

# Formal methods.

## Introduction to UPPAAL

**Vitaliy Mezhuyev**

---

---

# Introduction

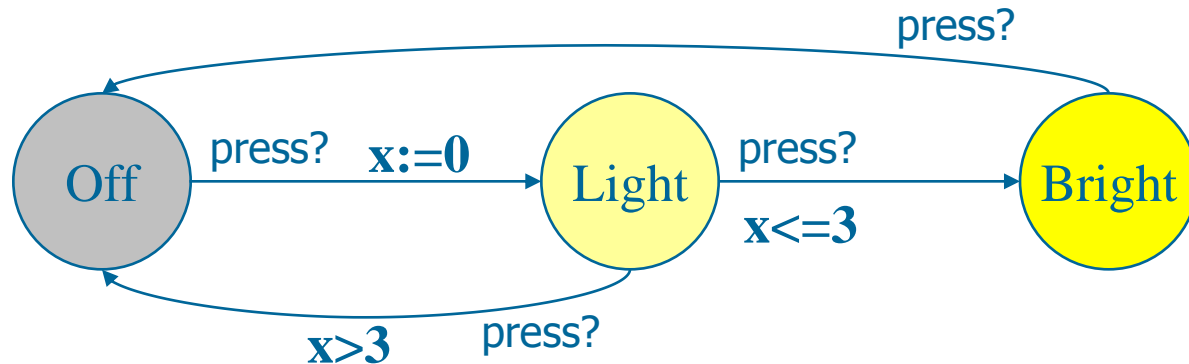
- UPPAAL is an integrated environment for modeling, simulation and verification of real time system modeled as network of timed automata.
  - UPPAAL was developed jointly by **Uppsala** University and **Aalborg** University.
  - The first prototype of UPPAAL named TAB, was developed at Uppsala University in 1993 by Wang Yi et al.
  - In 1995, Aalborg University was joined the development and then TAB was renamed UPPAAL with UPP standing for Uppsala and AAL for Aalborg.
-

---

# Introduction

- It serves as a modeling language to describe system behavior as networks of automata extended with clock and data variables (see example on next slide).
  - It models a system as a collection of processes, finite control structures and clocks, communicating through channels and shared variables.
  - It used to make a simulation of the system and check if there is an error in the system.
  - It allows to check both **invariants** and **reachability** properties by exploring the statespace of a system.
-

# Intelligent Light Control



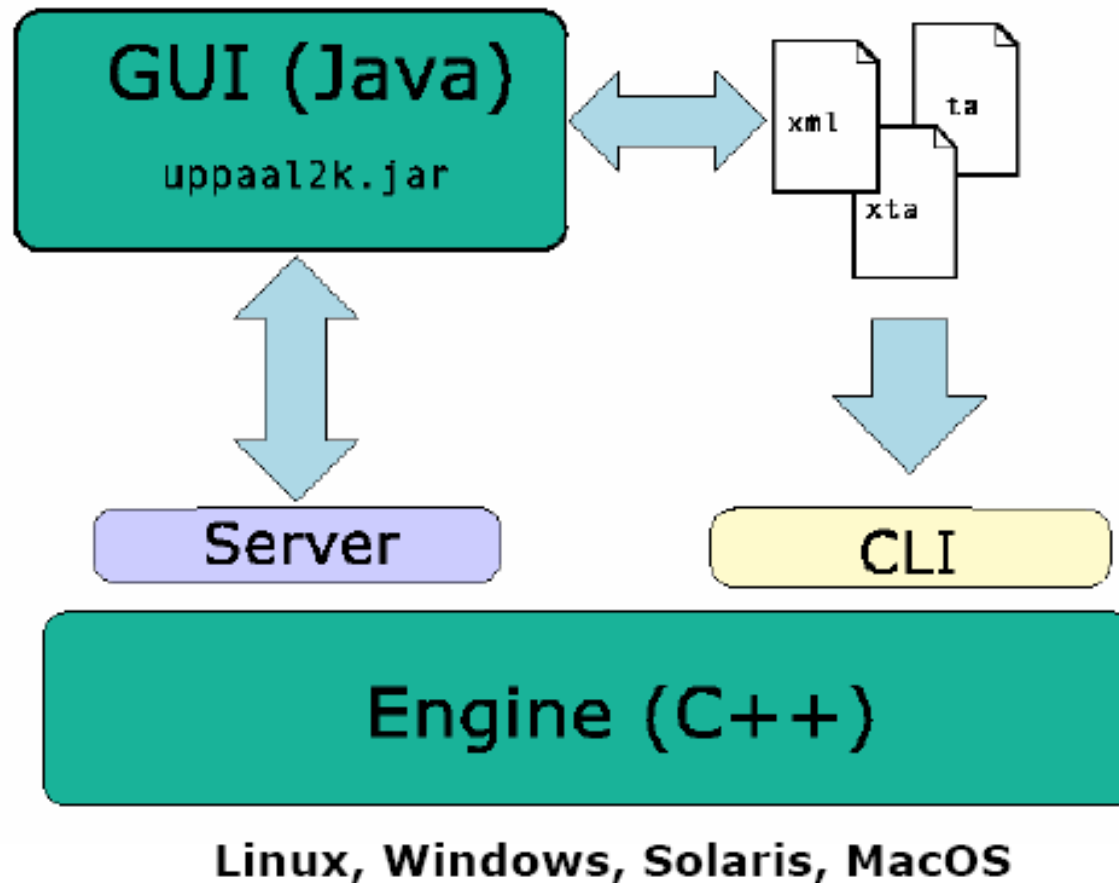
- **Requirements:** If a user **quickly** presses the light control twice, then the light should get brighter; on the other hand, if the user **slowly** presses the light control twice, the light should turn off.
- **Solution:** Add a real-valued clock,  $x$ .

