Model execution for software development:
a case study with UML state machines in Java

ICAASE 2020 – Tutorial

Eric Cariou

Université de Pau et des Pays de l'Adour
Collège STEE – LIUPPA

November 2020

About me

- Assistant professor at the university of Pau, France
  - Member of the LIUPPA laboratory
  - Eric.Cariou@univ-pau.fr
  - http://ecariou.perso.univ-pau.fr/
- Research interests in software engineering
  - Software architecture
    - Components of communication
    - Integration of component, agent and service approaches
  - Model-driven engineering
    - Verification by contract
    - Model execution adaptation
    - Model execution with focus on business parts

Outline

- What is model execution in the MDE domain?
  - Basic example of executable DSL in EMF/Ecore
- PauWare tools
  - For executing UML state machines in plain Java
  - Association with business operations
  - Three concrete code examples
    - Two given and one exercice to do
- Resources for using PauWare in this tutorial
  - http://ecariou.perso.univ-pau.fr/ICAASE20/
  - No installation required
    - Use the Java IDE of your choice: Netbeans, Eclipse, IntelliJ, ...
    - A small JAR (140 kB) to import in a Java project

Executable models

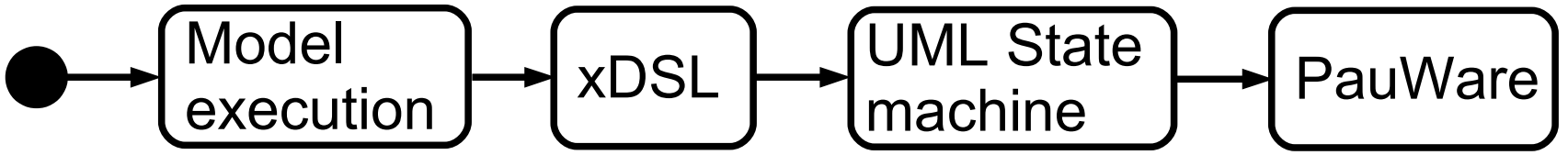
- Examples
  - State machines, activity diagrams, Petri nets, BPEL Web services orchestration …
- A (personal) definition for software development
  - An executable model defines the behavior of a system
  - Behavior = when, why and how calling business operations
    - UML state machine controlling a microwave oven
    - BPEL orchestration reserving a plane ticket calling Web services and making requests on databases
  - An executable model is evolving in time
    - Starting point
    - Execution step: from one point to another point

Executable DSL

- Model-Driven Engineering (MDE)
  - Models everywhere for everything!
  - Enable defining DSL (Domain Specific Language)
- Meta-model: definition of the DSL
  - Concepts of the DSL
  - Relations between the concepts
- Models of a DSL being executable
  - Requires an execution semantics
  - Express how the model is elvoving in time
- xDSL
  - DSL which models are executable

Process Definition Language

- Let define a basic DSL: PDL for Process Definition Language
  - A sequence of activities
  - Example (with an UML-style syntax)

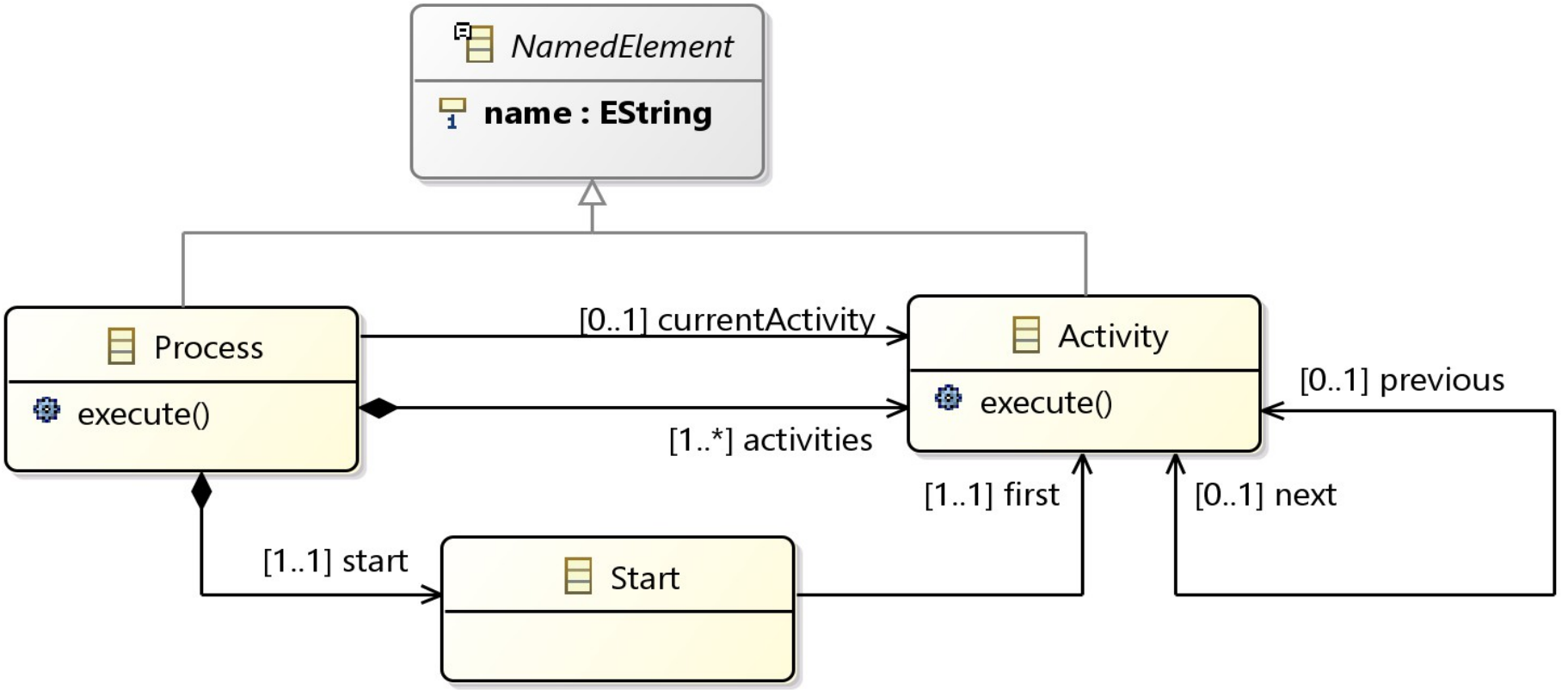


  - Concepts
    - Process, activity, start
  - Relations
    - A process contains activities
    - An activity has a previous and/or a next one

