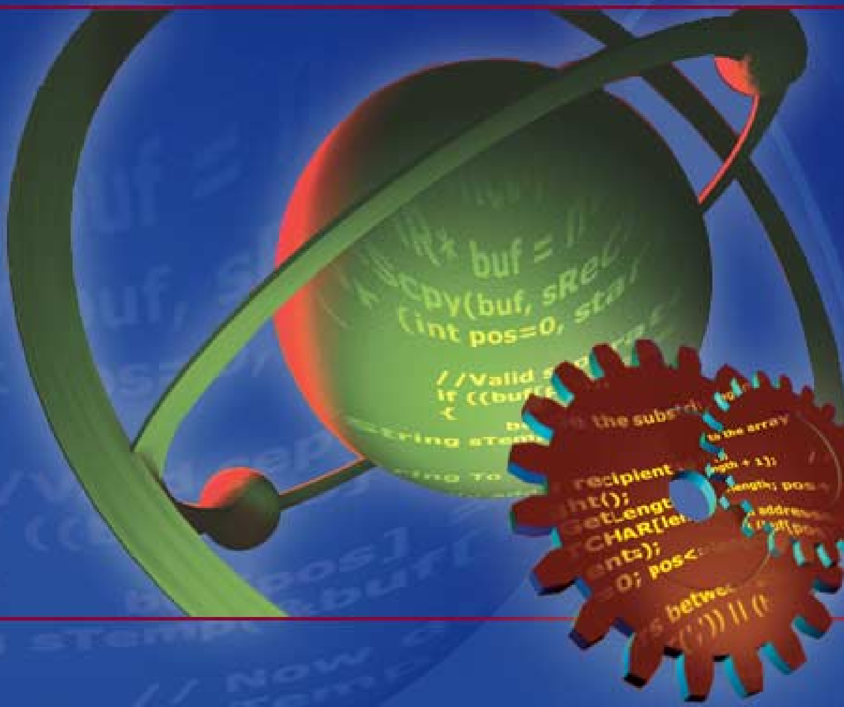
Building and Implementing Concurrent Specifications

Stephen J. Mellor

Chief Scientist

Cortland D. Starrett

Engineering Manager



Accelerated Technology®

A Mentor Graphics Division

A simple embedded system

Set the
cooking time,
close the door
press cook.

It's that easy!

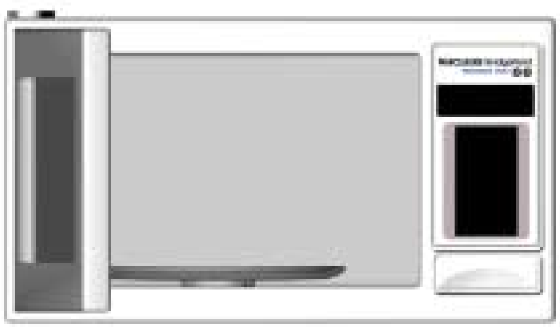
But... for safety, the door *must* be closed and the cooking time set.

If the door is opened, cooking *must* stop.

And there are some details about timers and power levels...... And.... And.... And....

Describing behavior

What happens when you push the Start button?



If the door is open, nothing.

But if the door is closed, start cooking

Why?

Why are concurrent specifications necessary?

- Because our problems *are* concurrent
- Concurrency problems—synchronization problems and race conditions—exist in the problem as well as the implementation.
- So we may understand the problem well enough to be able to establish priorities among tasks
- So we may maximize precious system resources when many things can happen at once.

Compelling reasons…

Systems are complex, so we must:

- Express them at a high level of abstraction
- Adapt easily to hardware and platform changes
- Develop concepts rapidly
- Test ideas with confidence
- Understand performance before implementation
- Reduce required code by identification of cross-cutting commonality
- Maximize parallel development

A solution is…

We need to specify systems in a language that is:

- Concurrent: assume the system is distributed
- Complete: not a blueprint to be filled in
- Executable, and
- Translatable: so the specification can be mapped into an implementation.