---

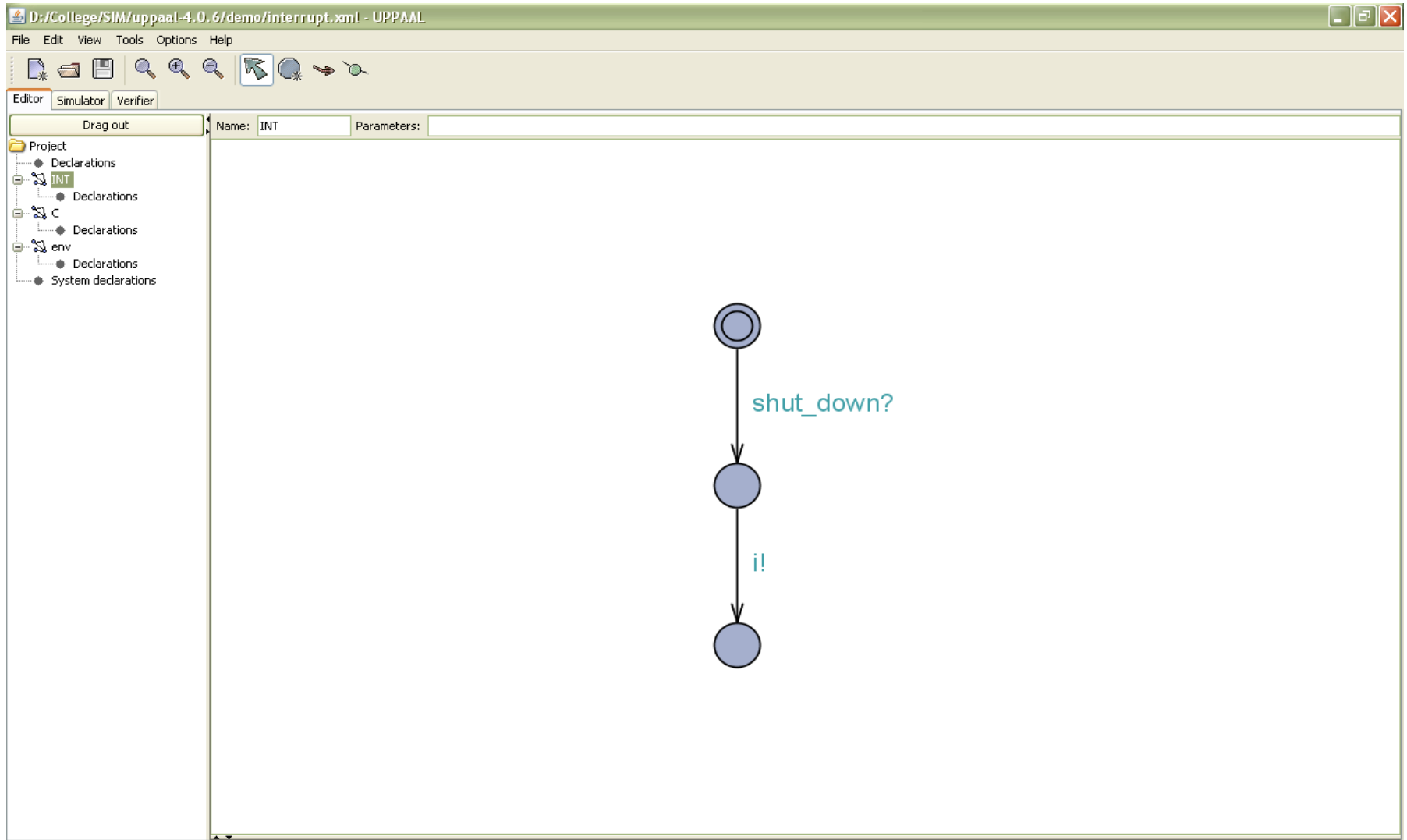
# UPPAAL's Features

- A graphical interface allowing networks of timed automata to be defined by drawing.
  - An automatic compilation of the graphical definition into a textual format used by the model-checker, thus supporting the important principle “what you see is what your verify”.
  - In case verification of a particular real-time system fails, a diagnostic trace is automatically reported by UPPAAL.
-

# UPPAAL Architecture



# UPPAAL - Editor (Modeller)



# UPPAAL - Simulator

D:\College\SIM\uppaal-4.0.6\demo\interrupt.xml - UPPAAL

File Edit View Tools Options Help

Editor Simulator Verifier

Drag out

Enabled Transitions

deadlock

Next Reset

Simulation Trace

```
(-, -, ON)
up: env --> C
(-, -, ON)
down: env --> C
(-, -, ON)
shut_down: env --> INT
(-, -, OFF)
i: INT --> C
(-, -, OFF)
```

Trace File:

Prev Next Replay

Open Save Random

Slow Fast

Drag out

count = 0

**INT**

```
graph TD
    S(( )) -- shut_down? --> I(( ))
    I -- i! --> R(( ))
```

**C**

```
graph TD
    S(( )) -- up? count_up() --> S
    S -- down? count_down() --> S
    S -- i? --> R(( ))
```

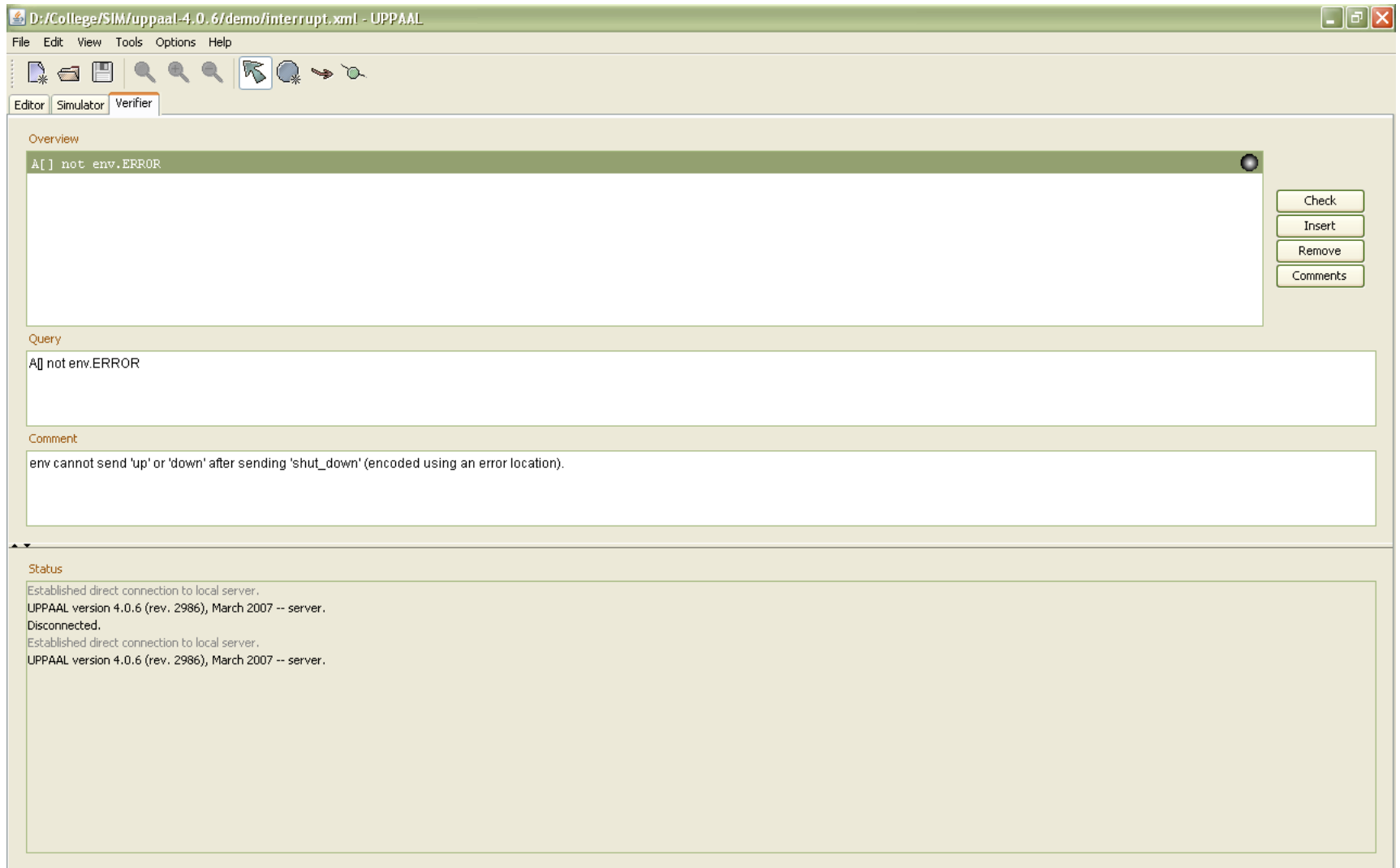
**env**

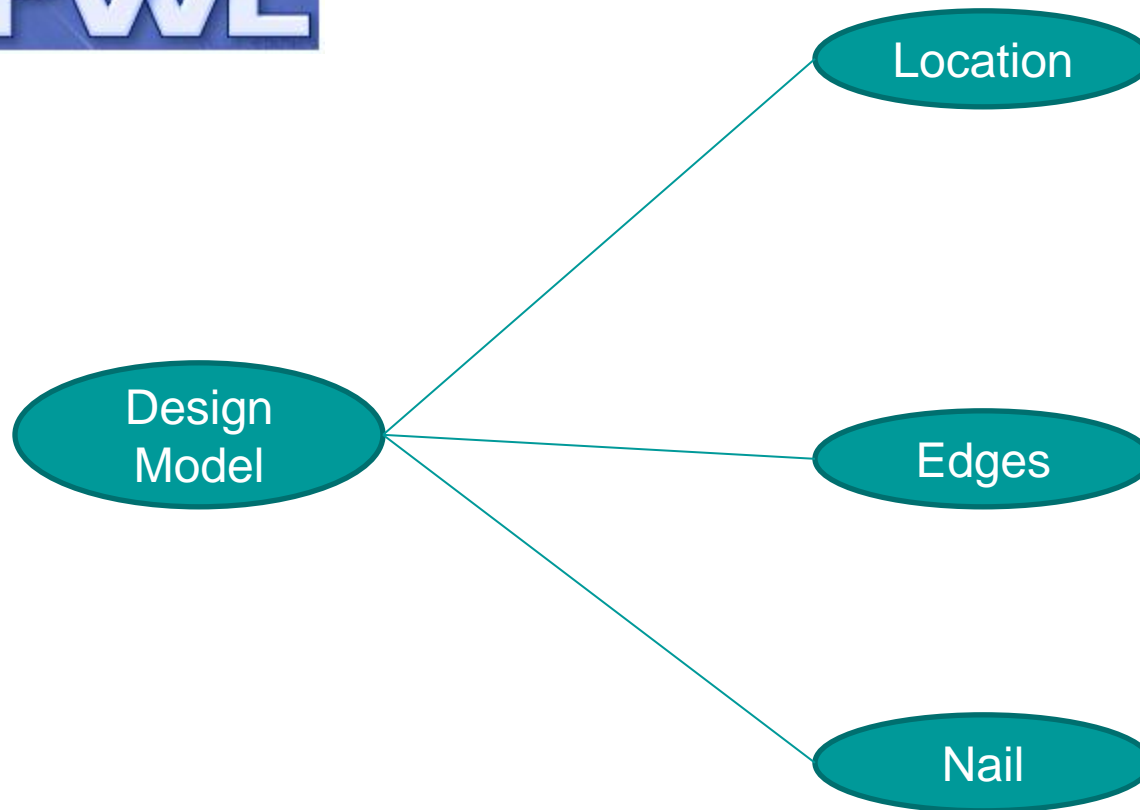
```
graph TD
    S(( )) -- down! --> S
    S -- up! --> S
    S -- shut_down! --> O((OFF))
    O -- down! --> E((ERROR))
    O -- up! --> E
```