Meta-model

- Eclipse Modeling Framework (EMF)
  - Reference environment for MDE tools
  - Enable to define DSL with concrete syntax and programs to verify, manipulate, transform or execute models
  - Ecore: meta-meta-model of EMF
- An Ecore model defines the meta-model of a DSL
  - Through a simplified UML-like class diagram
  - A class = a concept of the DSL
  - Completed with OCL invariants for the well-formedness rules

Ecore meta-model of PDL

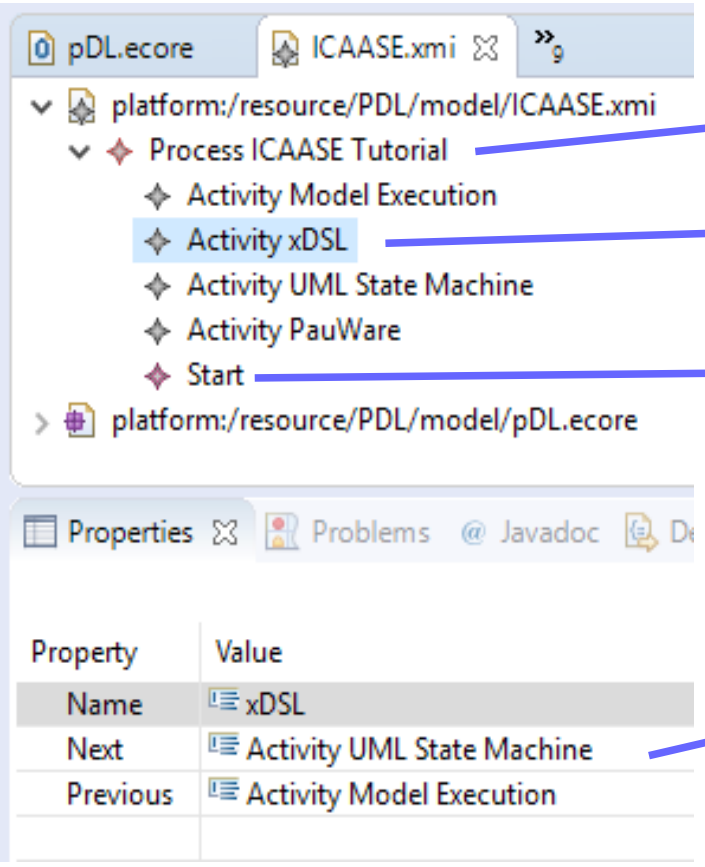


- One OCL invariant
  - The start activity has no previous activity:

```
context Start
inv: self.first.previous.oclIsUndefined();
```

Model of PDL

- To define a model conforming to PDL
  - Instantiate the elements of the meta-model
  - Model = set of instances of meta-elements
- Here, no concrete syntax is defined (but could be)
  - Use of the generic reflective model editor of EMF



