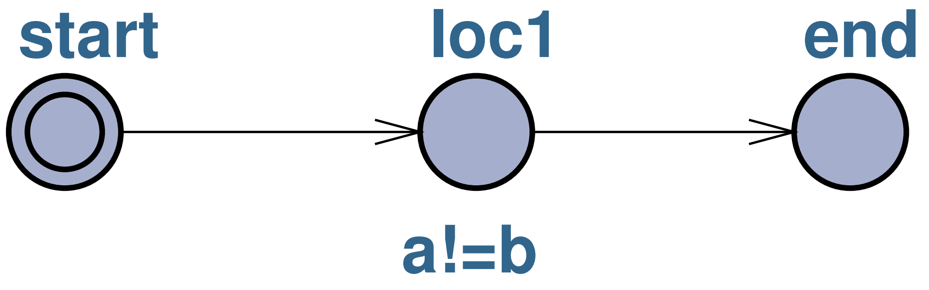
- **Committed locations** also freeze time; i.e. time is not allowed to pass when a process is in one of them.
  - When a model has one or more active committed locations, no transitions other than those leaving said locations can be enabled.
  - Notice that if several processes are in a committed location at the same time, then they will interleave.
  - Committed locations are useful for creating atomic sequences
- Committed locations are identified with a “C”

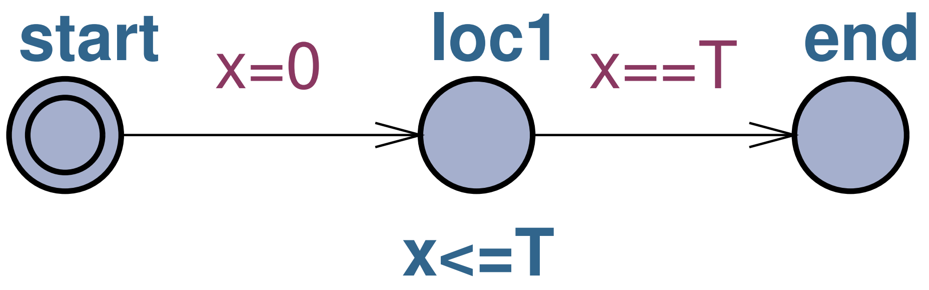


- Clock x will not increase as long as in loc1

Invariants

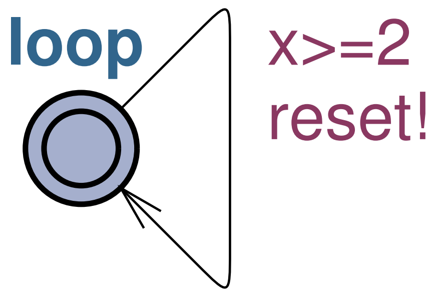
- Both initial and normal locations can have **invariants**.
- Invariants are conditions that must be fulfilled while the automaton is in that location.
- They can be related to **variables** and **clocks**.

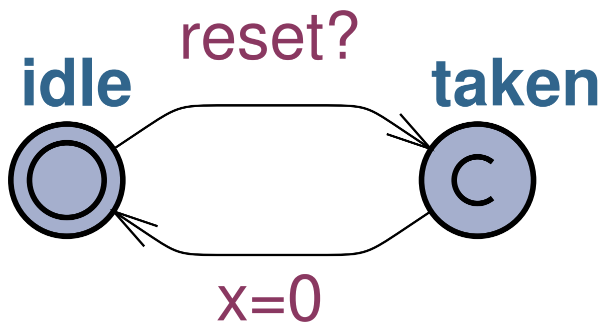




Invariants on clocks and time (1)

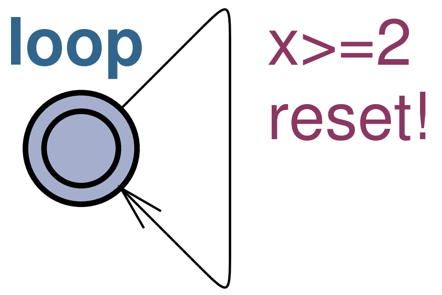
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **First** attempt **without an invariant**
  - Note that the "**observer**" allows the observation of the clock value at different instants (clearer when completing the tutorial)