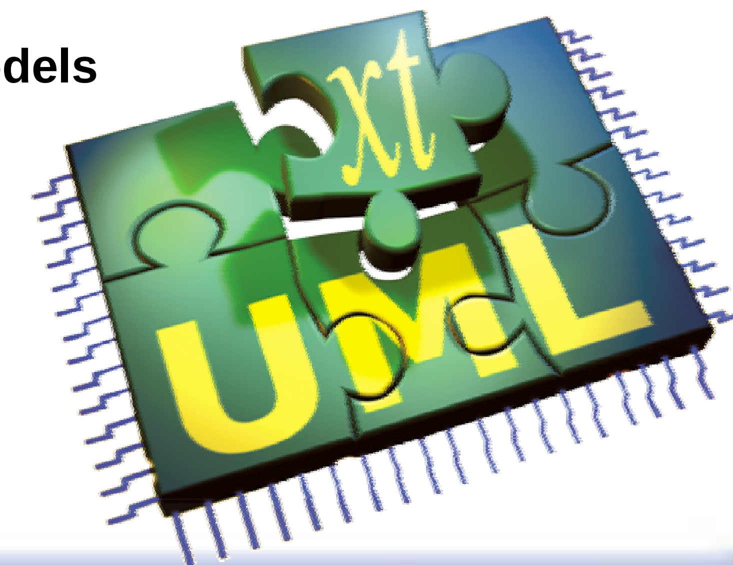
The knowledge **The software**

xtUML

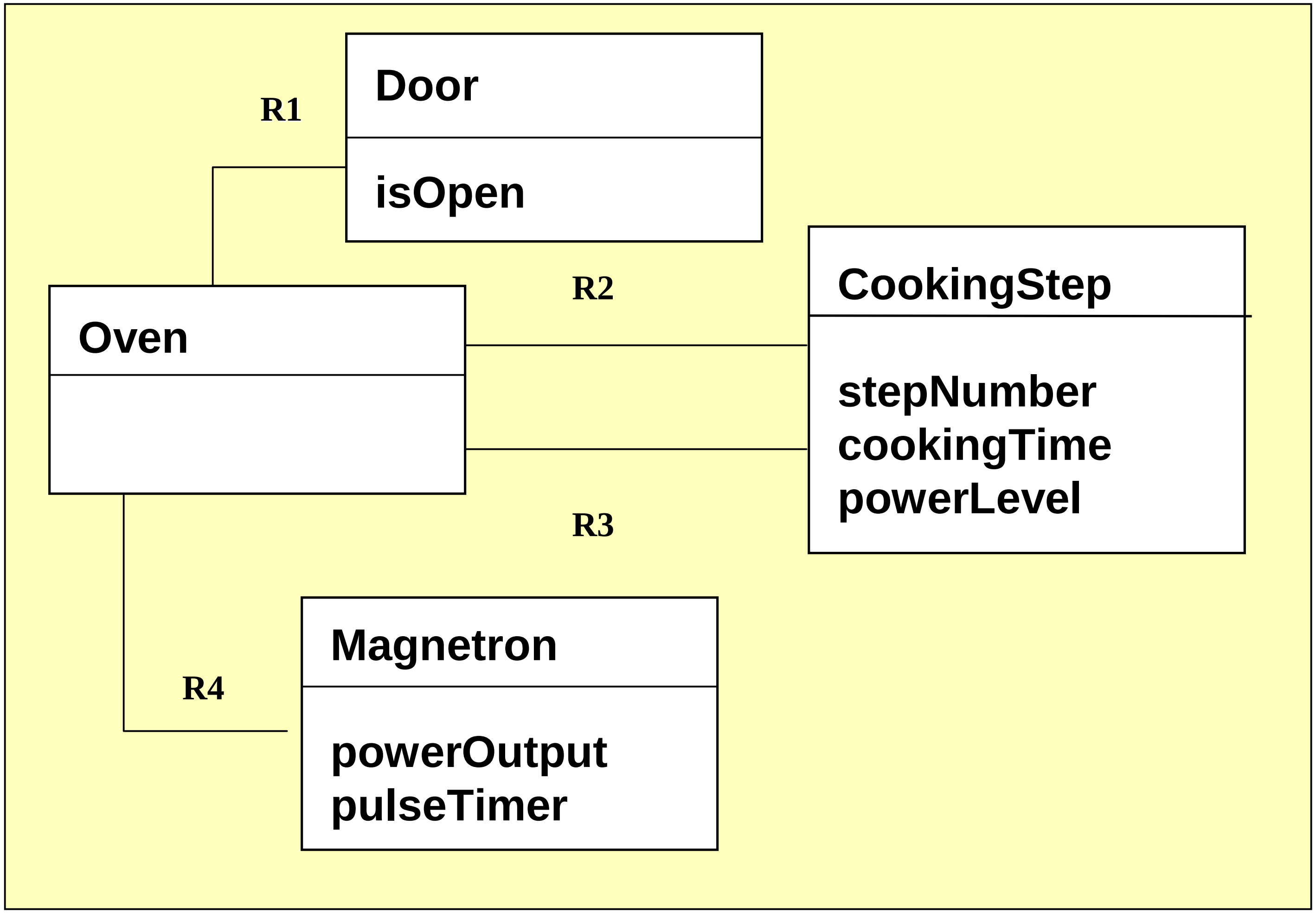
xtUML is a streamlined subset of the UML industry standard that:

(x) Executes models
- **Allows for early verification**
- **Pre-code interpretive execution**
- **Integration of legacy code**

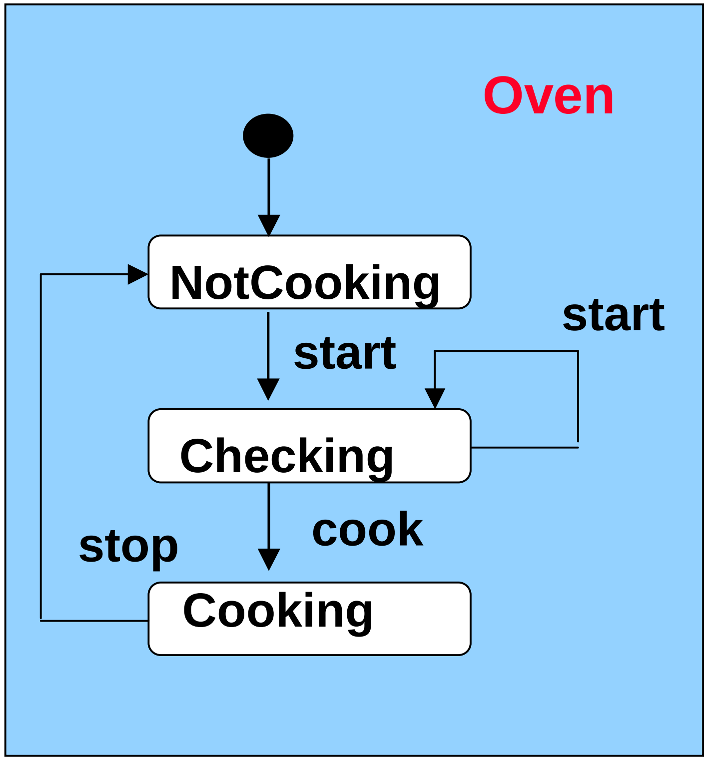
(t) Translates models
- **Complete code generation from models**
- **Customizable compilation rules**
- **Optimized code**