The instance of `Process`

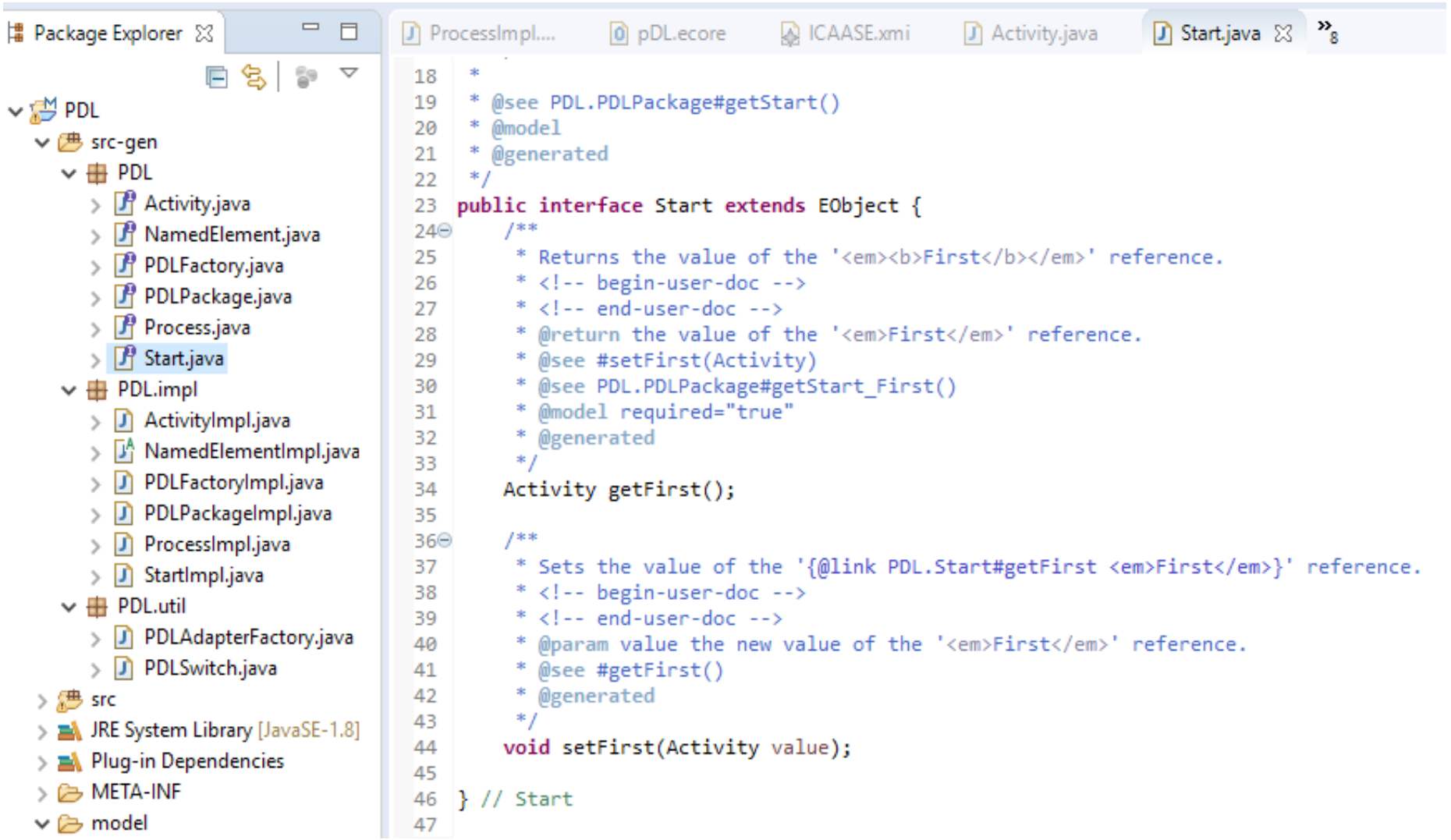
An instance of `Activity`

The instance of `Start`

Edition of the properties of the selected element (the "xDSL" activity)

Manipulation of models

- For an Ecore meta-model, EMF generates a set of Java interfaces and classes
  - Enable to read, save and manipulate models in Java



Execution of PDL process

- Implementation of the execution engine
  - With an operational semantics
  - Concretely : definition of the "execute" method of Process
    - In the `ProcessImpl` generated Java class

```
public void execute() {
   // print the name of the process
   System.out.println(" Process: " + this.getName());
   // get the first activity of the sequence
   Activity act = this.getStart().getFirst();
   // set the new value of the current activity
   this.setCurrentActivity(act);
   // follow the "next" reference through a while loop
   // until the end of the sequence
   while (act != null) {
      System.out.println(" -- " + act.getName());
      // get the next activity of the process
      act = act.getNext();
      // set this activity as the current one
      this.setCurrentActivity(act);
   }
}
```

Execution of the ICAASE model

- The execution for the ICAASE.xmi model prints in the console:

```
Process: ICAASE Tutorial
-- Model Execution
-- xDSL
-- UML State Machine
-- PauWare
```

- My execution engine works
  - It processes the activities following the sequence defined in the model
- But concretely what is done?
  - Nothing because I do not set what to do for an activity
  - Need to associate a business operation with an activity

xDSL and business operations

- For my model being the behavior of a concrete system
  - Business operations must be executed when going from an activity to another one
  - These operations could be regular Java methods with parameters and returned values
- Problem
  - My execution engine has no idea of what it is executed, it simply takes a model as parameter and executes it in a generic way:

```
Process proc = getProcessInFile("model/ICAASE.xmi");
proc.execute();
```

  - In [Cariou *et al.*, 2018], we explain how to attach and execute Java methods with xDSL models

Eric Cariou, Olivier Le Goaer, Léa Brunschwig and Franck Barbier, *"A generic solution for weaving business code into executable models"*, 4th International Workshop on Executable Modeling at MoDELS (EXE 2018), CEUR Workshop Proceedings, vol. 2245, October 2018

Interests of executable models

- Many interests of executable models
  - The explicit definition of the behavior of the system at a high level of abstraction
  - The executed model in the running system is the one defined at design
    - Seamlessness software development
    - Early detection of problems by simulating the model at design
- For the business part
  - We argue that programming them in regular language remains the best way
    - See our ICAASE paper: *"A software development process based on UML state machines"*

DSL versus GPL

- Classic debate
  - DSL: Domain Specific Language
    - Defined with EMF as seen and/or tools as the GEMOC studio
  - GPL: General Purpose Language
    - In the modeling context: UML

	DSL	GPL
Pros	• Define exactly what you need • Build your specific tools	• Well-known languages • Easier for sharing information or specification • Existing tools
Cons	• Often build everything from scratch (inc. user-friendly editors) • Dependency with the Eclipse/EMF-ecosystem	• The language must match your needs • More difficult to build specific tools

