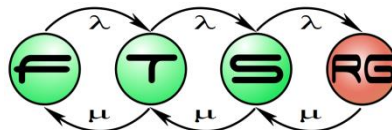


Modelling Environments, Code Generation

**Budapest University of Technology and Economics
Fault Tolerant Systems Research Group**



Content

Functions



Modelling Environments



Code Generation

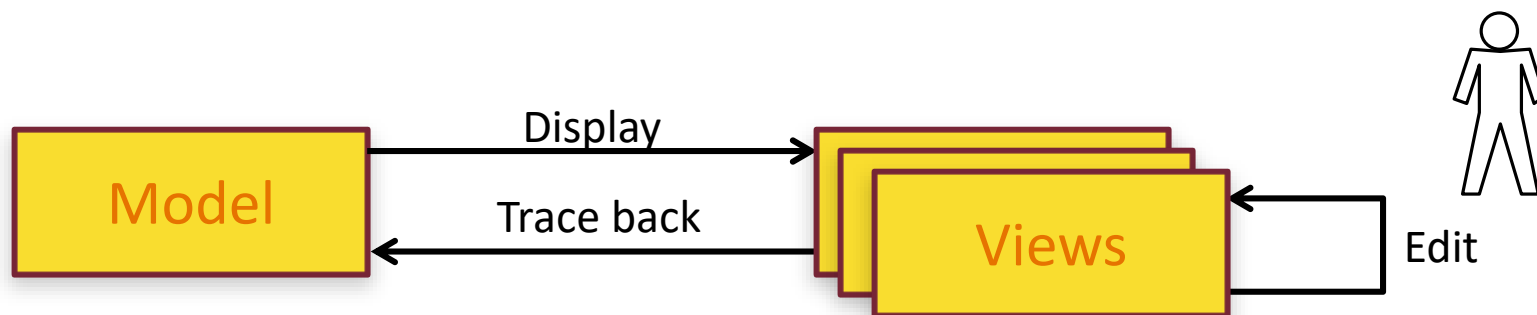
Functions

Environments

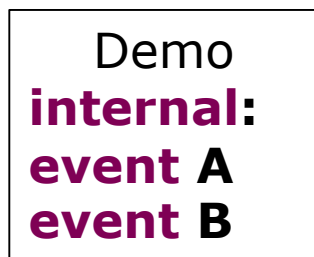
Code Generation

FUNCTIONS OF MODELLING ENVIRONMENTS

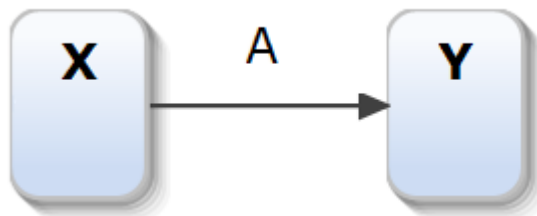