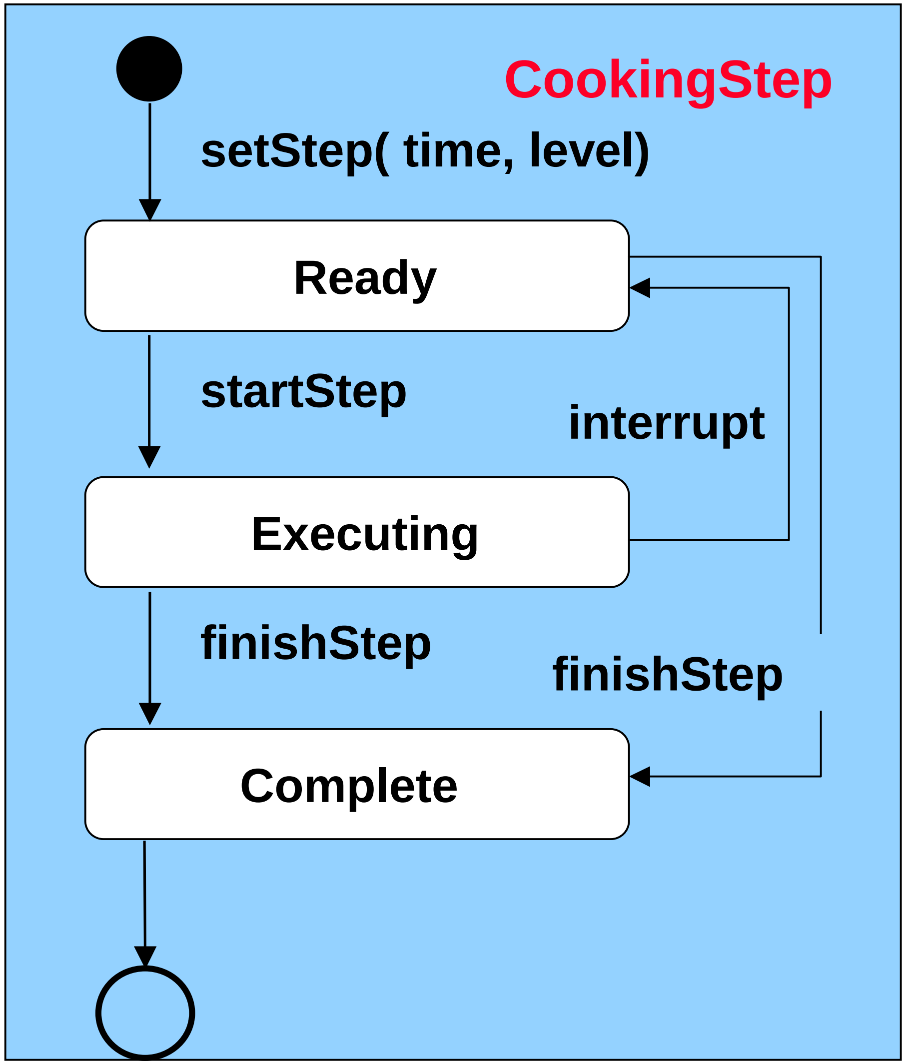


Class Diagram



State Machines

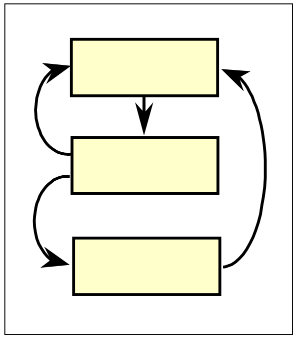




Activities

- There is one *activity* for each state in the state machine
- Each state machine is one state at a time
- All actions in the activity must terminate before the next state can be entered

State Machine



State Charts

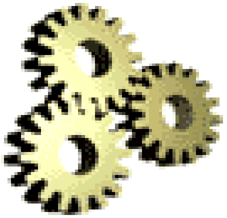
Activity Models

```
select one magnetron
  related by oven->[R4];
magnetron.powerOutput =
  self.powerLevel;
generate powerOn to
  magnetron;
```

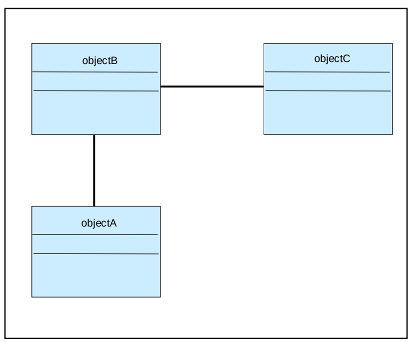
Action Language

Executable elements

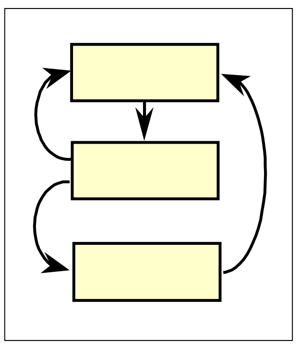
- Data
- Control
- Processing



Class Model



State Models



Activity Models

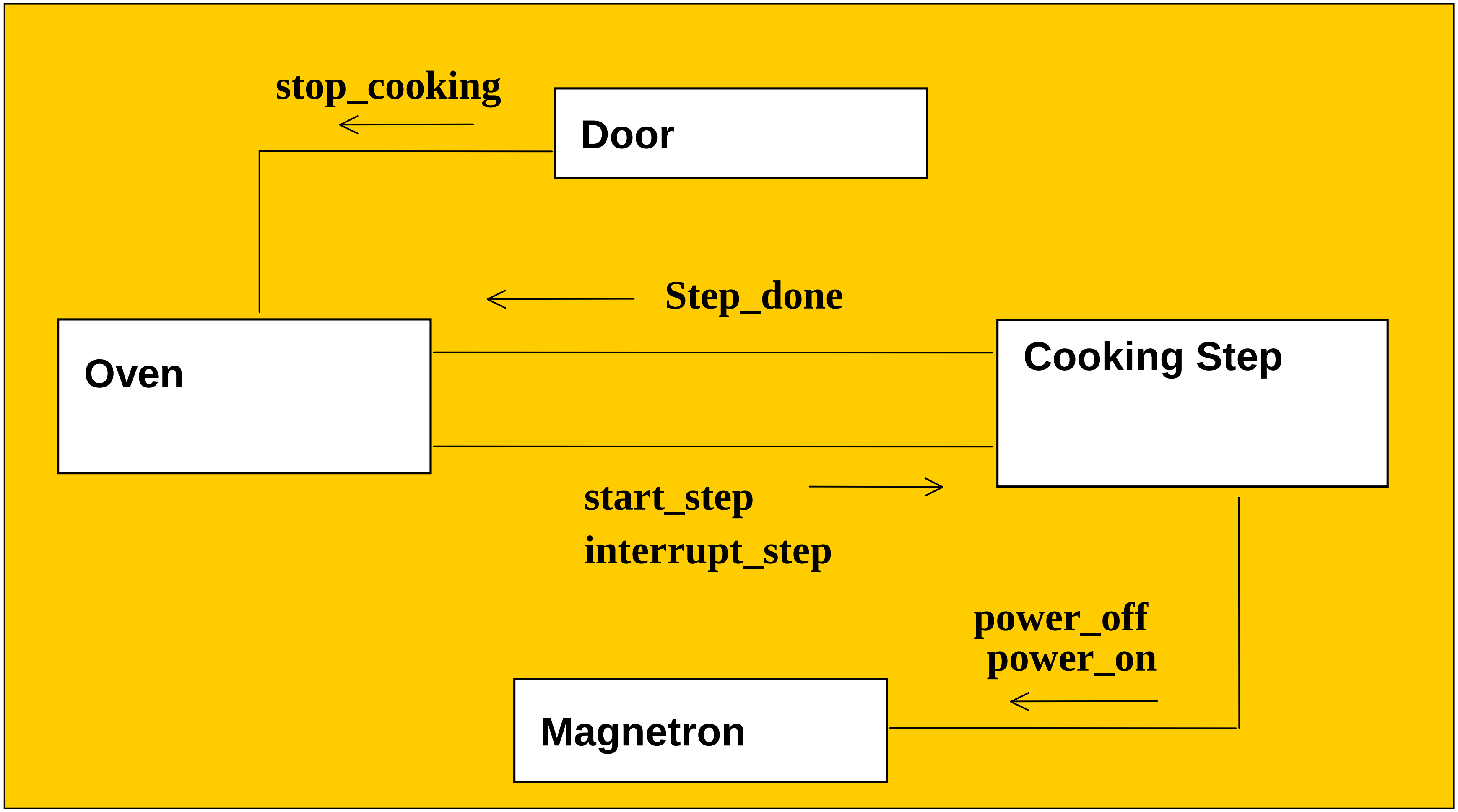
```
select one magnetron
  related by oven->[R4];
magnetron.powerOutput =
  self.powerLevel;
generate powerOn to
  magnetron;
```

Class Diagram

State Charts

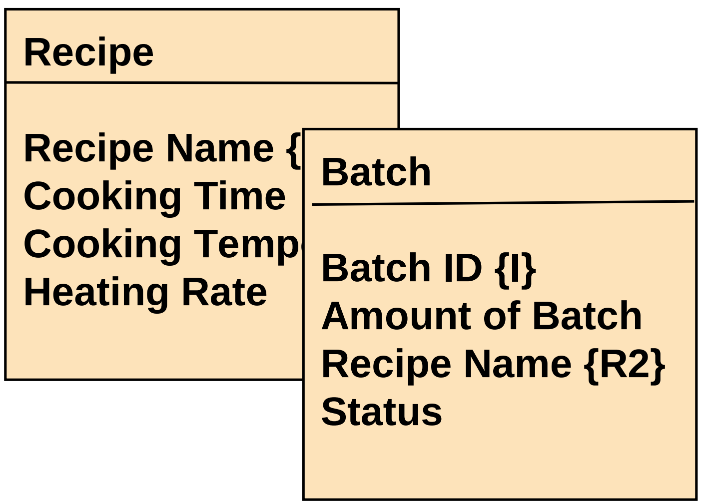
Action Language

Communication Diagram



Instances

An executable model operates on data about instances.