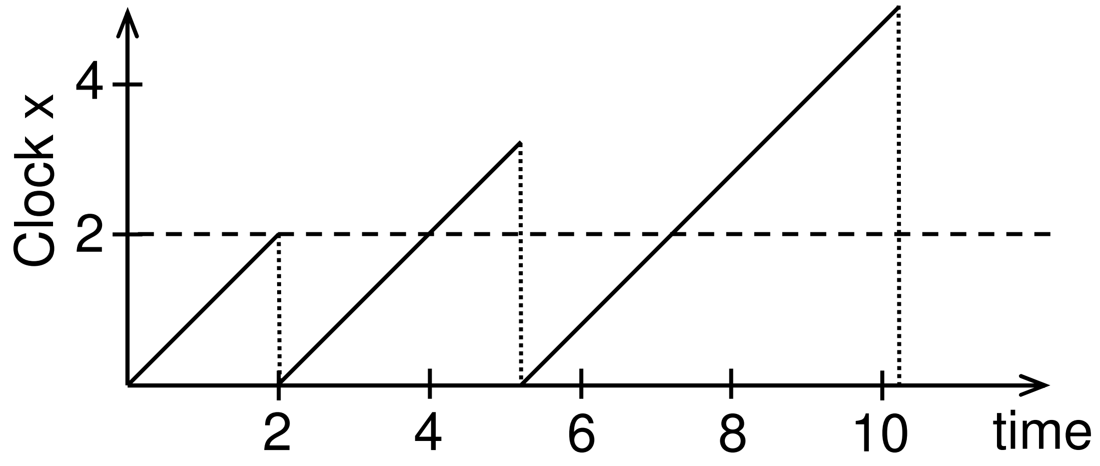




Invariants on clocks and time (1)

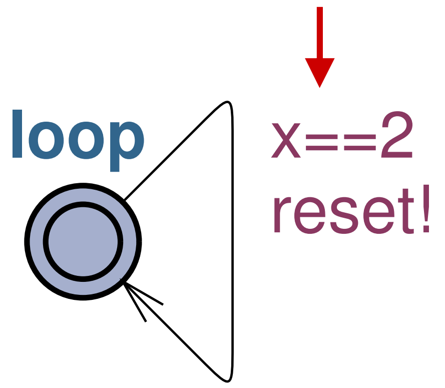
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **First** attempt **without an invariant**
  - Note that the "**observer**" allows the observation of the clock value at different instants (clearer when completing the tutorial)

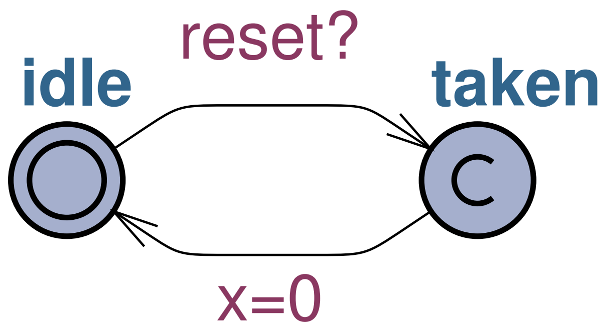




Invariants on clocks and time (2)

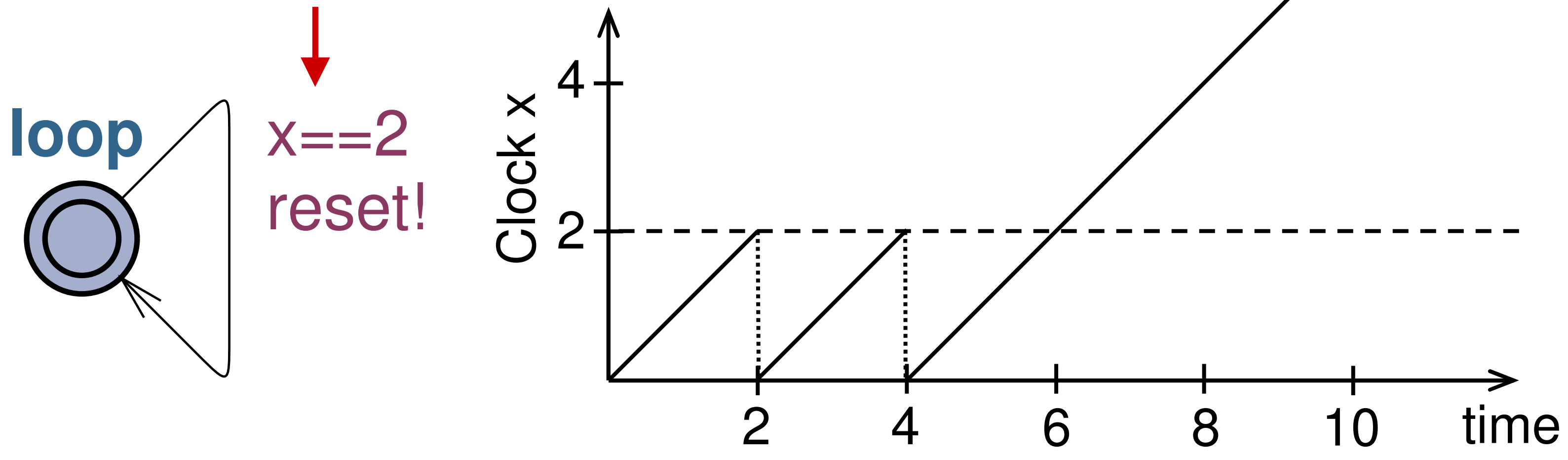
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **Second** attempt **without an invariant**