**INT C env**



# UPPAAL - Verifier





# Locations

The control nodes of an UPPAAL process.

## 1. Initial Locations

The beginning of the process. Each model must have exactly one initial location. The initial location is marked by a double circle.

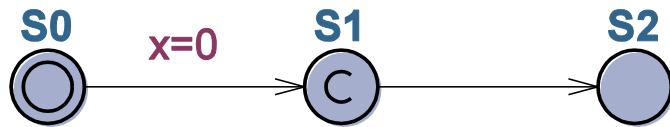
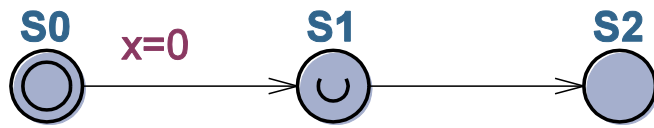
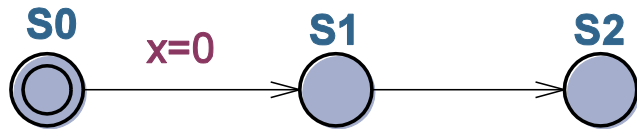
## 2. Urgent Locations

Urgent locations freeze time. This forces the actual process to always make a transition without any delay.

## 3. Committed Locations

Like urgent locations, committed locations freeze time. Furthermore, if any process is in a committed location, the next transition must involve an edge from one of the committed locations.

# Example



# Locations

Locations can have an optional name. Besides serving as an identifier allowing you to refer to the location from the requirement specification language. The name must be a valid identifier.

The screenshot shows a 'Location' dialog box with the following fields and options:

- Name:** Start
- Invariant:**  $x \leq 2$
- ☒ Initial
- ☐ Urgent
- ☐ Committed
- Buttons:** OK, Cancel

Conjunction of simple conditions on clocks, differences between clocks, and boolean expressions not involving clocks. The bound must be given by an integer expression. Lower bounds on clocks are disallowed. States which violate the invariants are undefined; by definition, such states do not exist.

Exactly one per Template

Freeze time; *i.e.* time is not allowed to pass when a process is in an urgent location.

Like urgent locations, committed locations freeze time. Furthermore, if any process is in a committed location, the next transition must involve an edge from one of the committed locations.

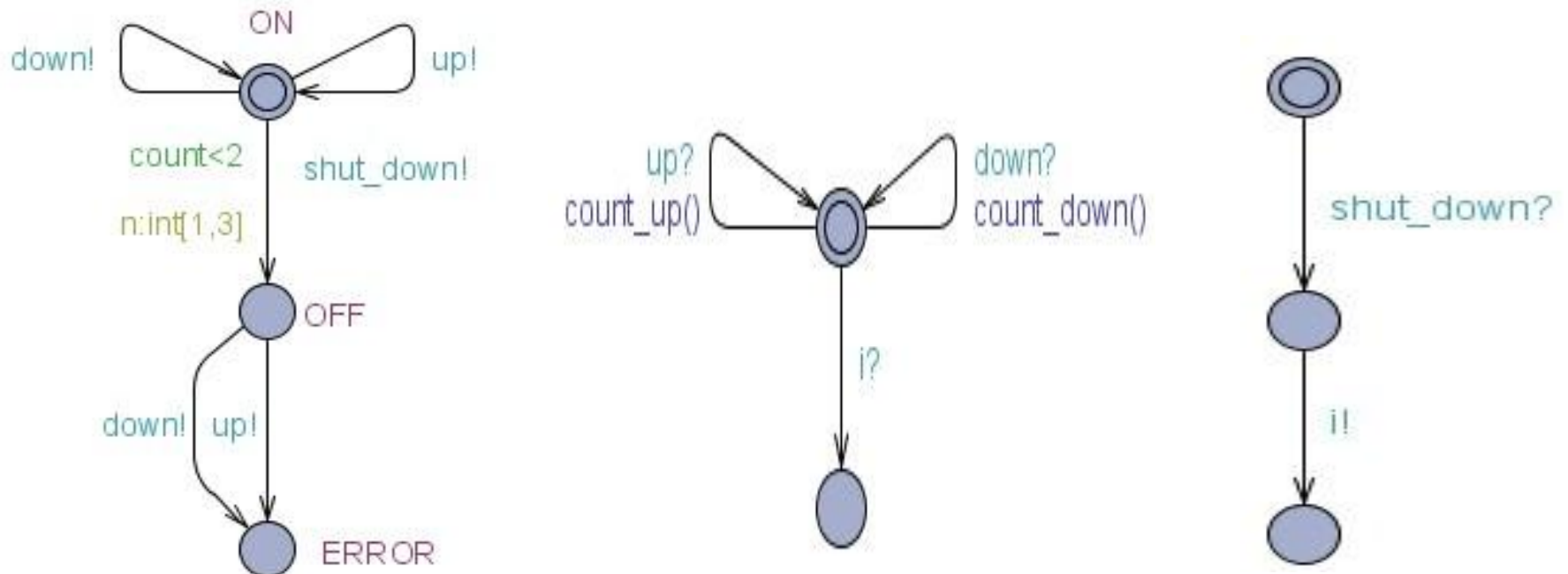
# Invariant

- Express condition at location on the value of clocks and integer variable that must be satisfied for the transition to be taken.
- Its evaluates as boolean
- Only clock variables, integer variables, constants are referenced (or arrays of such)

# Edges

Line between two control nodes (locations).