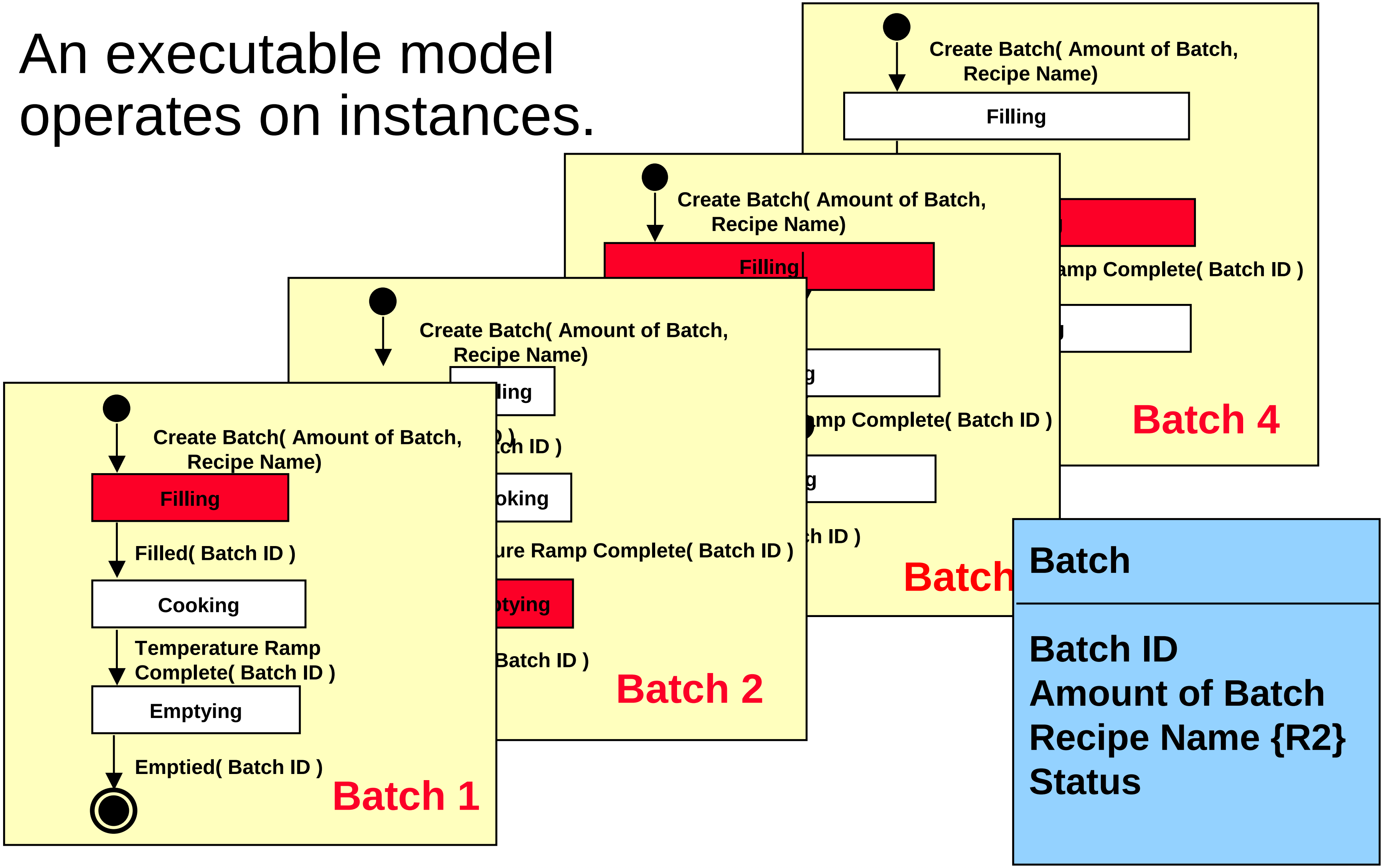


Recipe			
Recipe Name	Cooking Time	Cooking Temp	Heating Rate
Nylon	23	200	2.23
Kev			
Stu			

Batch			
Batch ID	Amount of Batch	Recipe Name	Status
1	100	Nylon	Filling
2	127	Kevlar	Emptying
3	93	Nylon	Filling
4	123	Stuff	Cooking

Instances

An executable model operates on instances.

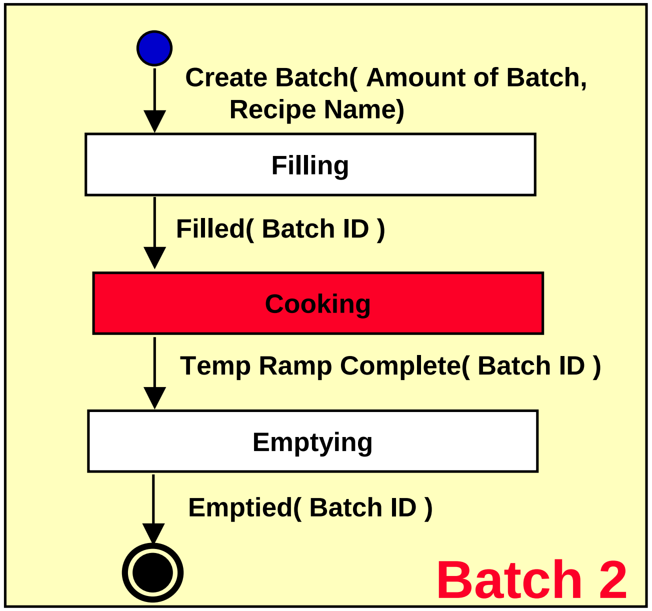


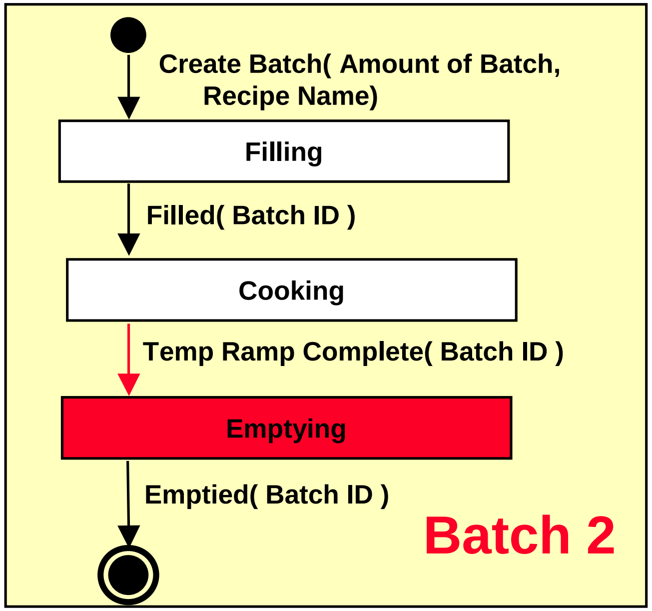
Execution

Batch			
Batch ID	Amount of Batch	Recipe Name	Status
1	100	Nylon	Filling
2	127	Kevlar	Emptying
3	93	Nylon	Filling
4	123	Stuff	Cooking

The lifecycle model prescribes execution.