UML state machines

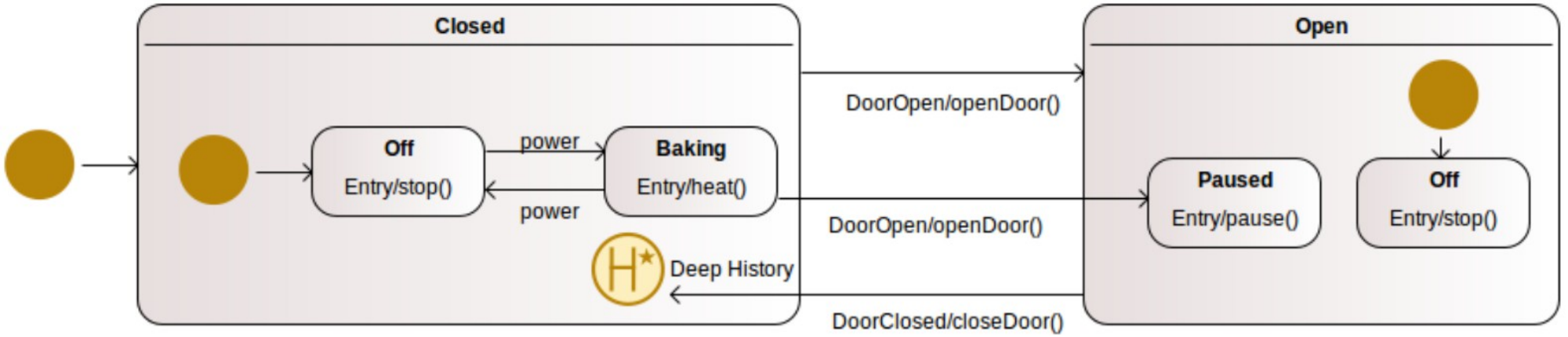
- In this tutorial
  - Software development with UML executable state machines
- UML: Unified Modeling Language
  - Reference modeling language in software enginnering
    - And in other domains
  - Semi-formal and graphical notation easy to use for software engineers
  - Standard of the OMG
- State machines
  - Well-known formalism for specifying event-based behavior
  - Original state machines: Harel's statecharts

    David Harel, *Statecharts: a visual formalism for complex systems*, Science of Computer Programming, 8(3), 1987

Microwave oven example

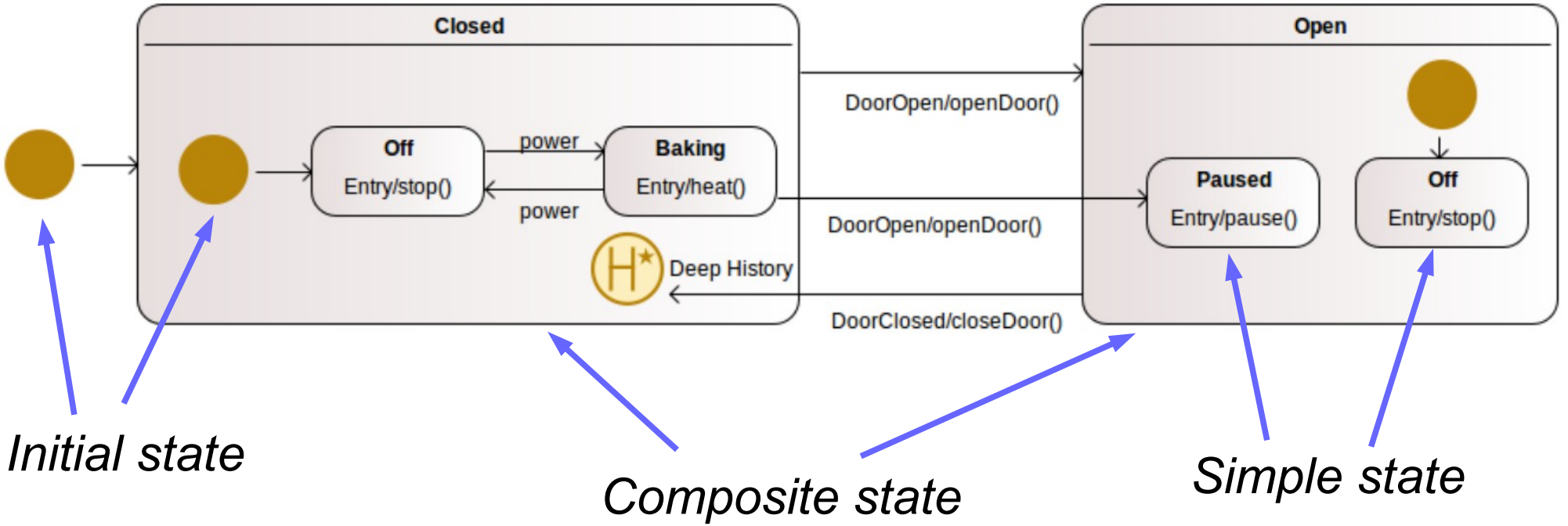
- Specification of the behavior of a microwave oven
  - The door is closed or open
    - This defines states
  - An open door can become closed and vice-versa
    - This defines events and transitions
  - The microwave can be activated for heating food
    - This defines business operations
  - When I open the door and close it, I want to get back in the previous mode of the oven
    - Either doing nothing or heating
    - This defines an history
  - When the oven is plugged, it is put in an off mode
    - This defines a starting state

Microwave oven example



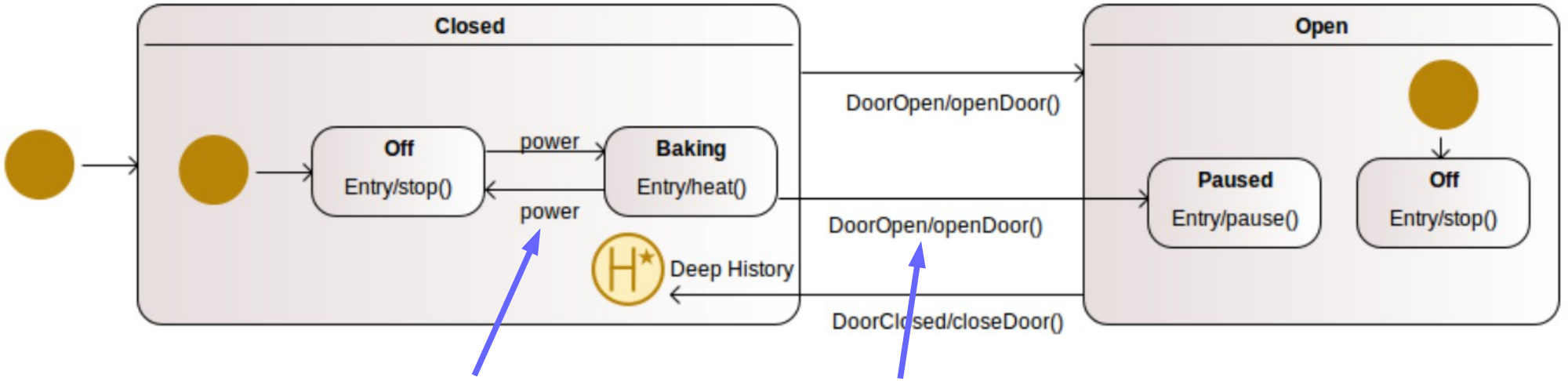
- A lot of a tools exist for editing UML diagrams
- Here, we use Modelio, open source version
  - https://www.modelio.org/