Edges are annotated with *selections*, *guards*, *synchronizations* and *updates*.



---

# Edges - Cont

## 1. Selections

Selections non-deterministically bind a given identifier to a value in a given range.

## 2. Guards

- ✓ Express condition at edge on the value of clocks and integer variable that must be satisfied for the transition to be taken.
  - ✓ It is side-effect free, type correct, and evaluates to boolean
  - ✓ Only clock variables, integer variables, constants are referenced (or arrays of such)
-



---

# Edges - Cont

## 3. Synchronisations

Synchronisation means that two processes change location in one simulation step. Synchronisation is done with (or via) channels. To synchronize two processes, the synchronisation labeled with the channel variable that has been declared before, followed by “!” for one of them and “?” for the other.

## 4. Updates

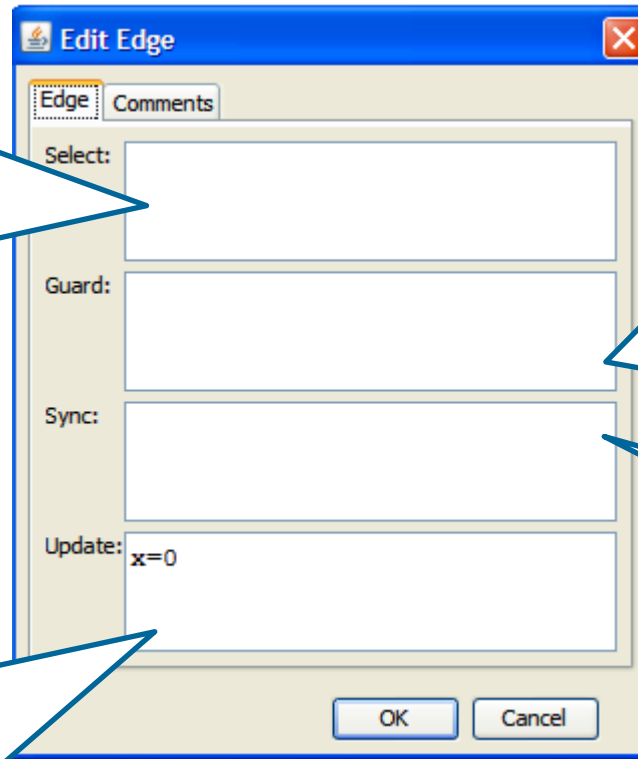
When executed, the update expression of the edge is evaluated. The side effect of this expression changes the state of the system.

---

# Edges

Non-deterministically bind a given identifier to a value in a given range. The other three labels of an edge are within the scope of this binding.

When executed, the update expression of the edge is evaluated. The side effect of this expression changes the state of the system.



The image shows a software dialog box titled "Edit Edge". It has two tabs: "Edge" (selected) and "Comments". The "Edge" tab contains four input fields: "Select:", "Guard:", "Sync:", and "Update:". The "Update:" field contains the text "x=0". At the bottom of the dialog are "OK" and "Cancel" buttons. Callout lines connect the four input fields to explanatory text blocks on the slide.

An edge is enabled in a state if and only if the guard evaluates to true.

Processes can synchronize over channels. Edges labeled with complementary actions over a common channel synchronize.

# Variable

Generally, there are 4 variable types in Uppaal

- Integer
  - Bool
  - Clock
  - Channel
- Same types with C and C++
- New types in Uppaal

# Integer

- The range is -32768....32767

# Boolean

- There are 2 values of Bool Variable:
  - ▣ True
  - ▣ False

---

# Clocks

- Used to schedule processes.
  - Declared with the keyword **clock**.
-

# Channel

- Used to synchronize two processes.
- Declared with the keyword **chan**.
- Done by annotating edges in the model with special **synchronisation** labels.

---

# Urgent Channels

- When a channel is declared as an Urgent Channel, synchronization via that channel has priority over normal channels and the transition must be taken without delay.
  - Declared with the keyword `urgent chan`.
  - No clock guard allowed on transitions with urgent actions.
  - Invariants and data-variable guards are allowed.
-



# Broadcast channel

- Broadcast channels allow 1-to-many synchronisations.
- An edge with synchronisation label **e!** emits a broadcast on the channel **e** and that any enabled edge with synchronisation label **e?** will synchronise with the emitting process.

---

# Declaration in Uppaal

The syntax used for declarations in UPPAAL is similar to the syntax used in the C programming language.

## Integer

- **int num1, num2;**  
two integer variables “num1” and “num2” with default domain.
  - **int[0,100] a;**  
an integer variable “a” with the range 0 to 100.
  - **int a[2][3];**  
a multidimensional integer array.
  - **int[0,5] b=0;**  
an integer variable with the range 0 to 5 initialized to 0.
-

---

# Declaration in Uppaal - Cont

## Boolean

- **bool yes = true;**  
a boolean variable “yes” initialize to true.
- **bool b[8], c[4];**  
two boolean arrays b and c, with 8 and 4 elements respectively.

## Const

- **const int a = 1;**  
constant “a” with value 1 of type integer.
  - **const bool No = false;**  
constant “No” with value false of type boolean.
-

---

# Declaration in Uppaal - Cont

## Clock

- **clock x, y;**  
two clocks x and y.