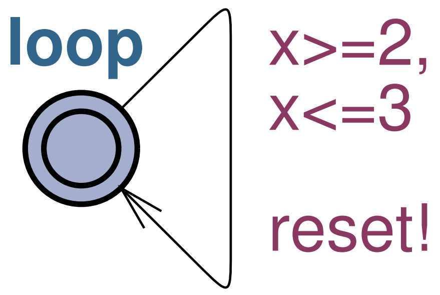
Invariants on clocks and time (2)

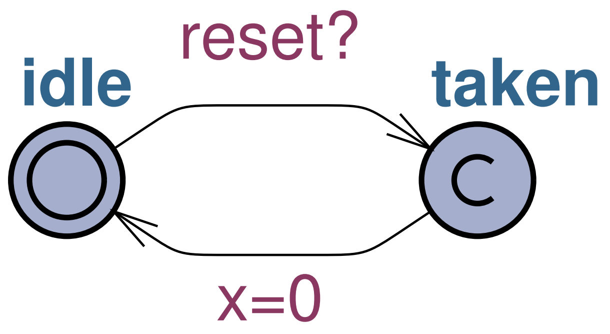
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **Second** attempt **without an invariant**



Invariants on clocks and time (3)

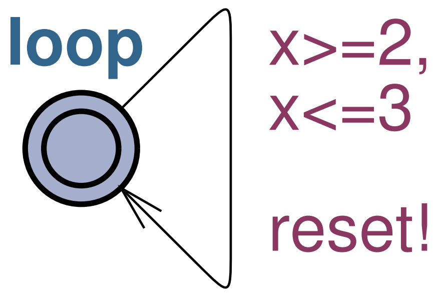
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **Third** attempt **without an invariant** (that will obviously have the same problem!)

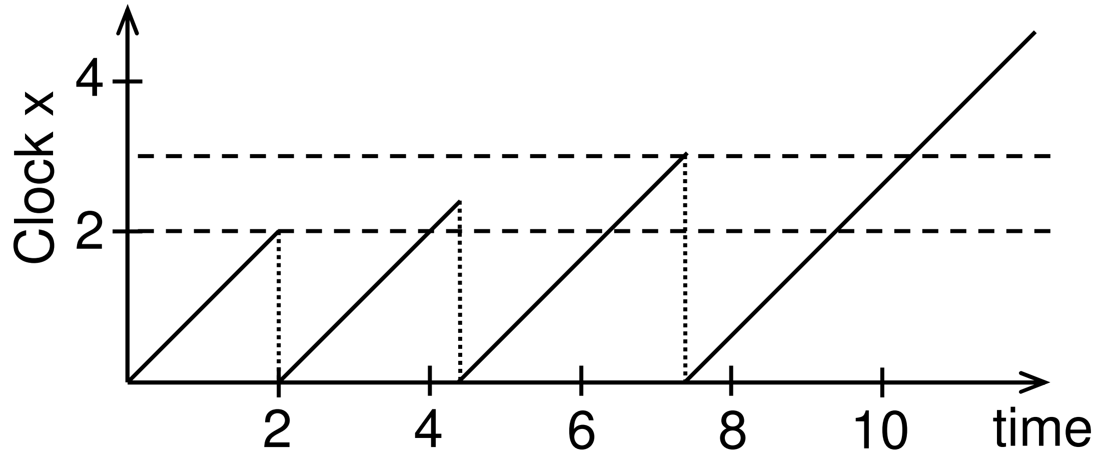




Invariants on clocks and time (3)

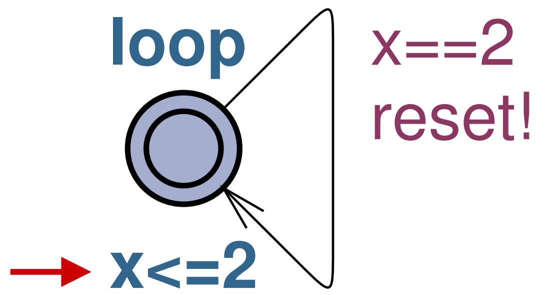
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **Third** attempt **without an invariant** (that will obviously have the same problem!)