Functions of Modelling Environments



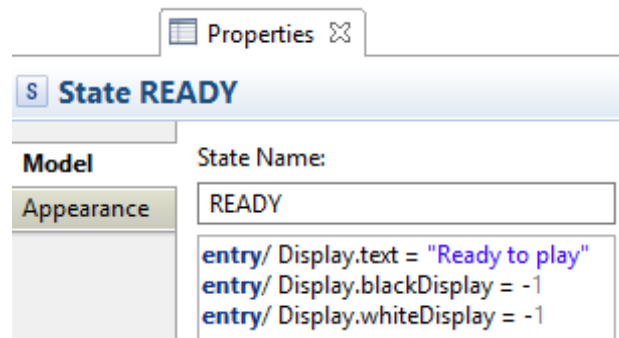
Textual



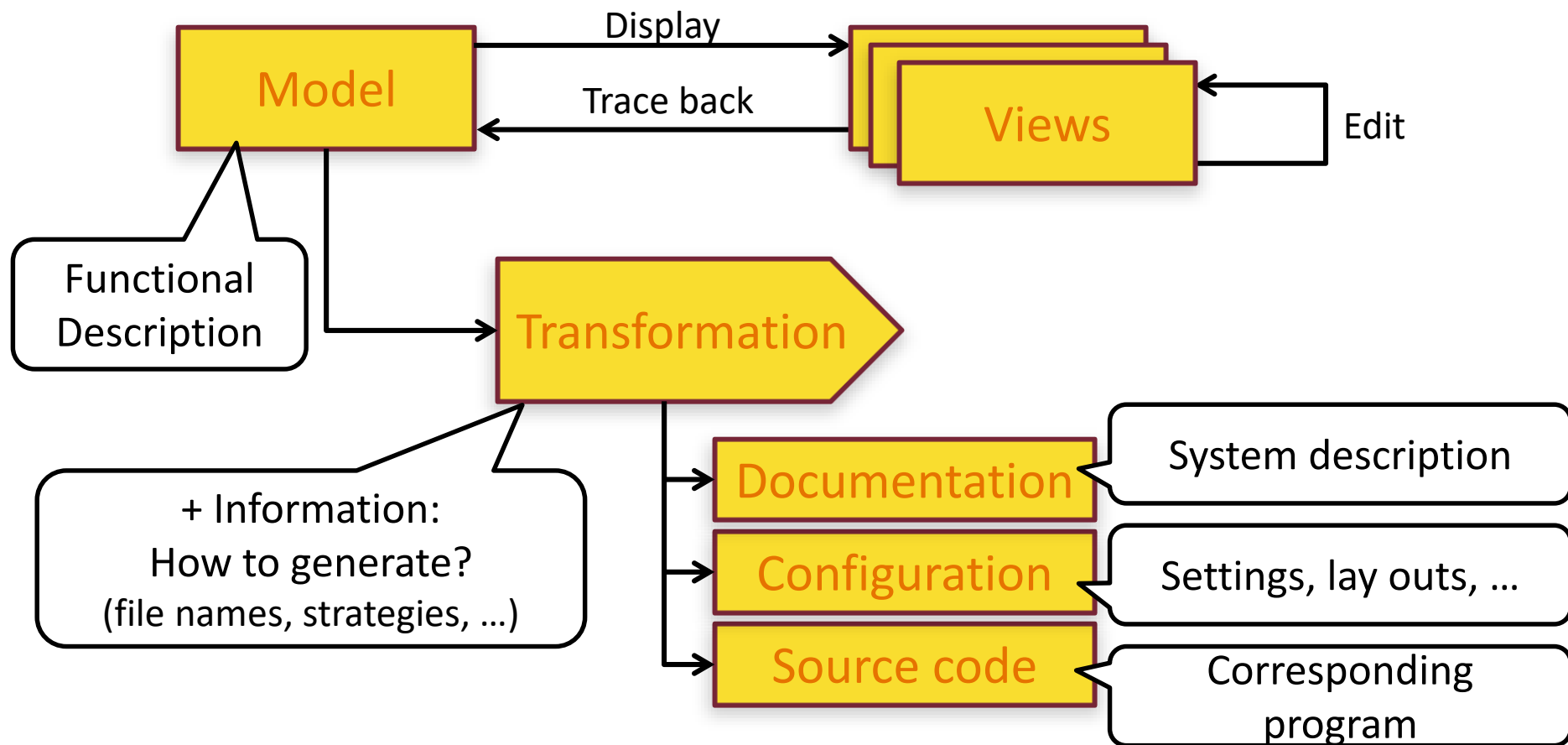
Graphical



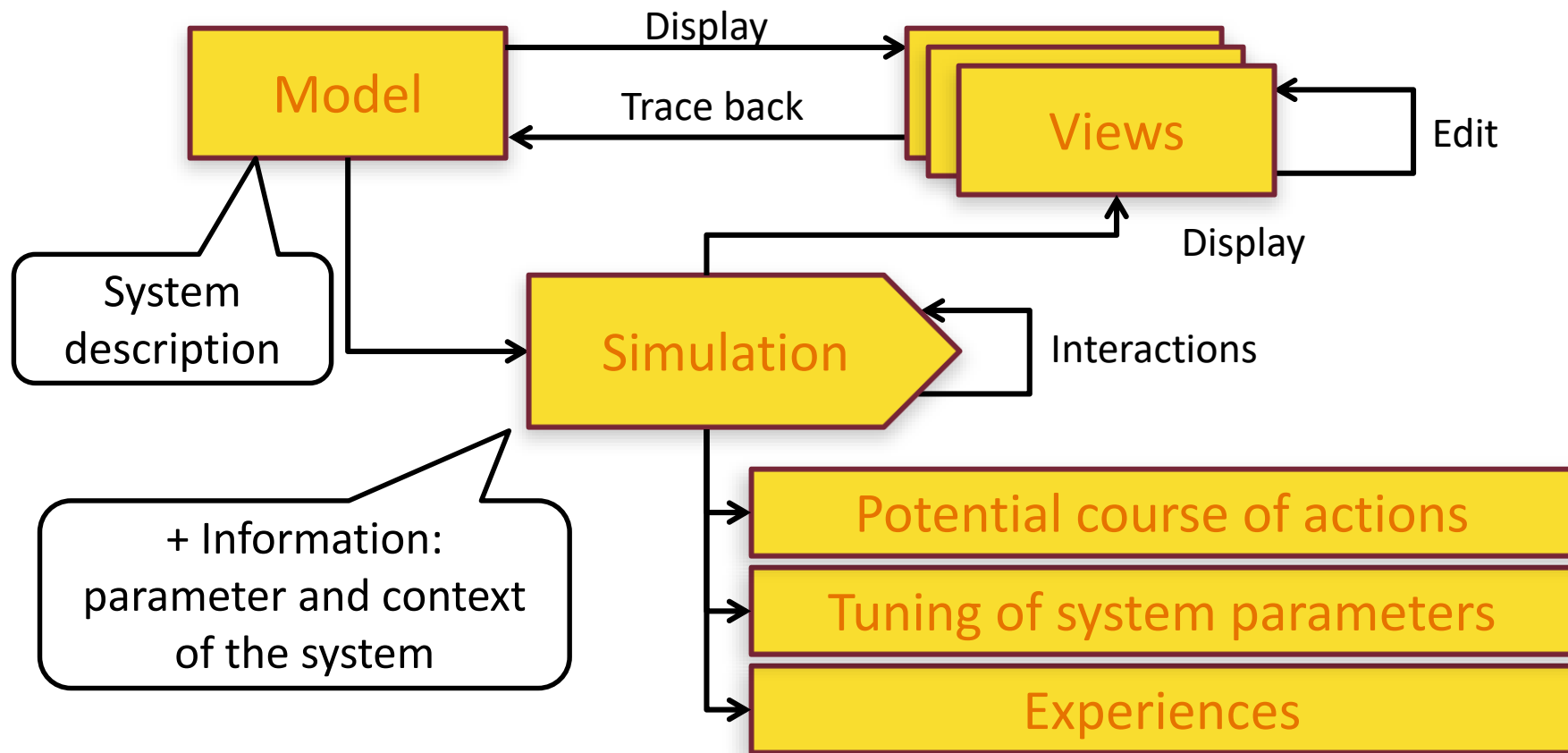
Structured Interfaces



Functions of Modelling Environments



Functions of Modelling Environments



Functions

Environments

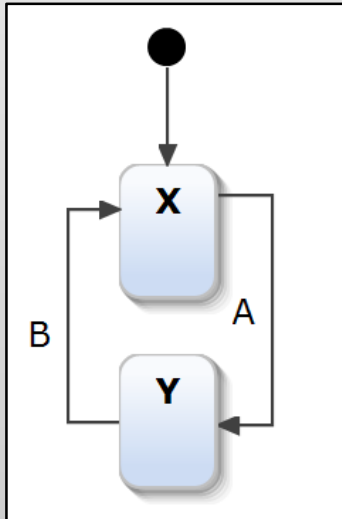
Code Generation

MODELLING ENVIRONMENTS

Modelling Functions of Yakindu

Concrete syntax
(for the User)

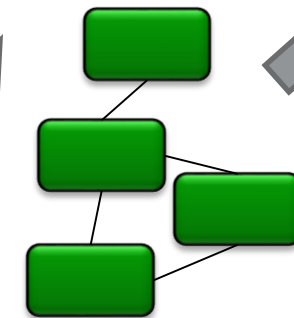
Graphical:



Textual:

Demo
internal:
event A
event B

Syntax $\rightarrow$ Semantics



Model
(Abstract syntax)

Modelling functions

Model verification

Errors (4 items)

- File platform:/resource/fr.irisa.triskell.kerme
- File platform:/resource/fr.irisa.triskell.kerme
- Unable to set the type of fsm::State::step:
- Unresolved type 'Sttring'. (missing using ?)