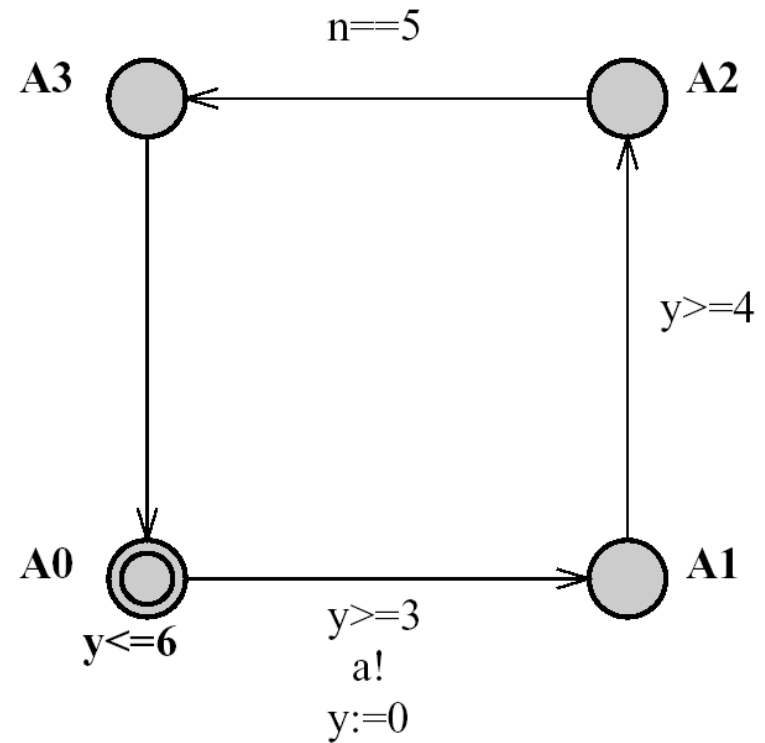
## Channel

- **chan d;**  
a channel.
- **urgent chan a, b ,c;**  
urgent channel.

# Example – textual format (.xta file)

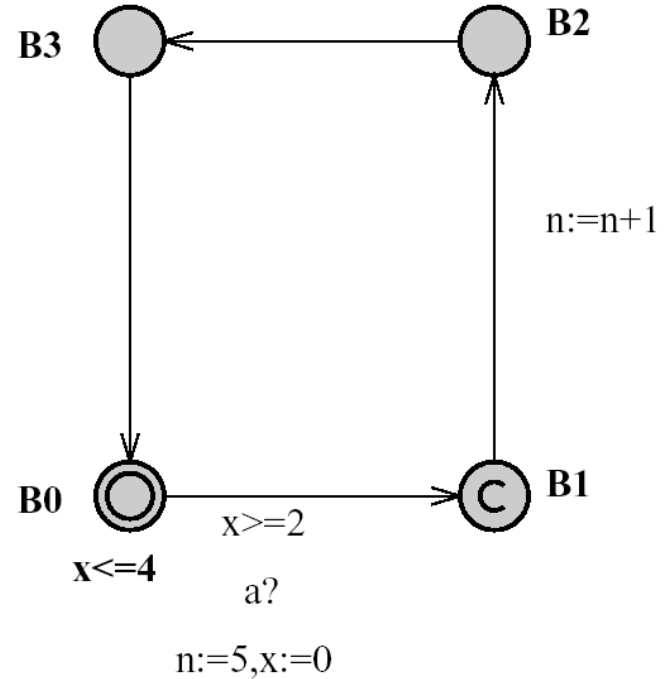
```
clock x, y;
int n;
chan a;

process A {
  state A0 { y<=6 }, A1, A2, A3;
  init A0;
  trans A0 -> A1 {
    guard y>=3;
    sync a!;
    assign y:=0;
  },
  A1 -> A2 {
    guard y>=4;
  },
  A2 -> A3 {
    guard n==5;
  }, A3 -> A0;
}
```



# Example (cont.)

```
process B {  
  state B0 { x<=4 }, B1, B2, B3;  
  commit B1;  
  init B0;  
  trans B0 -> B1 {  
    guard x>=2;  
    sync a?;  
    assign n:=5,x:=0;  
  },  
  B1 -> B2 {  
    assign n:=n+1;  
  },  
  B2 -> B3 {  
  }, B3 -> B0;  
}
```



```
system A, B;
```

# Example (cont.)

C:\uppaal-3.2.6\uppaalNutshell.xml - UPPAAL2k

File Templates View Queries Options Help

System Editor Simulator Verifier

Drag out

Enabled Transitions

Next Reset

Simulation Trace

(A.1, B.1)  
(A1, B1)  
(B.2)  
(A1, B2)  
(A.2)  
(A2, B2)  
(B.3)  
(A2, B3)  
(B.4)  
**(A2, B0)**

Trace File:

Prev Next Replay  
Open Save Random

Slow Fast

Drag out

Variables

n = 6  
x = 4  
y = 4  
y = x

**process A**

```
graph TD; A0((A0)) -- "y >= 3  
a!  
y := 0" --> A1((A1)); A1 -- "y >= 4" --> A2((A2)); A2 -- "n == 5" --> A3((A3)); A3 -- "y <= 6" --> A0
```

**process B**

```
graph TD; B0((B0)) -- "x >= 2  
a?" --> B1((B1)); B1 -- "n := n + 1" --> B2((B2)); B2 --> B3((B3)); B3 --> B0
```