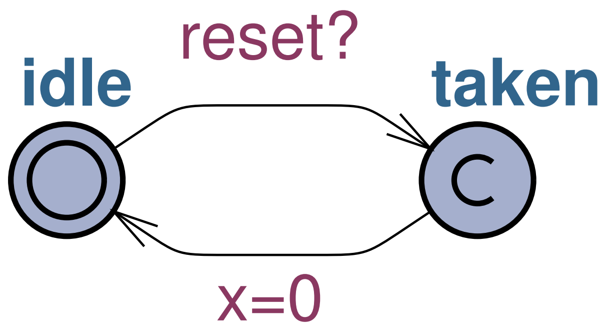




Invariants on clocks and time (4)

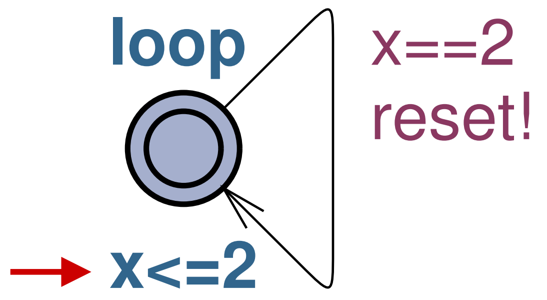
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **Finally with an invariant**

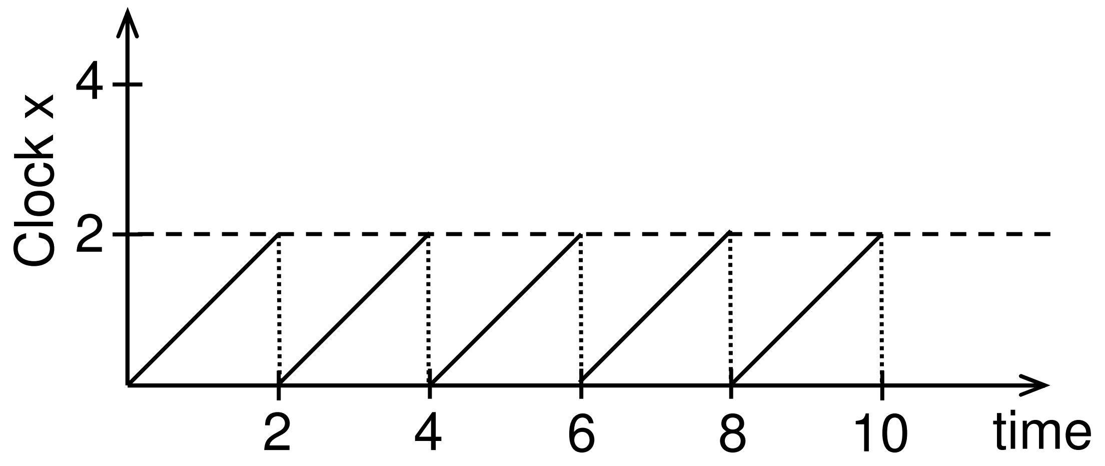




Invariants on clocks and time (4)

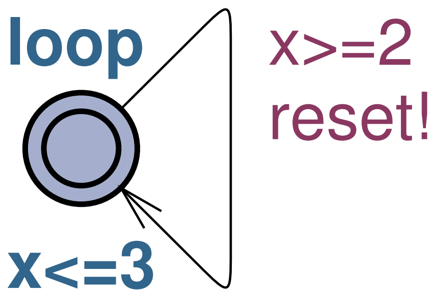
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **Finally with an invariant**

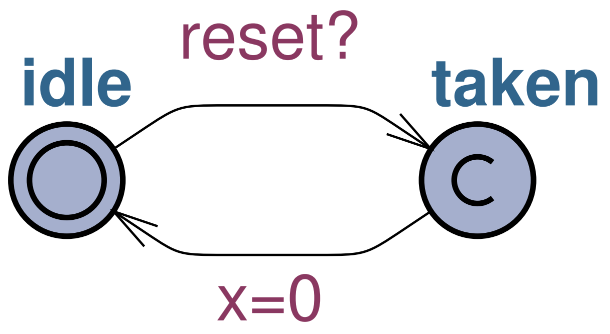




Invariants on clocks and time (5)

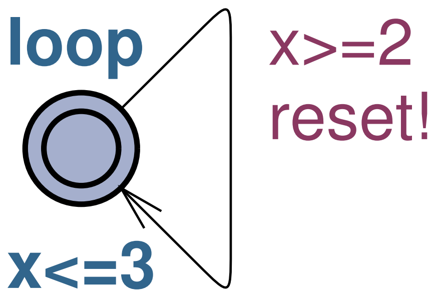
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **A different behaviour with an invariant**