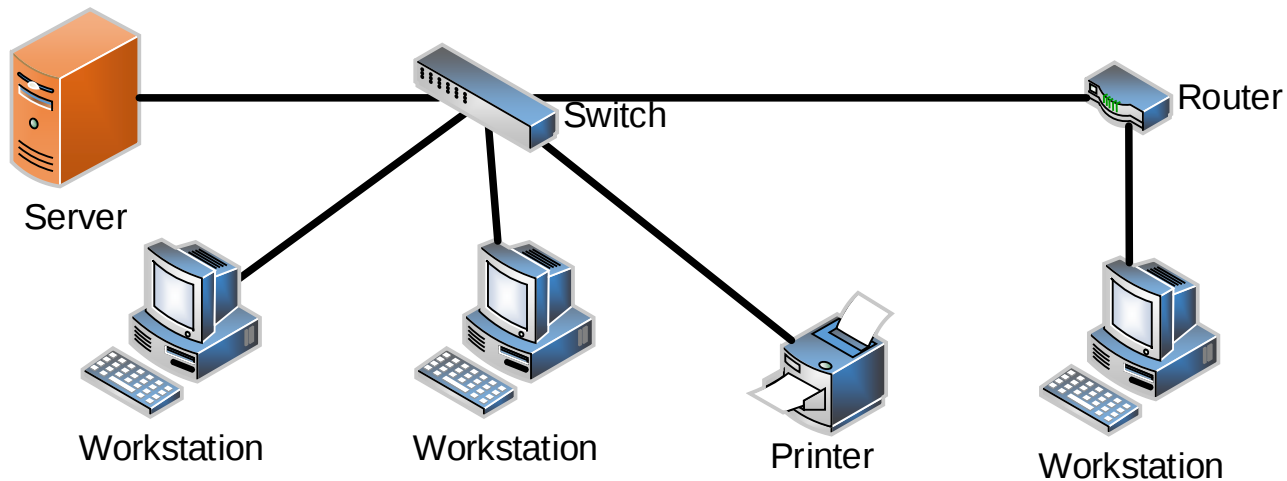
Code generation

```
</membership>
<profile defaultProvider="Sitefinity">
  <providers>
    <clear/>
    <add name="Sitefinity" connectionS
  </providers>
  <properties>
    <add name="FirstName"/>
    <add name="LastName"/>
    <!-- SNP specific properties -->
    <add name="NickName" />
    <add name="Gender" />
```

(Source code, documentation,
configuration)

Abstract Syntax

- **Definition:** Structural model of the system model under editing
 - Structural model of a model ???
- Will be maintained by the modelling environment
- Memory aid: structural model = **Graph**
 - **Graph of nodes, edges, properties**

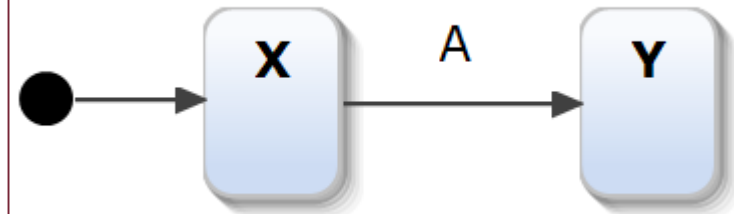


Example – Abstract Syntax: Yakindu

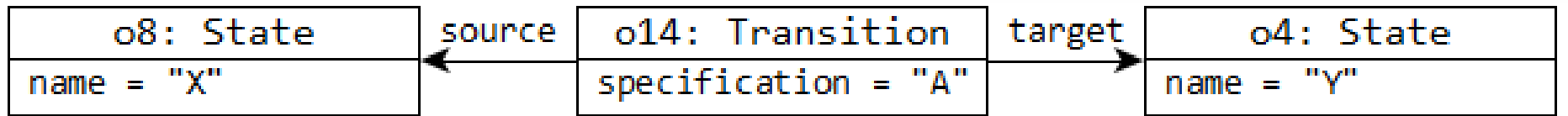
Question:

How would you implement a modelling environment tool?