---

# Linear Temporal Logic (LTL)

- LTL formulae are used to specify temporal properties.
  - LTL includes propositional logic and temporal operators:
    - $[ ]P$  = always P
    - $\langle \rangle P$  = eventually P
    - $P \text{ U } Q$  = P is true until Q becomes true
-



# LTL in Uppaal

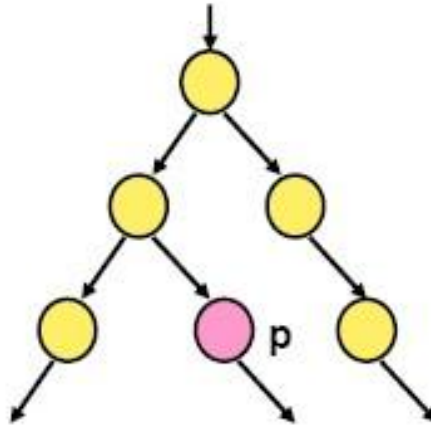
- **E** - exists a path ( “**E**” in UPPAAL).
- **A** - for all paths ( “**A**” in UPPAAL).
- **[]** – all states in a path
- **<>** - some states in a path

The following combination are supported:

- **A[ ], A<>, E<>, E[ ]**.

$E \langle \rangle p$  – “p Reachable”

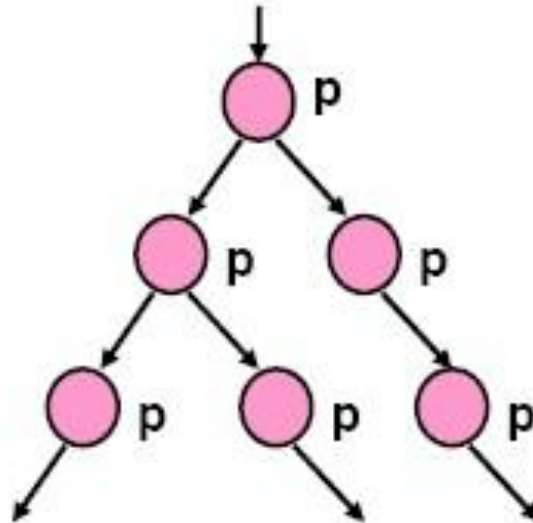
**$E \langle \rangle p$**  – it is possible to reach a state in which p is satisfied.



p is true in (at least) one reachable state.

$A[] p$  – “Invariantly  $p$ ”

**$A[] p$**  –  $p$  holds invariantly.

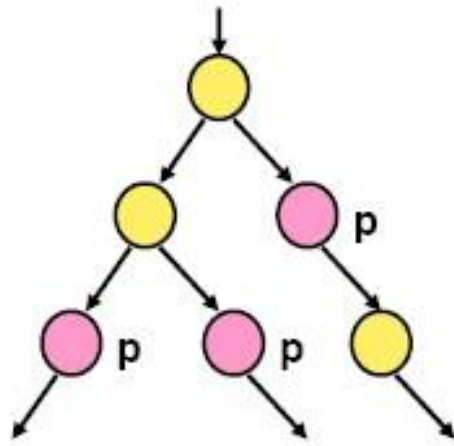


$P$  is true in all reachable states.

$A \langle \rangle p$  – “Inevitable  $p$ ”

$A \langle \rangle p$  –  $p$  will inevitably become true

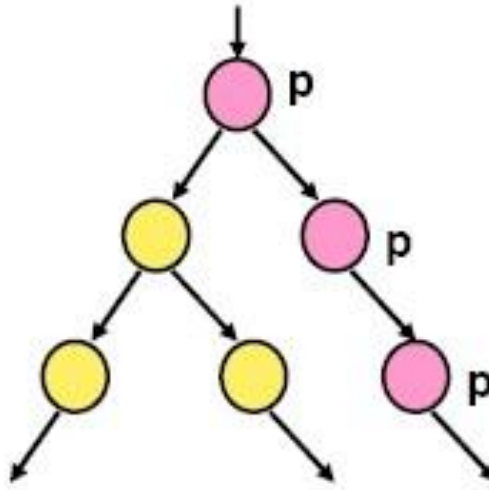
- The automaton is guaranteed to eventually reach a state in which  $p$  is true.



$P$  is true in some state of all paths.

$E[] p$  – “Potentially Always  $p$ ”

$E[] p$  –  $p$  is potentially always true.



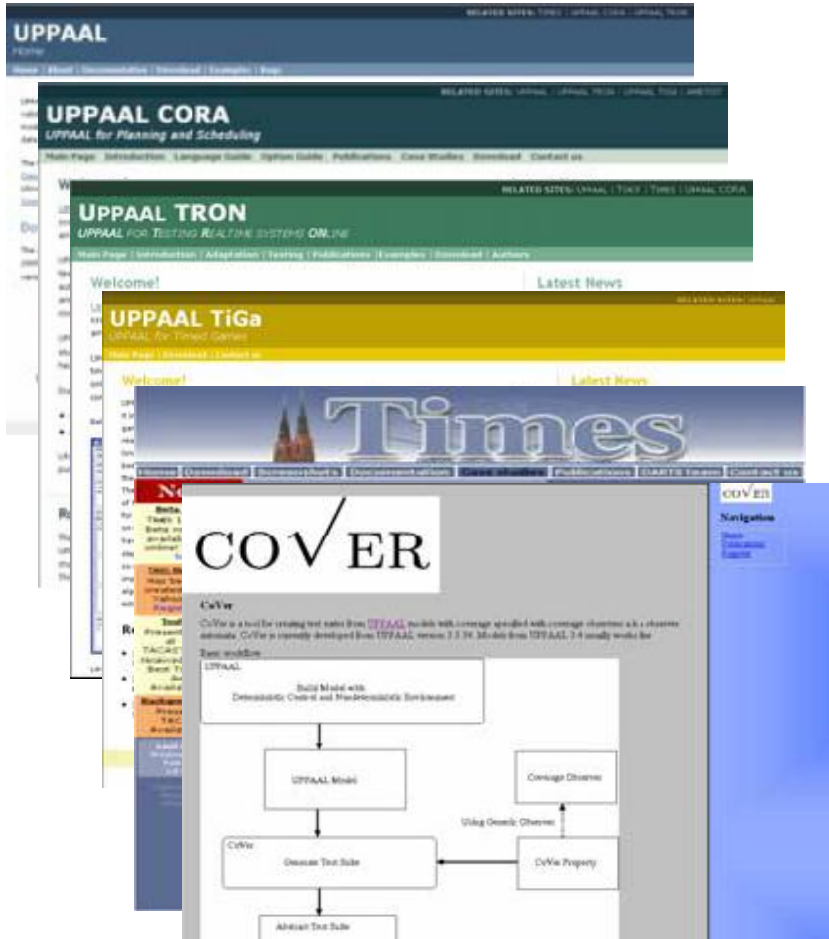
There exists a path in which  $p$  is true in all states.