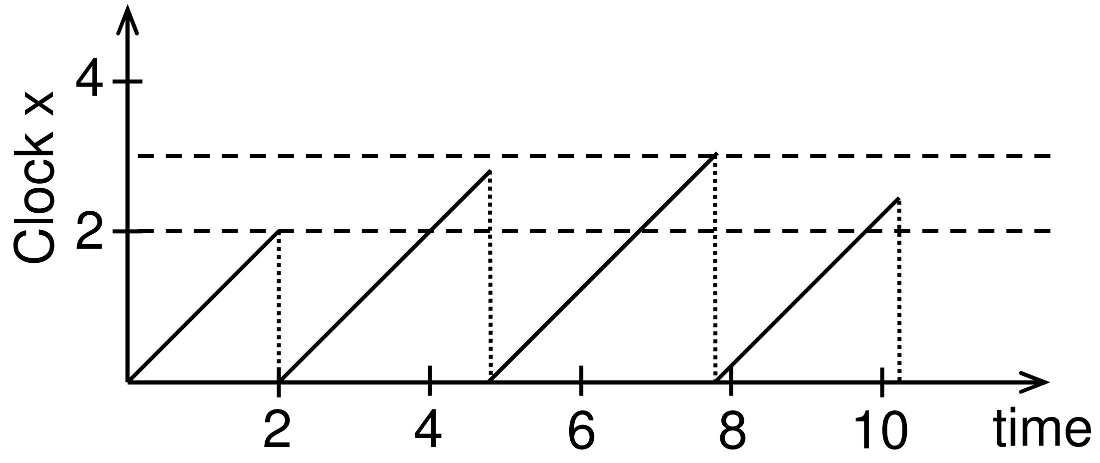




Invariants on clocks and time (5)

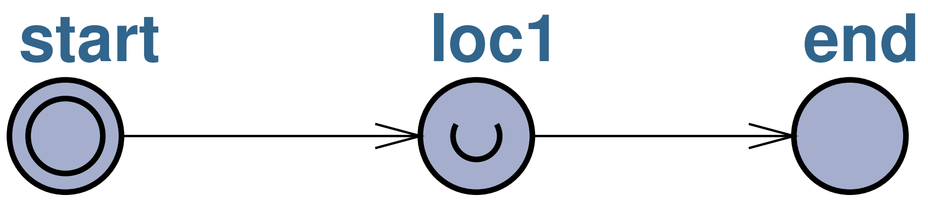
- **To better understand the role invariants on clocks play** let us consider the following **examples** (proposed at the UPPAAL tutorial)
  - The goal is to achieve a similar behaviour (stay in a location until a condition on a clock holds and then leave the location)
  - **A different behaviour with an invariant**

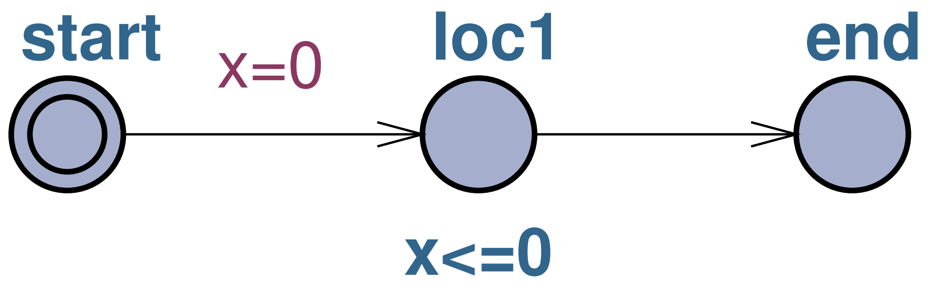




Invariants on clocks and time (6)

- **A final note:** Semantically, urgent locations are equivalent to:
  - adding an extra clock, **`x`**, that is reset on every incoming edge, and
  - adding an invariant **`x <= 0`** to the location.





Some modeling tricks

- They are **often necessary** to actually reproduce the behaviour of our system.

- It is possible to **encode "urgent transitions"** with a guard on a variable, i.e. busy wait on a variable, by using urgent channels.
  - Use a dummy automaton with one state looping with one transition **`read!`**. The urgent transition will be **`x>0 read?`** for example.

- There is no **value passing though the channels** but this is easily encoded by a shared variable:
  - Define globally a variable **`x`**, and use it to write and read it. Notice that it is not clean to do **`read! x:=3;`** and **`read? y:=x;`** but it is better to use a commit state: **`read?`** commit state and **`y:=x;`**.