Example: Yakindu Model



Abstract Syntax



Example – Abstract Syntax: Yakindu

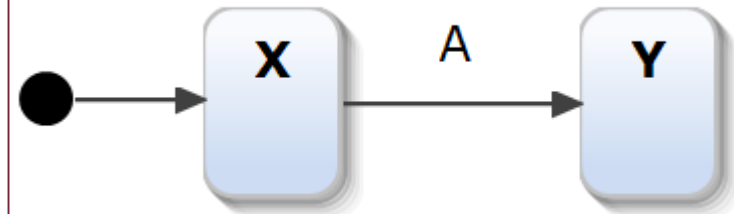
Question:

How would you implement a modelling environment tool?

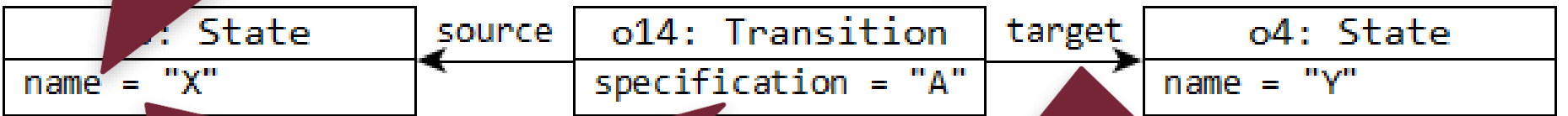
Names are stored as strings

```
name = "X"
```

Example: Yakindu Model



Abstract Syntax



Model elements as objects

Relations as references

Answer: It is a simple object-oriented program with some extra functions

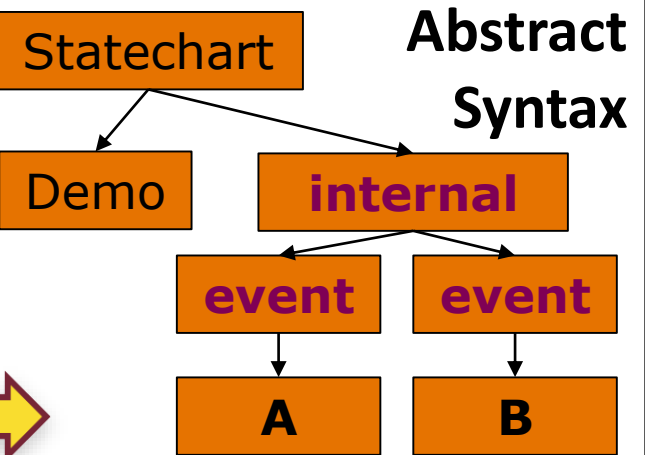
Concrete Syntax: Textual Syntax

- **Goal:** representation $\Leftrightarrow$ the model behind
- Textual syntax (e.g. programs)
 - Task: Text $\rightarrow$ Model
 - Rule based (otherwise it would be hard!)

Demo
internal:
event A
event B

Grammar

```
<Statechart> ::= <Name> <Interface>*  
<Interface>  ::= ("internal" | <Name>)  
              ":" <Event>*  
<Event>      ::= "event" <Name>  
<Name>       ::= ...
```



With **appropriate** technologies (e.g. Xtext) **anyone** can implement **his own** modelling/programming environment!

Counterexample for „appropriate technologies”

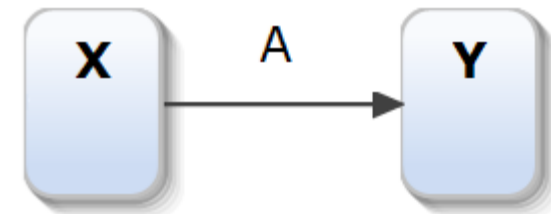
- **Task:** to decide whether a text consists of e-mail addresses

- A single address or a list
- Route specific information may appear
- Optionally names may also appear
- Hierarchical structure with '.' separator

- A single address or a list
- Route specific information may appear
- Optionally names may also appear
- Hierarchical structure with '.' separator

Concrete Syntax: Graphical Syntax

- **Goal:** representation $\Leftrightarrow$ the Model behind
- Graphical Syntax (e.g. diagram)
 - Task: Diagram $\leftrightarrow$ Model
 - More clear to read, harder to write, rule based



Condition over the Model

Id*:

Domain Class*:

Semantic Candidates Expression:

Creation of diagram elements

Label Alignment: ☐ Left ☒ Center ☐ Right

Label Expression:

Label Position:

Color*:

Label Color*:

Border Color*:

Condition satisfied $\rightarrow$ Diagram element will be created
Diagram will be changed $\rightarrow$ Model will also be changed

Concrete Syntax: Graphical Syntax

Result:

The screenshot shows the Sirius modeling tool interface. At the top is a toolbar with various icons for creating and editing models. Below the toolbar is a diagram area containing two states: 'X' (a solid blue box) and 'Y' (a dashed blue box). Below the diagram area is a tabbed interface with 'Properties' and 'Problems' tabs. The 'Properties' tab is active, showing a table for 'State Y'.