When the Temperature Ramp is complete, the instance moves to the next state....and executes actions.

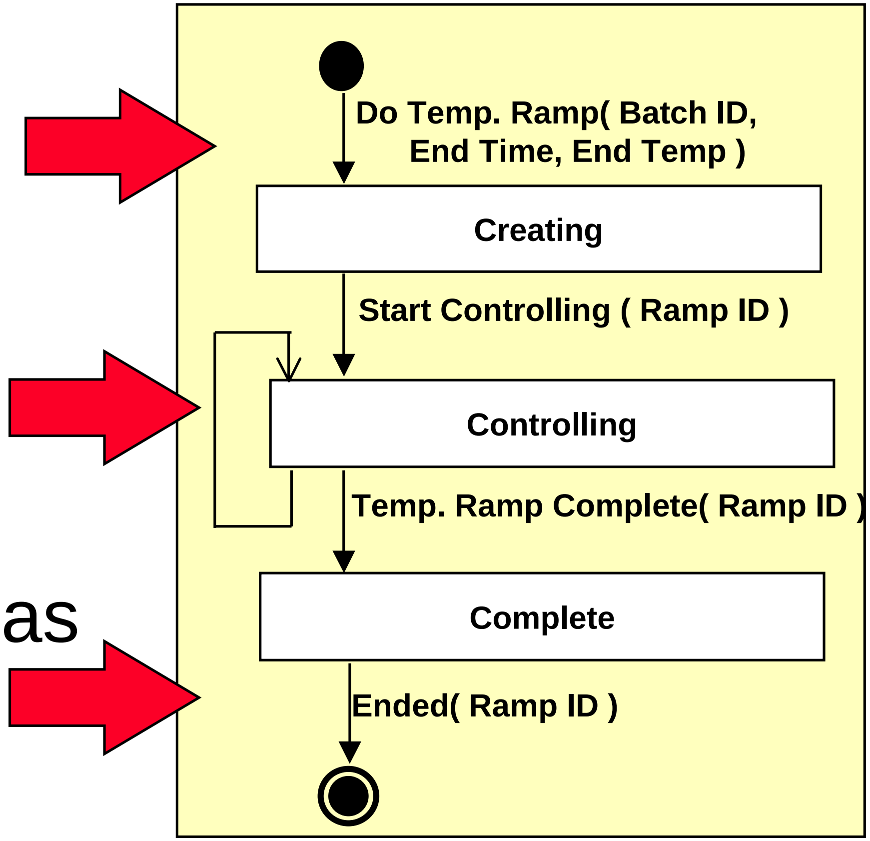




Executing the Model

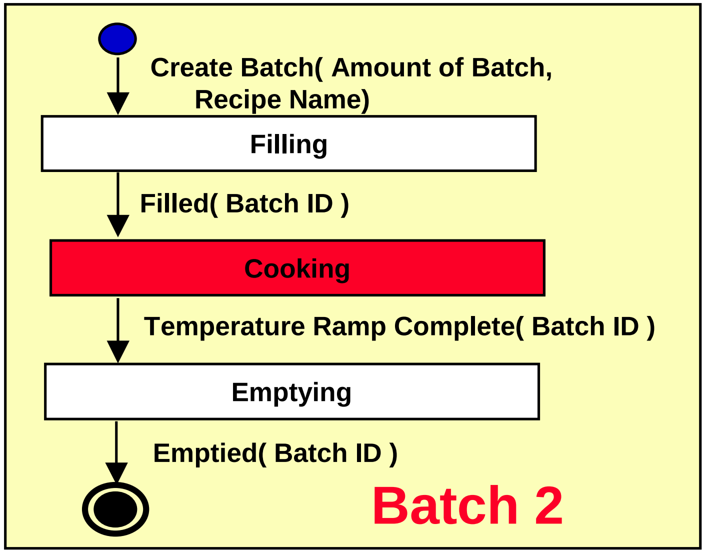
The model executes in response to signals from:

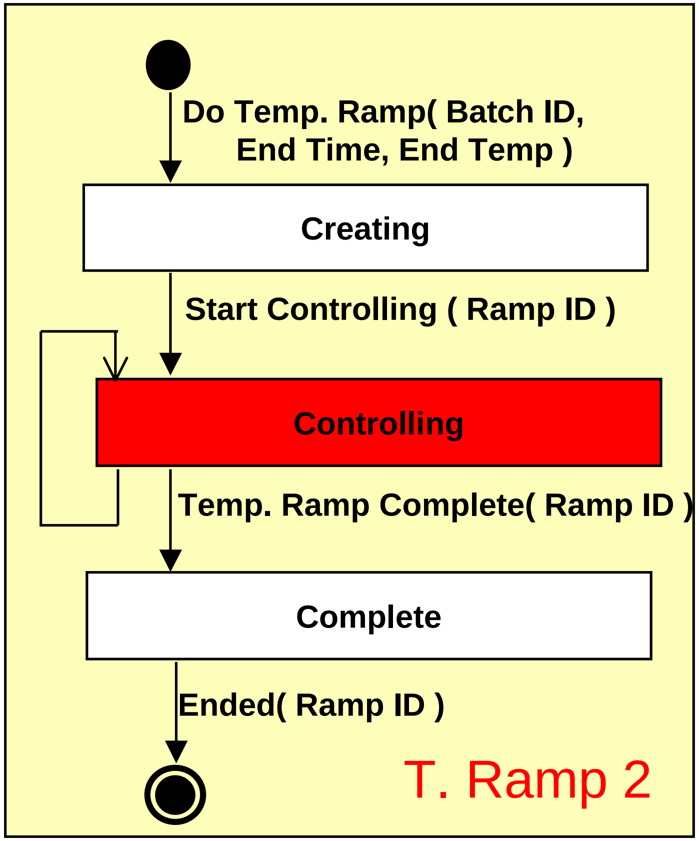
- the outside
- timers
- other instances as they execute