A recommendation on modeling

- **The state space grows very quickly** with the model complexity (state space explosion). It is necessary to:
  - Find the suitable level of abstraction for the model
  - Model only the features related to the properties to be verified
  - Also omit the features that are “obviously” correct
- More specifically:
  - The number of clocks has an important impact on the complexity
  - The use of committed locations can reduce significantly the state space, but it can possibly take away relevant states.
  - The number of variables is also relevant and even more their range. In particular, avoid unbounded loops on integers since the values will then span over the full range.
- **This is rather an “art”** (so model checking is not so “perfect” as it initially looks but it helps to think)

Presentation Outline

1. The role of Model Checking in design validation

2. The UPPAAL Tool

    1. Introduction
    2. Modeling
    3. Verification
    4. A first example
    5. Installation instructions

3. References

Verifying with UPPAAL

- **After using the simulator** to ensure that the model behaves as the system we wanted to model (and sometimes also to detect some errors in the original design), **the next phase is to check that the model verifies *the* properties**

- Then we need to know/decide what those properties are… and **formalize** them!
  - Ex. from def. of consensus service: *Consistency:* All correct servers agree on the same value and all decisions are final.

- After that, we have to **translate** those properties into the **UPPAAL query language** (not one-to-one)

The UPPAAL Query Language

- The types of properties that can be directly checked using the UPPAAL queries are quite simple.
- UPPAAL designers have taken this approach, instead of allowing complex queries, in order to improve the efficiency of the tool.
- For this reason the verification of complex properties may need the checking of many different queries and even the addition to the model of specifically designed “testing automata”

Types of Queries in UPPAAL

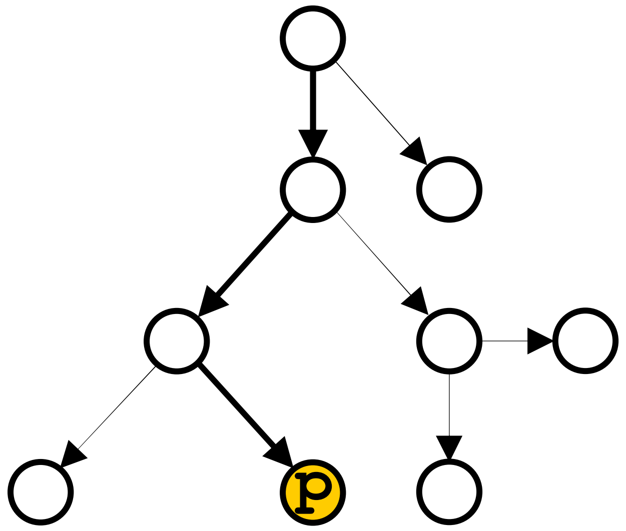
- The specific **types of properties** that can be expressed in the UPPAAL query language can be classified as:
  - ***Reachability properties***. *A specific condition holds in some state of the model's potential behaviours*
  - ***Safety properties***. *A specific condition holds in all the states of an execution path*
  - ***Liveness properties***. *A specific condition is guaranteed to hold eventually (= at some moment)*
  - ***Deadlock properties***. *A deadlock is possible or not in the model*

UPPAAL Reachability properties

*A specific condition **holds in some state** of the model's potential behaviours*

- They are always **expressed in the form:**

`E<> p` "Exists eventually p" meaning there is an execution path in which p eventually (in some state of the path) holds



UPPAAL Reachability properties

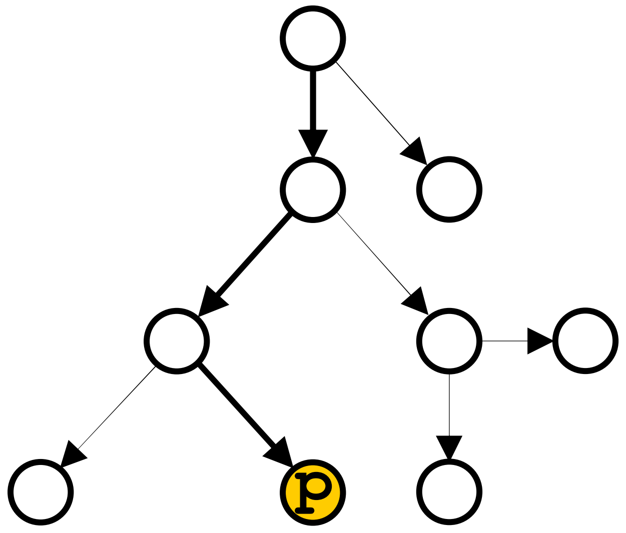
*A specific condition **holds in some state** of the model's potential behaviours*

- They are always **expressed in the form**:

Some path

Some state

`E<> p` "Exists eventually p" meaning there is an execution path in which p eventually (in some state of the path) holds