---

# Verifying Properties

- $E<>p$ : there exists a path where p eventually true. (**Possibly**)
  - $A[]p$ : for all paths p always true. (**Invariantly**)
  - $E[]p$ : there exists a path where p always hold. (**Potentially Always**)
  - $A<>p$ : for all paths p will eventually hold. (**Eventually**)
  - $p \rightarrow q$ : whenever p holds q will eventually hold. (**Leads To**)
-

# UPPAAL Family

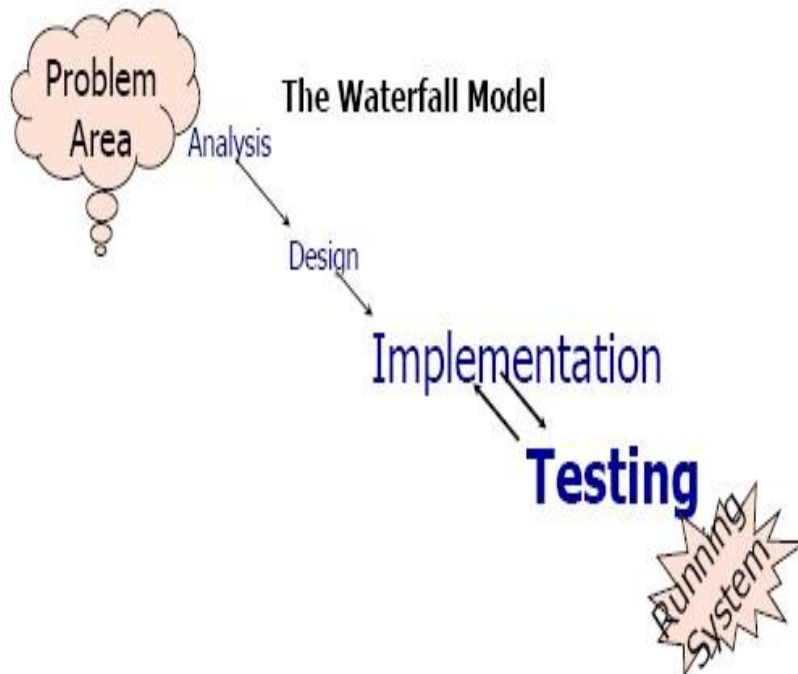


- “Classic”: real-time verification
- Cora : real-time scheduling
- Tron: online real-time testing
- TiGa: timed game
- Times: schedulability analysis
- CoVer: test case generation

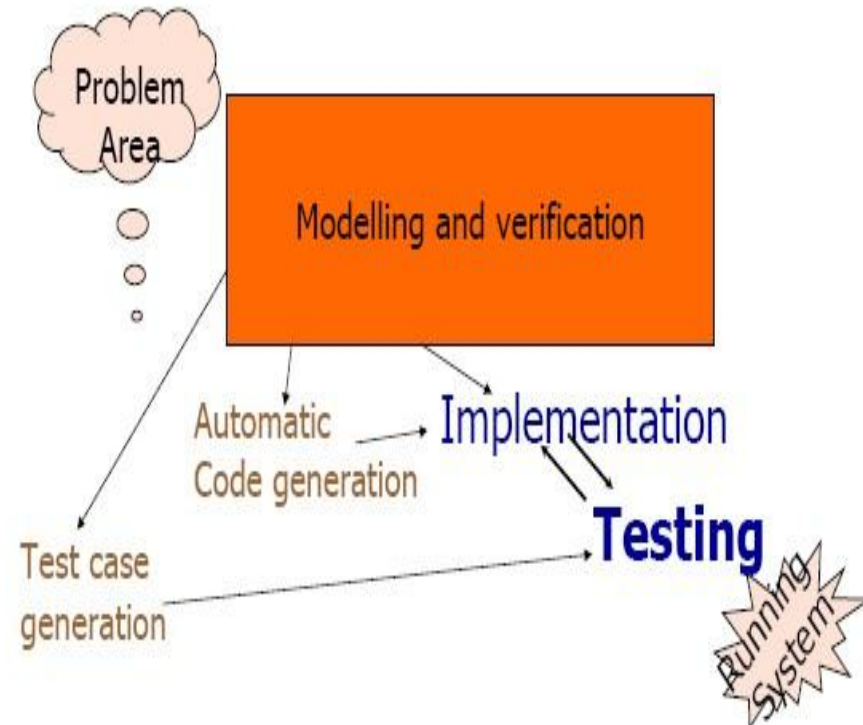
# Different Software Development



## Traditional



## Modern





---

# Terima Kasih

---