Microwave oven example



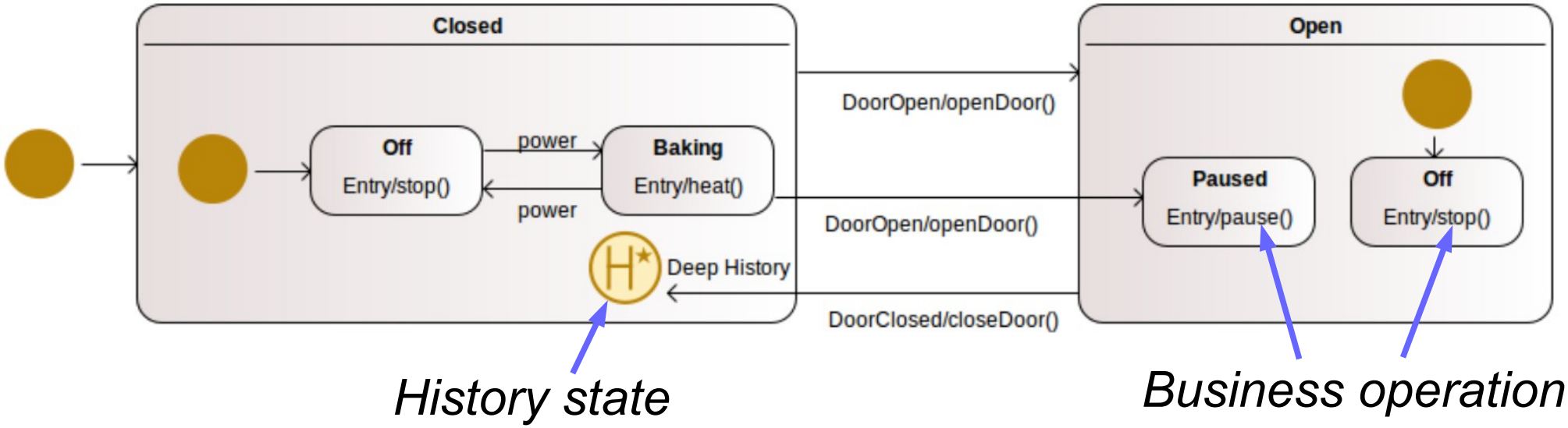
- Hierarchical definition of states
  - Composite and simple states
  - A composite state has an initial state
    - The state machine also
  - Possibility to have parallel regions of states

Microwave oven example



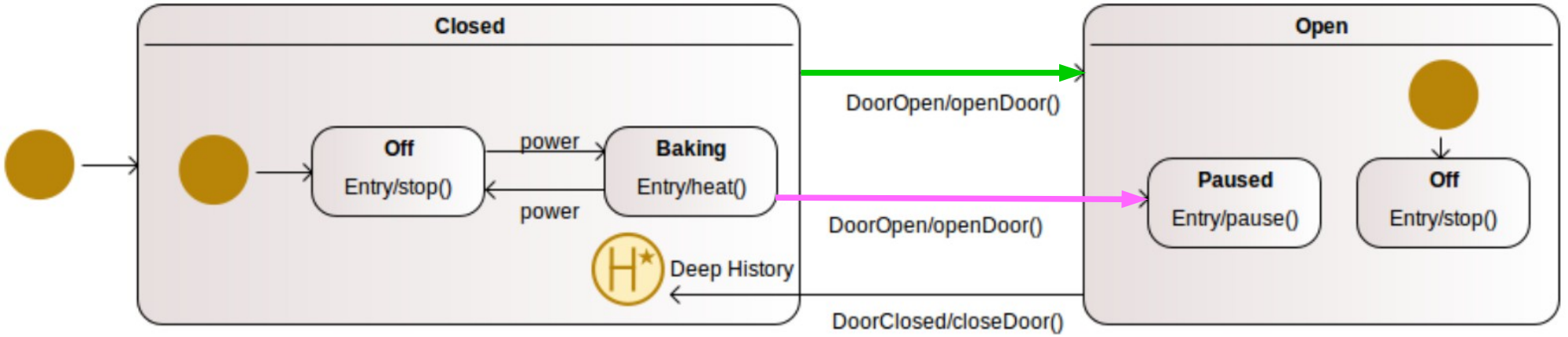
- Transitions
  - Syntax: [guard] event / operation()
  - Mandatory elements
    - An event, a source and a target states
  - Optional
    - A guard (boolean expression)
    - A business operation

Microwave oven example



- Business operations associated with states
  - Entry: when entering in the state
  - Exit: when leaving the state
  - Do: when being in the state
- History state
  - The concrete target of the transition is the last active state of Closed
  - If none, is by default the initial state