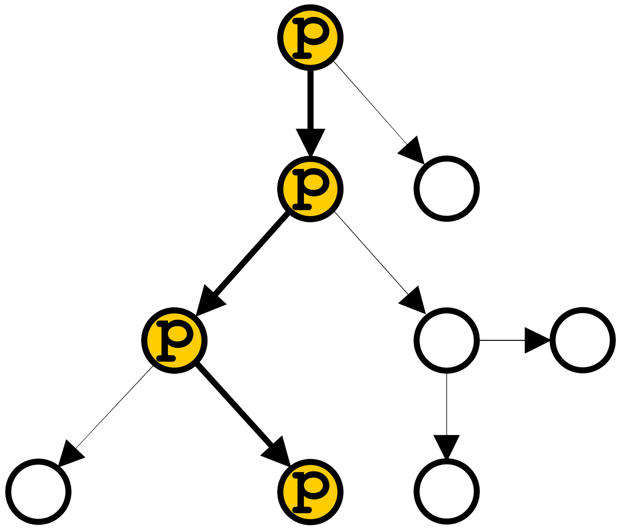


UPPAAL Safety properties

A specific condition ***holds in all the states*** *of an execution path.* **Two possibilities**:

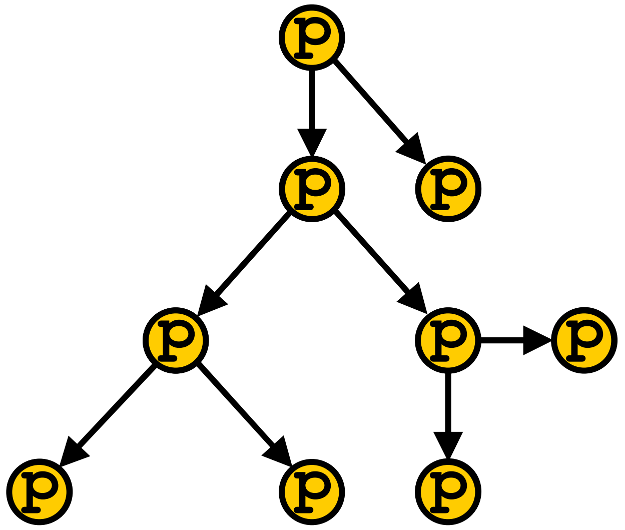
`E[] p` "Exists globally p"

meaning there is an execution path in which p holds for all the states of the path



`A[] p` "Always globally p"

For each (all) execution path p holds for all the states of the path

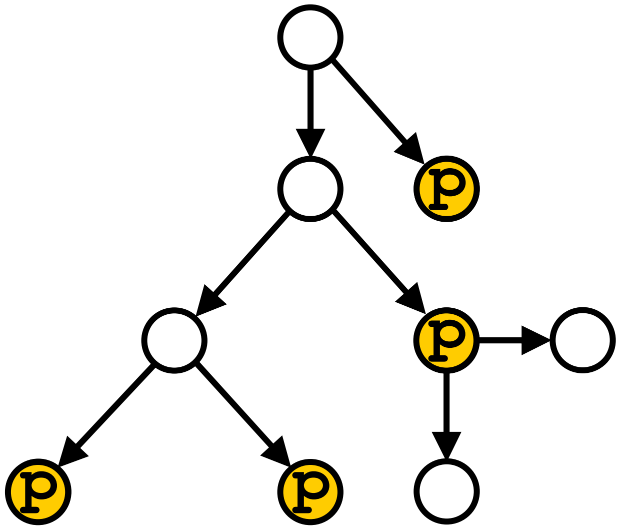


UPPAAL Liveness properties

A specific condition is ***guaranteed to hold eventually*** *(= at some moment).* **Two possibilities:**

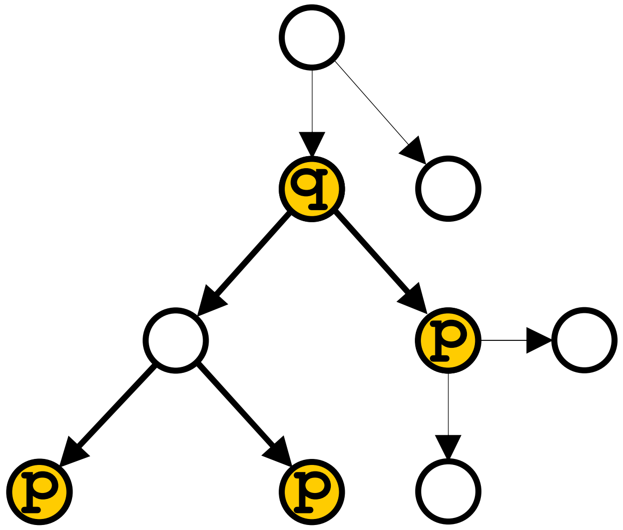
`A<> p` "Always eventually p"

For each (all) execution path p holds for at least one state of the path



`q-->p` "q always leads to p"

Any path that "starts" with a state in which q holds reaches later a state in which p holds



UPPAAL Deadlock properties

*A **deadlock** is possible or not in the model.*

- **A state is in a *deadlock* if** it is impossible that the model evolves to a successor state neither by waiting some time nor by a transition between locations, i.e. there are no enabled transitions