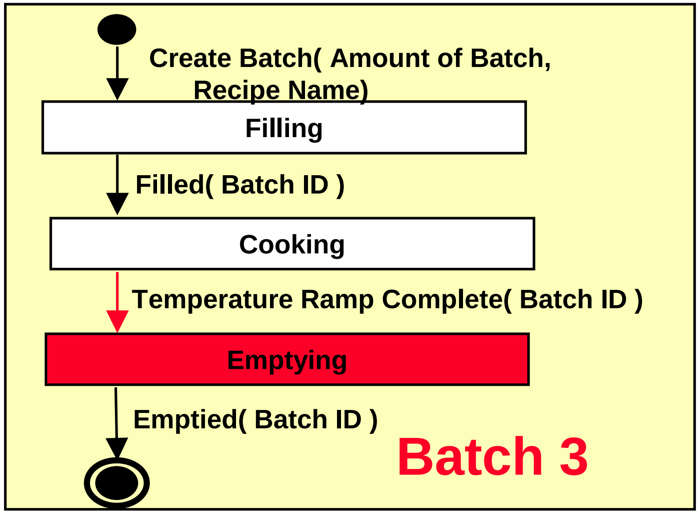


Concurrent Execution

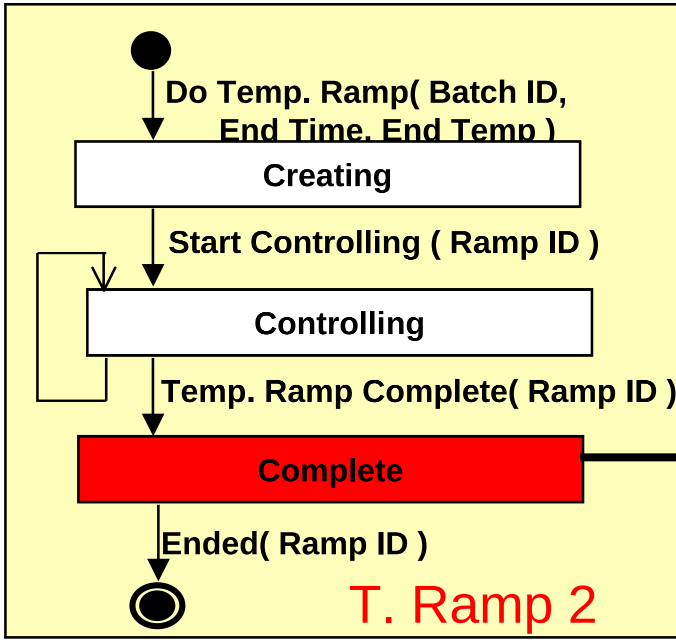
All instances execute concurrently.