Property	Value
State Y	
Composite	☒ false
Documentation	
Incoming Transitions	→ X -> Y (A)
Leaf	☒ true
Name	Y

With **appropriate** technologies (e.g. Sirius) **anyone** can implement **his own** modelling/programming environment!

Model Validation: Syntax Analysis

- Syntax analysis: modelling tools connect logically cascading model elements

Declaration in interface:

var **clock**: integer = 60

Usage in model:

after 1 s [**clock**>0]/ **clock** -= 1

- Syntax-driven editor
 - Fault during editing → **Couldn't resolve reference**
 - Advanced editor (offering possibilities for instance)

- Code and diagram together

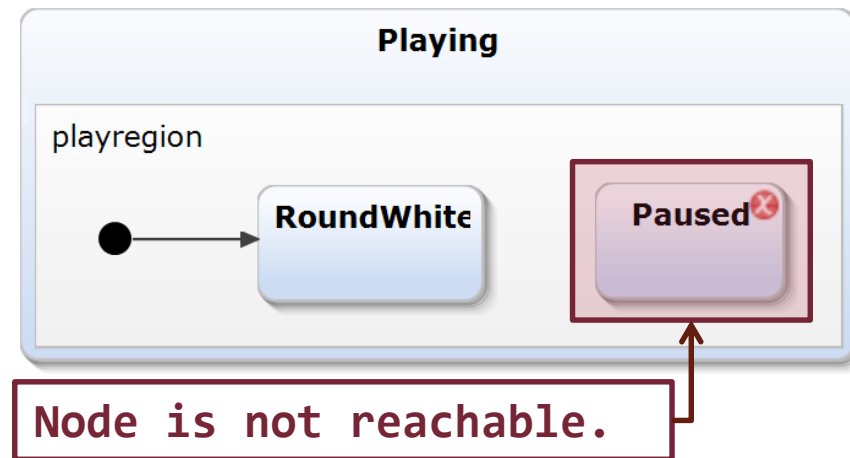
after 1 s [**clock**>0]/ **clock** -= 1



- Programming: **incorrect** during editing
Modelling: **correct** during editing

Model Validation: Structural Correctness

- Structural analysis: examining model graph
- Looking for error-patterns during editing
- Unreachable state, for instance:



- Further analysis: missing initial state, deadlock, variable assignment, etc.