Microwave oven example



- Execution semantics
  - Express how the state machine is evolving when events occur
  - Question: "Baking" is the active state and the "DoorOpen" event occurs, which transition is followed? The pink or the green?
    - Depends of the execution semantics
    - UML: the most specific (pink)
      - Lead to the "Paused" state
    - Harel: the most general (green)
      - Lead to "Off" as the initial state of "Open"

PauWare tools

- PauWare engine
  - API for programming UML state machines in plain Java
    - Associated with Java business operations
  - Execution engine
    - Process events and make the state machine evolving
    - Execute the required business operations
- Addons
  - PauWare code generator
    - Model compilation: translational semantics (UML → PauWare)
    - Web application generating PauWare code from a UML model
  - PauWare viewer
    - Draw in a Web browser the state machine under execution
- https://pauware.univ-pau.fr/

Execution semantics of PauWare

- Specification of the UML state machine execution semantics
  - Informally in the UML specification of the OMG
    - Last version, 2.5.1, 2017: https://www.omg.org/spec/UML/2.5.1/
  - But recently, formally in the Precise Semantics of State Machine (PSSM) specification
    - Version 1.0, 2019: https://www.omg.org/spec/PSSM/1.0/
- Execution semantics of PauWare
  - PauWare implemented several years ago, before PSSM
  - PauWare implements the informal UML semantics
    - Almost all concepts of UML state machines are in PauWare
    - Some choices have been made for secondary ambiguous semantics point

Software development with PauWare

- PauWare engine
  - Standalone and lightweight
    - JAR file (140 kB size)
    - No dependency with specific IDE or frameworks
  - Can be used in any Java development
    - Java EE, Java SE, Java ME, Android, …
  - Business operations are implemented in plain Java
    - The link with the PauWare state machine is straigthforward
  - PauWare code of the state machine
    - Written by hand or generated from UML diagrams
- Limits
  - No integration with design tools (for simulation or early verification for instance)

Weaving of business operations

- Link with the business operations when defining states and transitions
  - Three elements to set for an operation
    - The object on which the Java method is called
    - The name of the method in a `String`
    - An array of `Object` containing the parameters (optional)
  - The method is executed by a dynamic call through the reflection mechanisms of Java
    - No compilation error if you define the call of a method that does not exist in the class of the object
- Same principle for defining
  - A guard or an invariant
  - Both are Java methods returning a `boolean`

Implementation of the microwave

- Two Java classes
  - One implementing the business part of the microwave
    - Basic implementation: 3 boolean attributes
      - Light : the light of the microwave (on or off)
      - Door : open or closed
      - Magnetron : heating or off
    - Business methods modifying these attributes
  - One implementing the state machine
    - Definition of the hierarchy of states and transitions
    - Link with the business operations of the business class
- For understanding the classes of the PauWare API
  - Javadoc : https://pauware.univ-pau.fr/assets/Javadoc_files/

Business class of the microwave

```
public class MicrowaveBusiness {

    private boolean lightOn = false;
    private boolean doorOpen = false;
    private boolean magnetronOn = false;

    public void stop() {
        lightOn = false;
        magnetronOn = false;
    }

    public void heat() {
        lightOn = true;
        magnetronOn = true;
    }

    public void pause() {
        magnetronOn = false;
        lightOn = true;
    }

    public void openDoor() {
        doorOpen = true;
    }

    public void closeDoor() {
        doorOpen = false;
    }

    public String toString() {
        return "[Light on: "+lightOn+", magnetron on: "+magnetronOn+", door open: "+doorOpen+"]";
    }
}
```

Microwave: PauWare state machine

```
public class MicrowaveStateMachine {

    // The states of the state machine
    protected AbstractStatechart open;
    protected AbstractStatechart closed;

    protected AbstractStatechart offOpen;
    protected AbstractStatechart offClosed;
    protected AbstractStatechart baking;
    protected AbstractStatechart paused;

    // The state machine
    protected AbstractStatechart_monitor stateMachine;

    // The business object associated with the state machine
    protected MicrowaveBusiness mwb;

    public void buildAndStartMicrowave() throws Statechart_exception {

       // The state off (of open) executes the "stop" method as entry
       // and is  the input state of its composite state
       offOpen = new Statechart("Off");
       offOpen.set_entryAction(mwb, "stop");
       offOpen.inputState();
       ...
```

Microwave: PauWare state machine

```
...
offClosed = new Statechart("Off");
offClosed.set_entryAction(mwb, "stop");
offClosed.inputState();

baking = new Statechart("Baking");
baking.set_entryAction(mwb, "heat");

paused = new Statechart("Paused");
paused.set_entryAction(mwb, "pause");

// The closed state is a composite containing the off and baking states,
// has a deep history pseudo state and is the input state of its composite
// (that is the state machine)
closed = offClosed.xor(baking).name("Closed");
closed.deep_history();
closed.inputState();

open = offOpen.xor(paused).name("Open");

// Build the global state machine as being composed of the open
// and closed states
stateMachine = new Statechart_monitor(closed.xor(open),"Microwave", true);
...
```

Microwave: PauWare state machine

```
        ...
        // Basic transitions for the Power event
        stateMachine.fires("Power", offClosed, baking);
        stateMachine.fires("Power", baking, offClosed);
        // Transitions with a business action for managing the door
        stateMachine.fires("DoorOpen", closed, open, true, mwb, "openDoor");
        stateMachine.fires("DoorOpen", baking, paused, true, mwb, "openDoor");
        // Implicit transition towards the history of Closed
        stateMachine.fires("DoorClosed", open, closed, true, mwb, "closeDoor");

        // Start the state machine
        stateMachine.start();

        // A file tracer can be attached to the statemachine: see the code
    }

    // Process the event passed as parameter: trigger the
    // transitions (if any) and executes all required business operations
    public void runEvent(String name) throws Exception {
        // Run to completion cycle: executes everything required
        stateMachine.run_to_completion(name);
        System.out.println("Business object after "+name+ " "+mwb);
    }
}
```