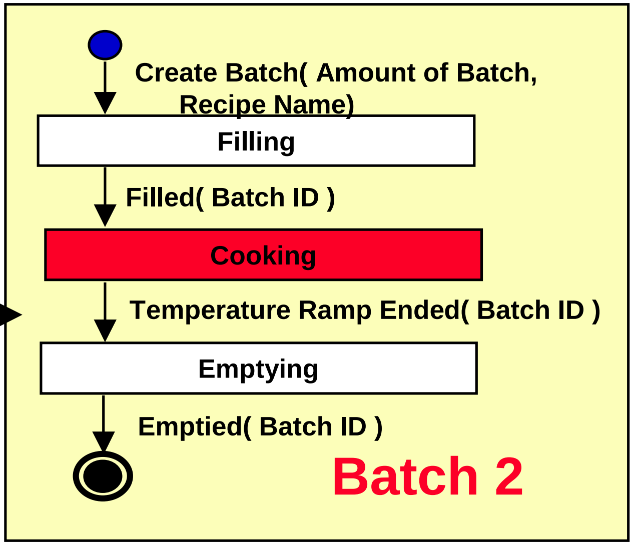




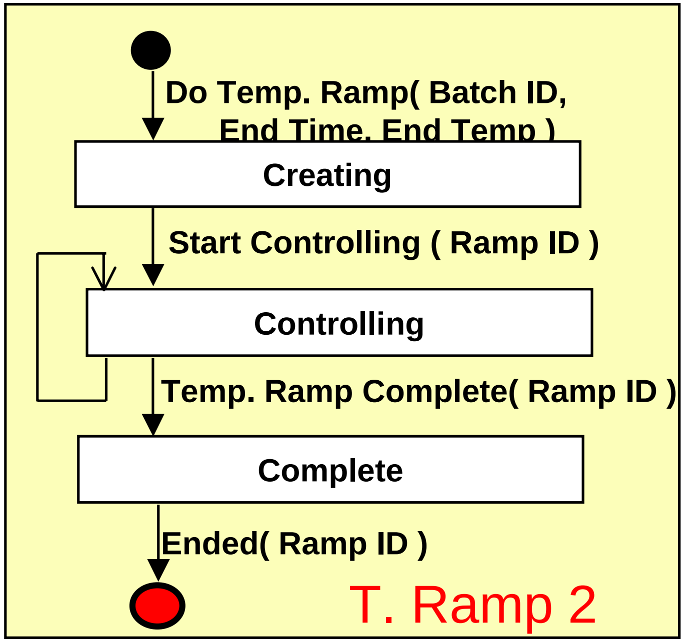


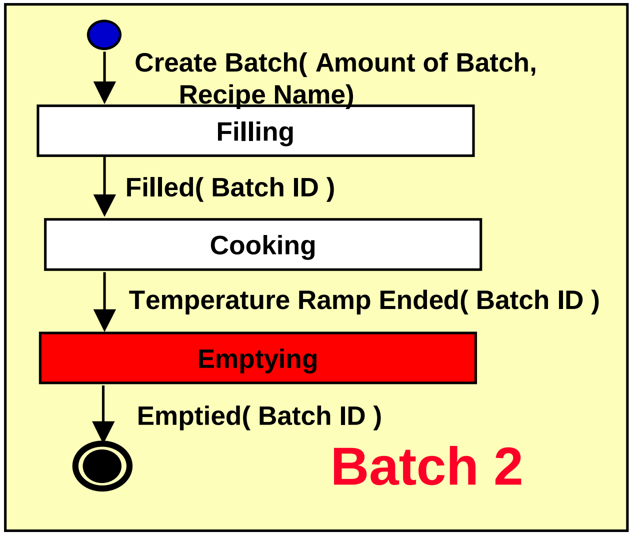
Communication





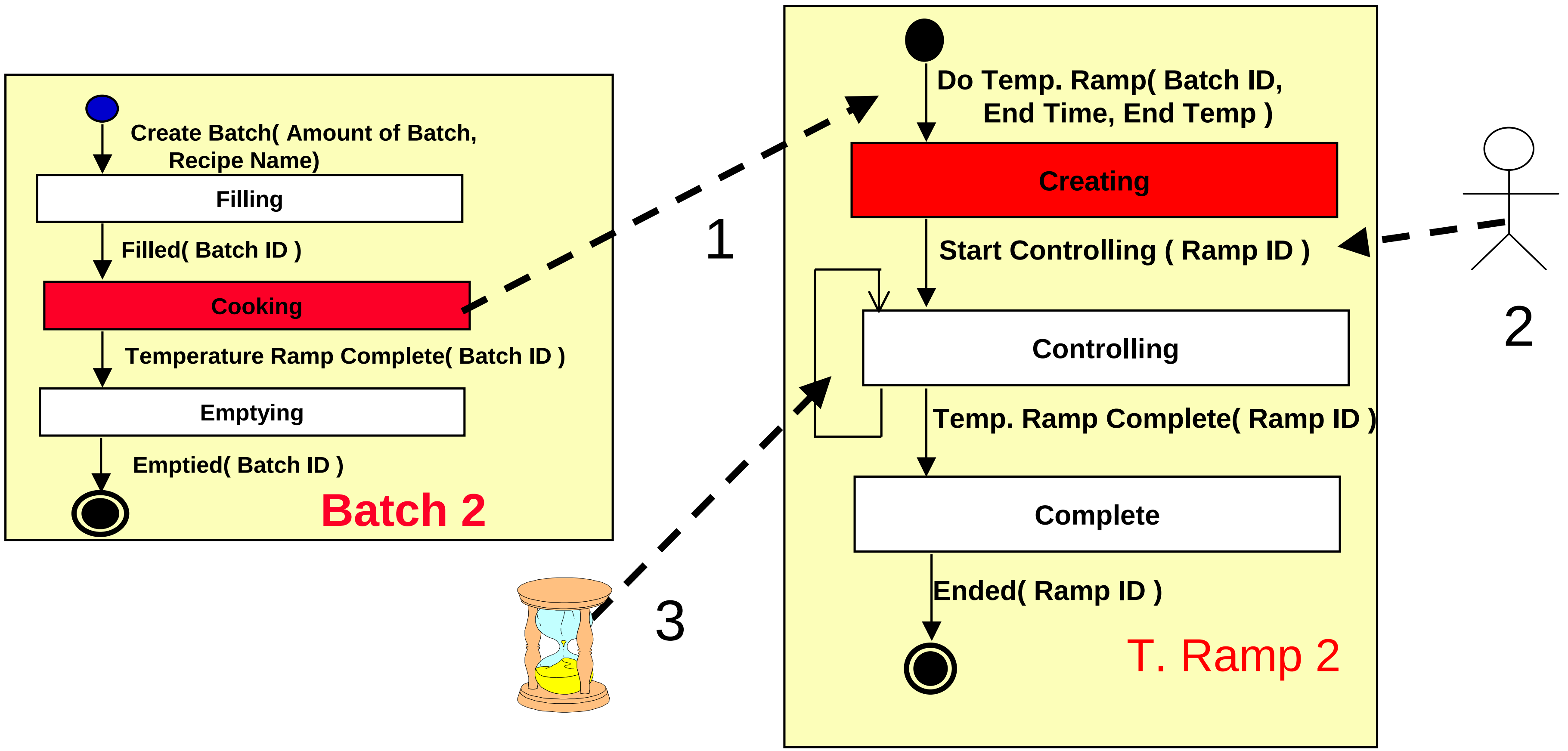
Before and after.





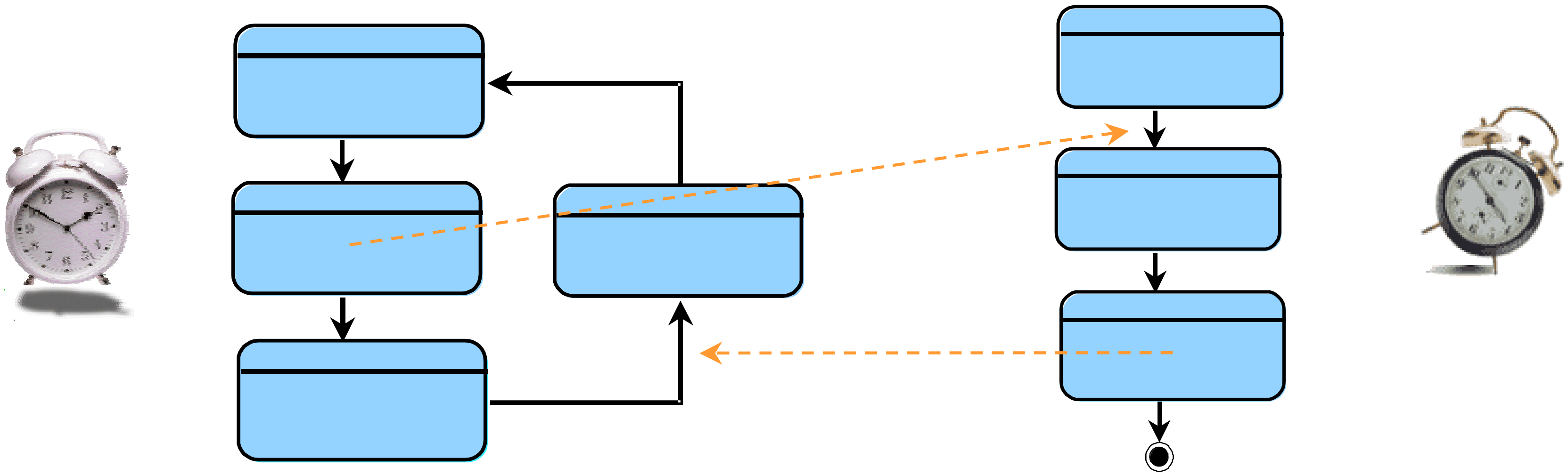
Verification

The model can be interpreted.



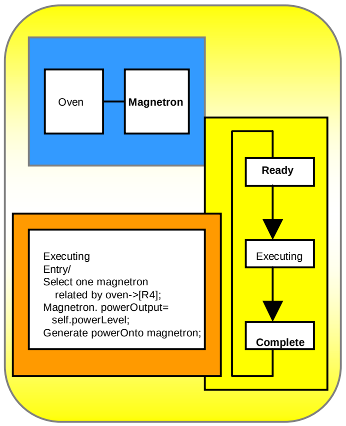
How does time work….?

- There is no global clock
- Time is relative to each observer



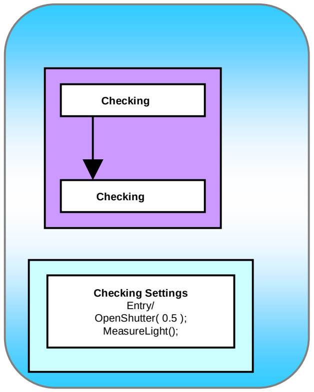
Separation

Application



Design can be split between:

Subject matter experts who understand the application, and



Architecture

Experts who understand the *application-independent* architecture

Each evolves at its own pace.