Functions

Environments

Code Generation

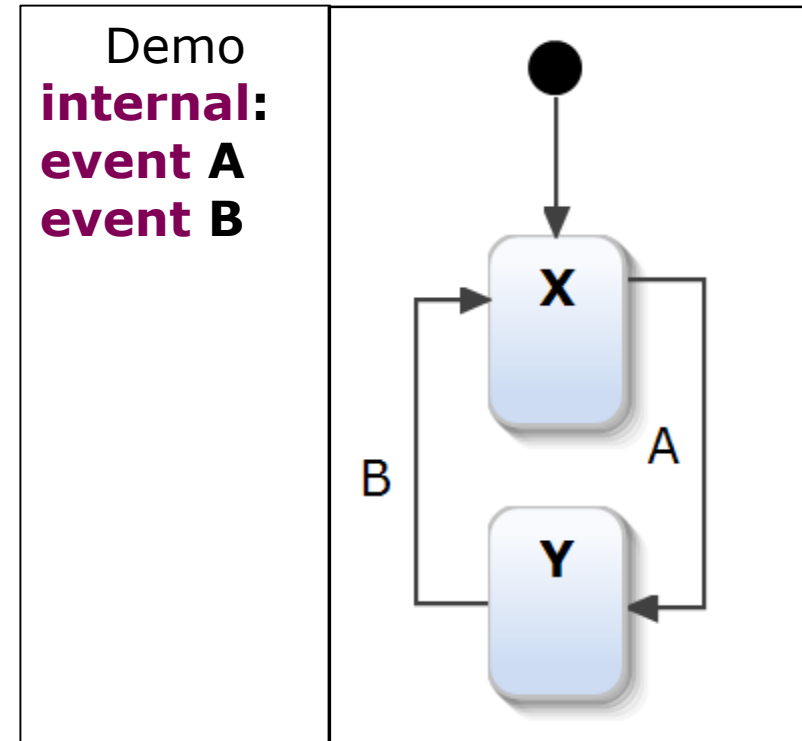
CODE GENERATION

Tasks of the Code Generation

- **Task:** automatic generation of models with *corresponding* behaviour
- Several options → Design decisions
 - **Interpretation:** Model will be loaded and executed
Program code: Source code will be generated
 - **Programming languages:** Java, C, C++, ...
 - **Optimising:** memory vs. processor
observability vs. performance
 - How can the generated code be extended with manually coded parts?
- Code generator: customizable + extendable

Code Generators – Example

- **Task:**
Generate C-code for a Yakindu state machine
- Write a function, which:
→ takes a model object
← returns a text
- The text will be written in a file „Demo.c”
- It will be compiled (by a C-compiler)



Template Based Code Generator (Xtend)

- Goal: States → Enum

The output will be collected in a char*, instead %s will the names X,Y be written

- Solution: a C program

```
sprintf(result,  
    "enum states {\n\tState%s,\n\tState%s\n};",  
    state1->name,  
    state2->name);
```

```
enum states {  
    StateX,  
    StateY  
};
```

😊 Works immediately!
😞 Unreadable

- Template (Xtend):

```
'''  
enum states {  
    State«state1.name»,  
    State«state2.name»  
}'''
```

The variable parts will be indicated

A template will be written

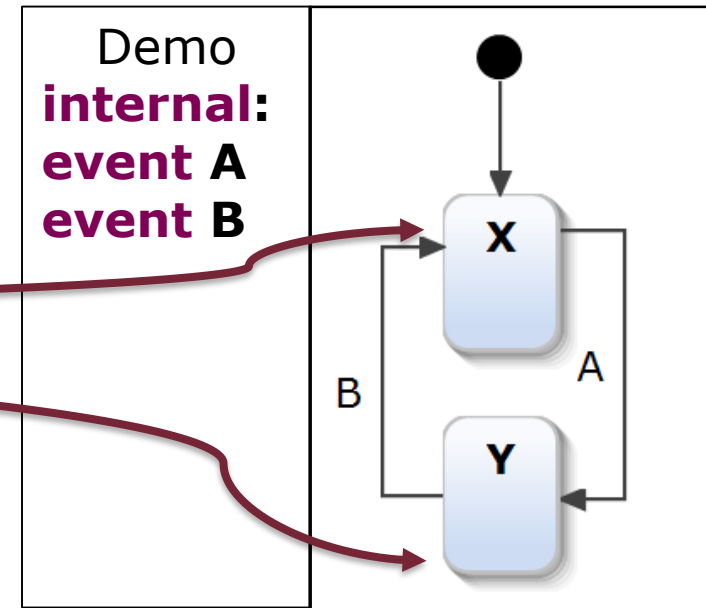
😊 Easier to write
😊 More clear to read
😊 Easier to modify
😞 +1 technology

Code Generator Example – States

■ Expected C-code:

```
//States of the statemachine  
enum states {  
    StateX,  
    StateY  
};
```

Possible states:
listed as an Enum



■ Template:

```
//States of the statemachine  
enum states {  
    «FOR state : states»  
        State«state.name»,  
    «ENDFOR»  
};
```