PauWare microwave execution

- When executing this code:

```
MicrowaveBusiness business = new MicrowaveBusiness();
MicrowaveStateMachine sm = new MicrowaveStateMachine(business);
sm.buildAndStartMicrowave();
sm.runEvent("Power");
sm.runEvent("DoorOpen");
sm.runEvent("Power");
sm.runEvent("Foo");
sm.runEvent("DoorClosed");
sm.runEvent("Power");
sm.stop();
```

- Print this execution trace:

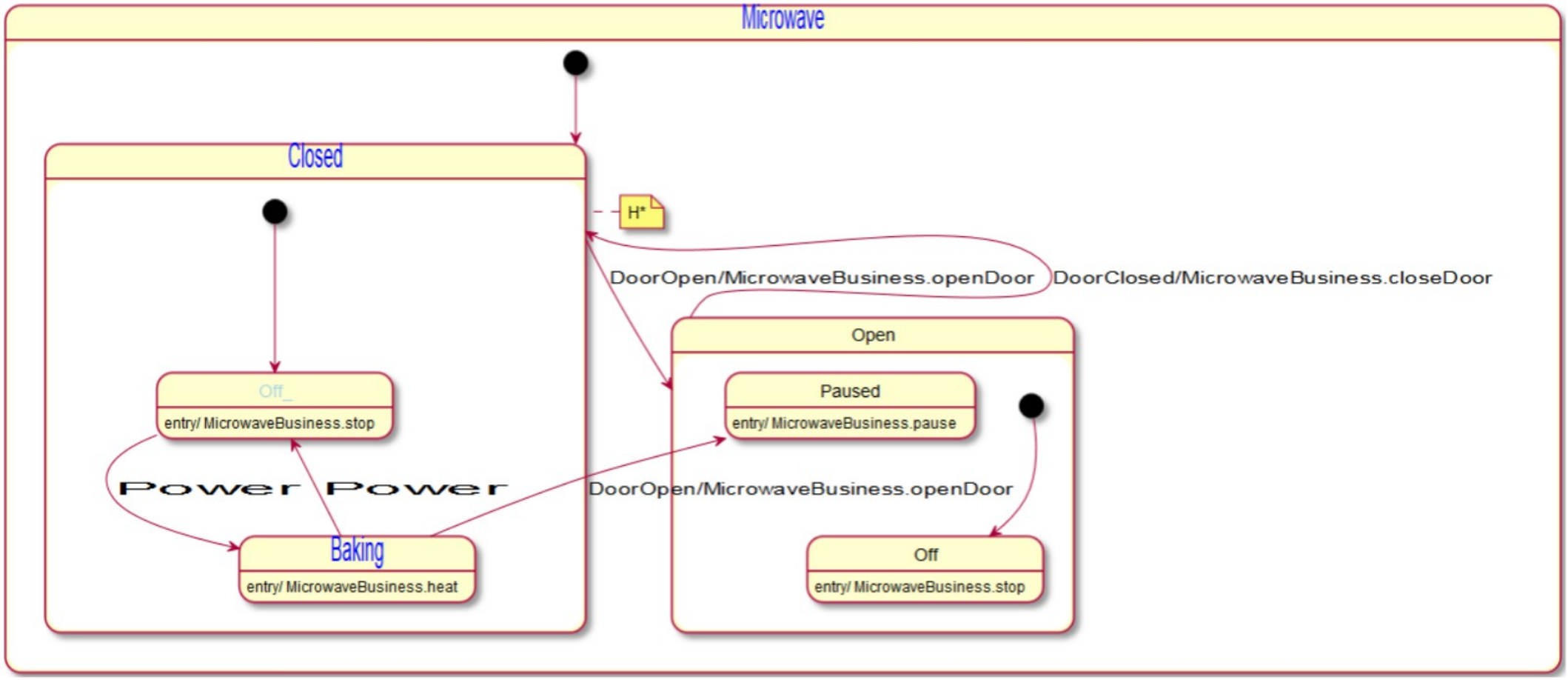
```
Business object after Power [Light on: true, magnetron on: true, door open: false]
Business object after DoorOpen [Light on: true, magnetron on: false, door open: true]
Business object after Power [Light on: true, magnetron on: false, door open: true]
Business object after Foo [Light on: true, magnetron on: false, door open: true]
Business object after DoorClosed [Light on: true, magnetron on: true, door open: false]
Business object after Power [Light on: false, magnetron on: false, door open: false]
```

PauWare microwave program

- Download the sources and try the program
- http://ecariou.perso.univ-pau.fr/ICAASE20/