- **Two typical examples:**
  - `E<> deadlock` = “Exists deadlock”
  - `A[] not deadlock` = “There is no deadlock”

- Note that the word “deadlock” can be used inside any expression formalizing a specific property

Some remarks on verification

- When using model checking for verifying a system, it is usually studied **whether there is any *deadlock* in the model or not**
  - A deadlock can be evidence of a design error but not necessarily
  - It could be enough to verify that deadlock only happens when expected. E.g. :

    ```
    A[] deadlock imply (cont==max and autom.end)
    ```

- **Even using abstraction the state space may explode**. There are **verification options** that may help.
  - If some options are enabled, the output of the verifier might be that property is "maybe satisfied".→The verifier cannot determine the truth value of the property due to the approximations used.

Presentation Outline

A simple example from the tutorial (1)

- **A mutex algorithm**. Specification.

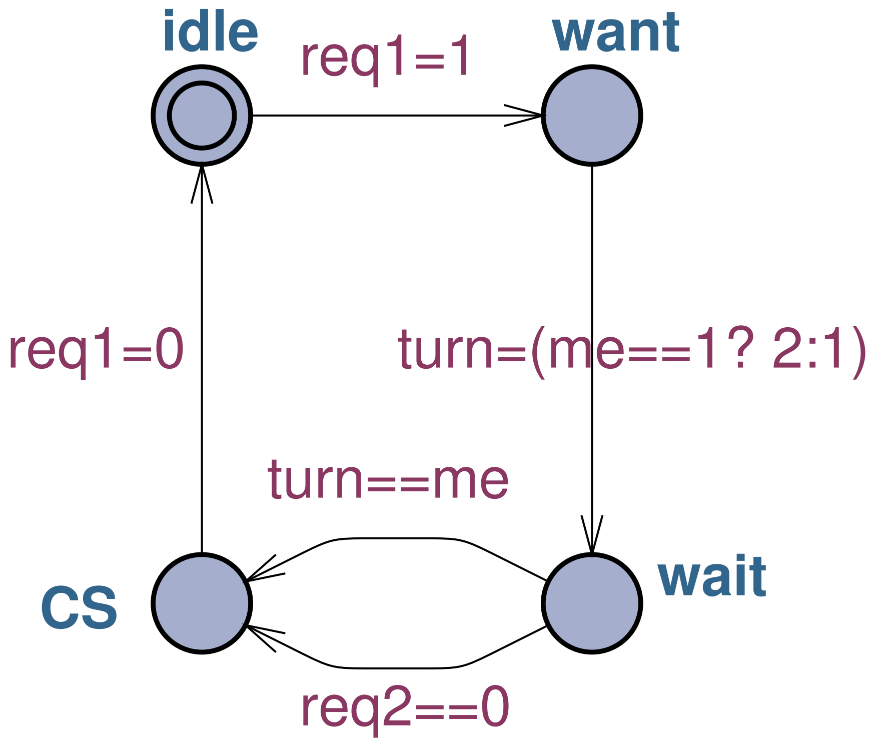
Process 1	Process 2

`req1=1;`
`turn=2;`
`while(turn!=1 && req2!=0);`
`//critical section`
`job1();`
`req1=0;`
	`req2=1;`
	`turn=1;`
	`while(turn!=2 && req1!=0);`
	`//critical section`
	`job2();`
	`req2=0;`

A simple example from the tutorial (2)

- **A mutex algorithm**. Modeling with UPPAAL.

```
Process 1
idle:
 req1=1;
want:
 turn=2;
wait:
 while(turn!=1 && req2!=0);
CS:
 //critical section
 job1();
 //and return to idle
 req1=0;
```



```
A[] not(P1.CS and P2.CS)
E<> P1.CS
```

Presentation Outline

Installation Instructions

- Make sure you have the Java version 5 installed.
  - E.g.: www.java.com/es/download/manual.jsp
- Go to the UPPAAL page: www.uppaal.com
- Click on the download tag and then on the link Uppaal 4.0 (current official release)
- Fill the license agreement form. Click on “Register & Download”
- Unzip files
- To run UPPAAL double-click the file uppaal.jar

Presentation Outline

References

Some used to create this presentation and some useful for further reading

- UPPAAL (all available at www.uppaal.com)
  - *Uppaal2k: Small Tutorial.* 16 October 2002
  - G. Behrmann, A. David, and K. G. Larsen. *A Tutorial on Uppaal*
  - UPPAAL Online Help

- Model Checking
  - C. Baier and J.-P. Katoen. *Principles of Model Checking.* MIT Press. 2008.
  - E.M. Clarke, O. Grumberg and D.A. Peled. *Model Checking*. The MIT Press. 2000.