1. We iterate through all states
2. Their names will be written
(separated by commas):

State«Name» e.g.: StateX

Code Generator Example – Initial State

■ Expected C-code:

```
// The current state  
// First = entry state.
```

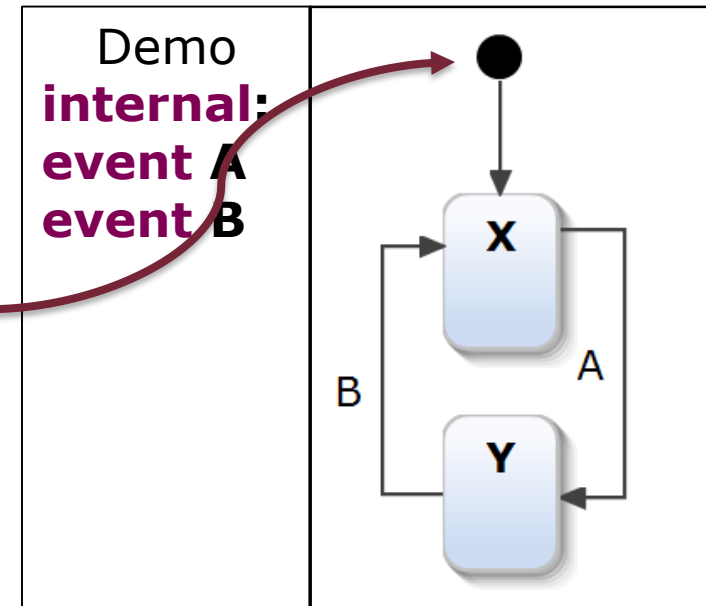
```
enum states actualState = StateX
```

current state = initial state

■ Template:

```
// The current state  
// First = entry state.
```

```
enum states actualState = State«findEntry(states).name»
```



1. We search for the initial element
2. Its name will be written

Code Generator Example – State Transitions

■ Expected C-code:

```
// Execute "A" event  
void doA{  
  switch(actualState) {
```

```
    case StateX:
```

```
      actualState = StateY;
```

```
      break;
```

```
    case StateY:
```

```
      break;
```

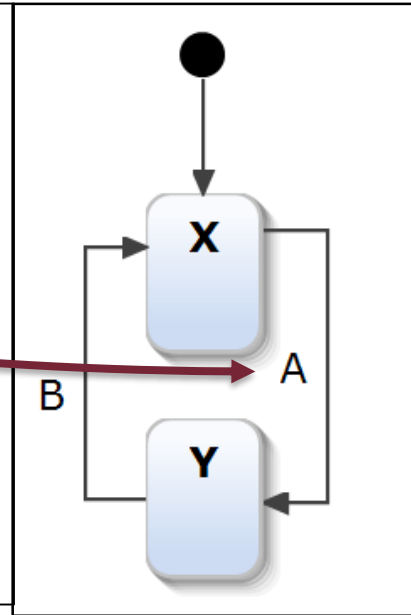
```
  }
```

```
}
```

A / X → Y

A / -

Demo
internal:
event A
event B



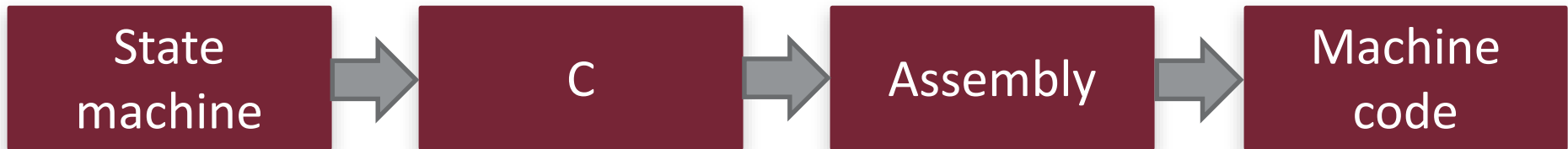
■ Template (sketch):

1. For each event a function
`do«NameOfEvent»`
2. Its content corresponds to the transitions (with guard, action)

For a (simple) state machine
the code generator is
nothing more than this!

Code Generator – Summary

- Code generation = translation
- Same steps:



- Working in the domain of the problem: Productivity ++
- Many boring, complex coding automatized Performance ++
- Checking in the domain of the problem: Reliability ++
- Projects at our research group: up to **95%** generated code

Code Generator – Summary

- Code generation = translation
- Same steps:

Prophecy:

State
machin

yesterday design pattern →

today function of the code generators →

tomorrow element of the modelling languages

machine
code

- Working in the domain of the problem: Productivity ++
- Many boring, complex coding automatized Performance ++
- Checking in the domain of the problem: Reliability ++
- Projects at our research group: up to **95%** generated code