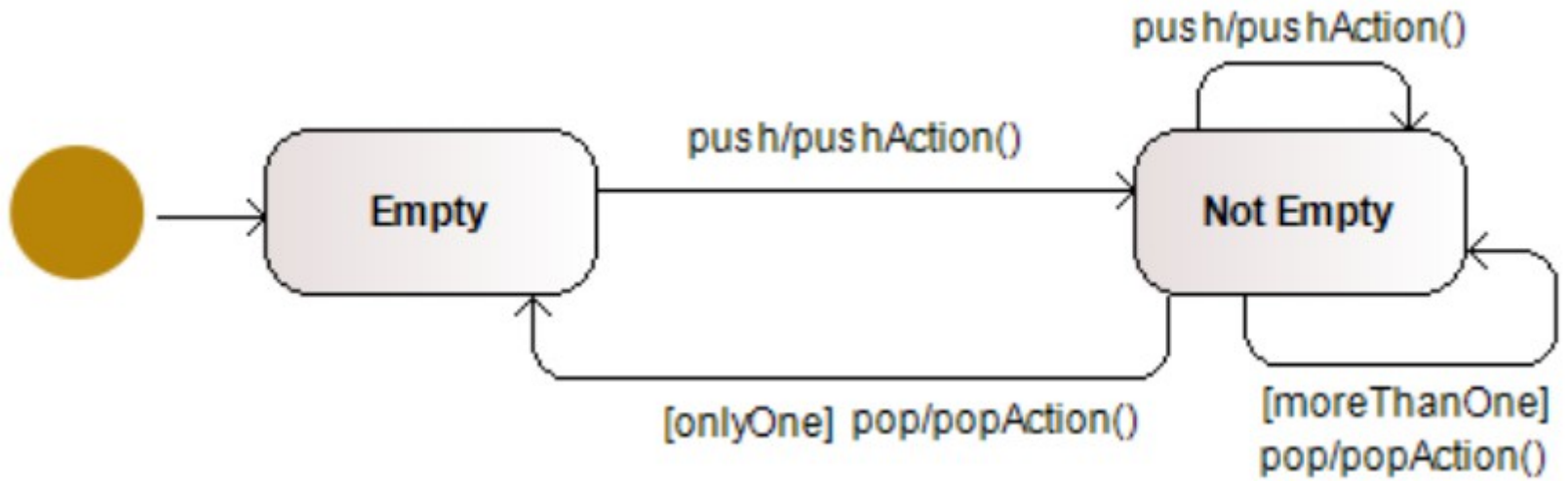
PauWare viewer

- Experimental (and not maintained) tool
  - Draw the PauWare state machine under execution in a Web browser
  - We clearly retrieve the microwave state machine



Stack state machine

- Interests of this example
  - Guards and invariants
  - Business operations with parameters and returned values
- Stack state machine
  - Two basic states : empty or not
  - Two events : push or pop (a value)
  - The business object is a Java stack of String



Stack state machine: invariant

- Invariant
  - Constraint to be ensured during the state machine execution
  - Define with a Java method returning a boolean
- Example: in the empty state, the Java stack is empty
  - The invariant method

```
public boolean stackEmpty() {
   return stack.isEmpty();
}
```

  - Association of the invariant with the state empty

```
empty = new Statechart("Empty");
// the invariant method will be called on the current object
empty.stateInvariant(this, "stackEmpty");
```

  - Verification of the invariants after a run to completion cycle

```
stateMachine.run_to_completion("pop", AbstractStatechart_monitor.Compute_invariants);
```

Stack state machine: guard & bus. op.

- The transition from Not Empty to Empty for the event "pop"
  - A guard: only one element to go to the Empty state

```
public boolean onlyOne() {
   return stack.size() == 1;
}
```

  - A business operation : `actionPop()`

```
public String actionPop() {
   String value = stack.pop();
   return value;
}
```

  - Then the definition of the transition (both methods are called on the current object)

```
stateMachine.fires("pop", notEmpty, empty, this, "onlyOne", this, "actionPop");
```

Control and data flows

- When the "pop" event occurs and the stack is not empty
  - The `actionPop()` is called and returned the top of the stack
- But who is getting the returned value?
  - No one!
- In classic imperative programming, we can write

```
String res = this.actionPop();
if (res==null) System.err.println("Null value");
else myFileObject.writeObject(res);
```

  - The control flow is the sequence of operations and the `if … then … else` statement
  - The data flow is the variable `res` returned from `actionPop()` and passed as parameter to `writeObject()`
  - The control and the data flows are mixed

Control and data flows

- With executable models
  - Control flow = the behavior reified in the model
- Our two kind of models
  - State-machine: event-based
    - When an event occurs, I do something
  - DSL of PDL: ordered sequence
    - When an activity is finished, I start the next one
    - We could also have executed business operations in activities as in states and transitions with PauWare
  - In none, it is/can be (directly) expressed how the data are going from one state/activity to another one
- Executable models + business operations
  - Clear separation of concerns but can be a too big separation!
  - Solution: data shared within a common object (see the ICAASE and [Cariou *et al.,* 2018] papers)

Parameters of business operations

- Business operations can have parameters
  - In PauWare, it is passed as an array of `Object`
- If parameters change
  - For states: redefine the entry/exit/do action with the new parameters
  - For transitions: redefine the complete transition
    - Concretely, the transition is not redefined, it is detected that the same transition exists, it only changes the parameters
- The "push" event with the value to push on the stack

```
public void pushEvent(String value) {
   // The transitions are redefined for putting the value as parameter when calling the "actionPush" method
   stateMachine.fires("push", empty, notEmpty, true, this, "actionPush", new Object[] { value });
   stateMachine.fires("push", notEmpty, notEmpty, true, this, "actionPush", new Object[] { value });
   // Now the push event can be processed
   stateMachine.run_to_completion("push", AbstractStatechart_monitor.Compute_invariants);
}
```

Parameters of business operations

- This way of passing parameters
  - Justify the implementation choice to have a method for processing an event
  - Because of the need to redefine the transitions and/or business operations of states
- Too complex?
  - PauWare v2 enables to redefine only the parameters of the business operations
  - Remember that the PauWare code generator can generate this code for you from a UML model
- The same logic is also applied to guards and invariants methods

Stack & car state machines

- Try the code of the stack state machine
- And then, you can implement the car state machine

Conclusion

- Executable models
  - Clear separation between the behavior and the business parts
  - The behavior is defined at a high level of abstraction in a model
  - Same model at design and runtime
    - Seamlessness development
- PauWare
  - Set of tools for executing UML state machines
  - Lightweight library
    - Can be used for any Java development
  - Business operations defined at standard Java code
    - Most efficient and suitable way to do it