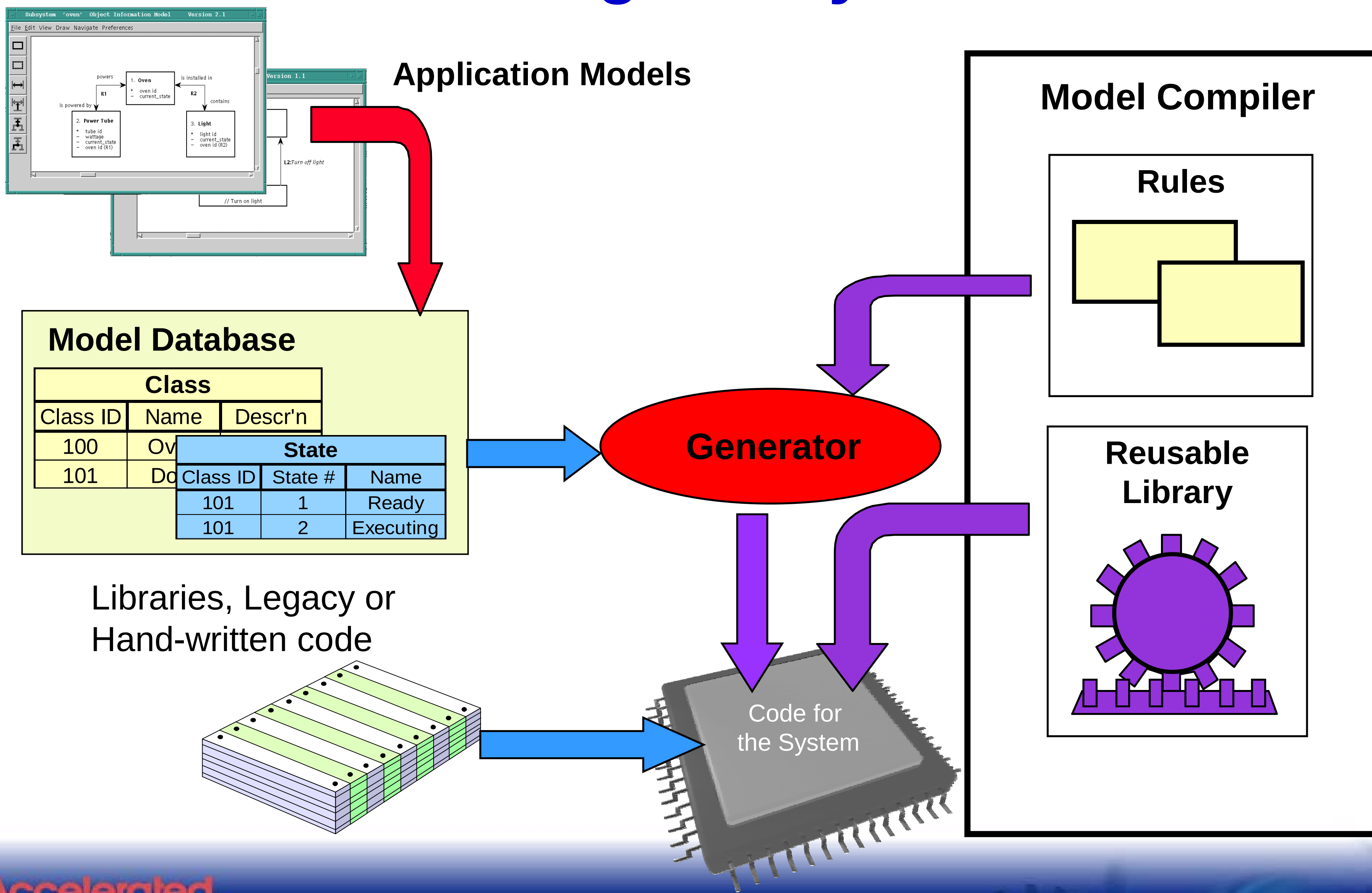
Mappings

Mapping by hand:

- Is error prone
- Not repeatable
- Not scalable
- Not predictable
- Leads to integration problems

The knowledge **The software**

Building the System



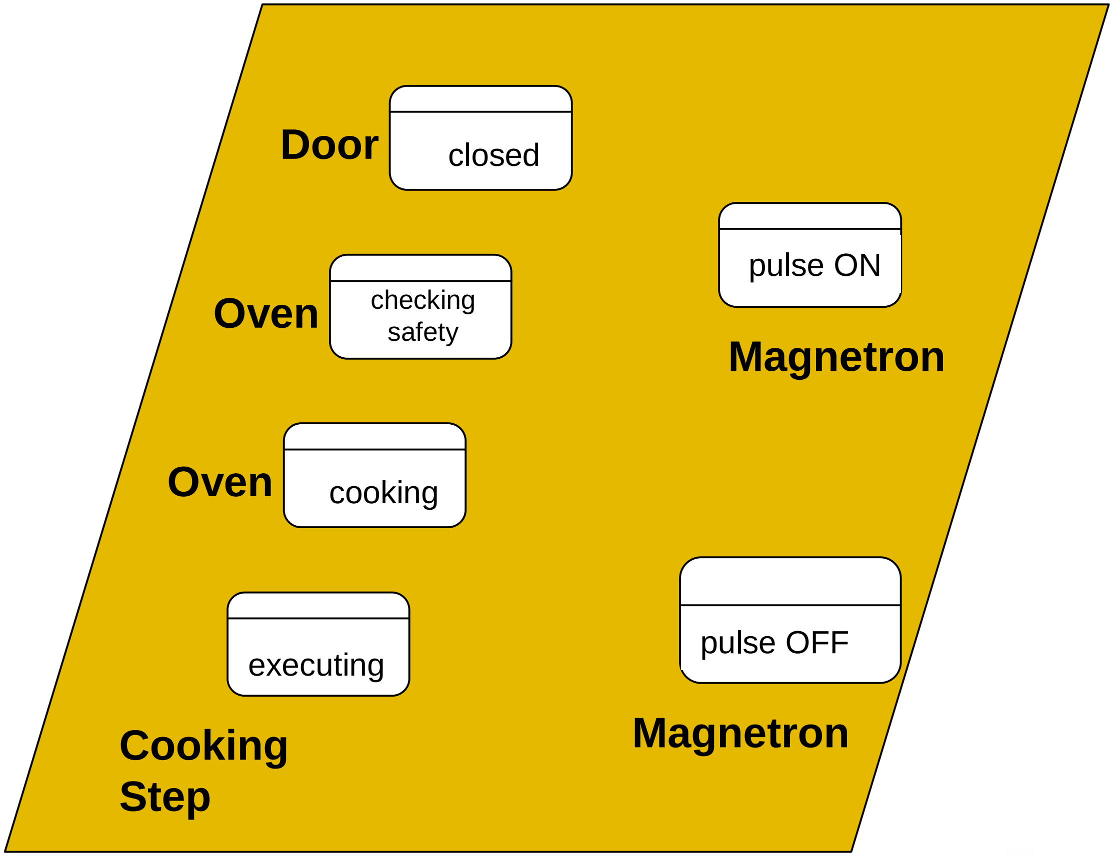
Concurrency in the generated code

- An application is inherently concurrent except insofar that it is purposefully sequenced
- xtUML captures necessary sequentialization in the model.
- Model compilers translate the concurrent specification onto an implementation of concurrency.
- What should the structure of that implementation be?

Activities

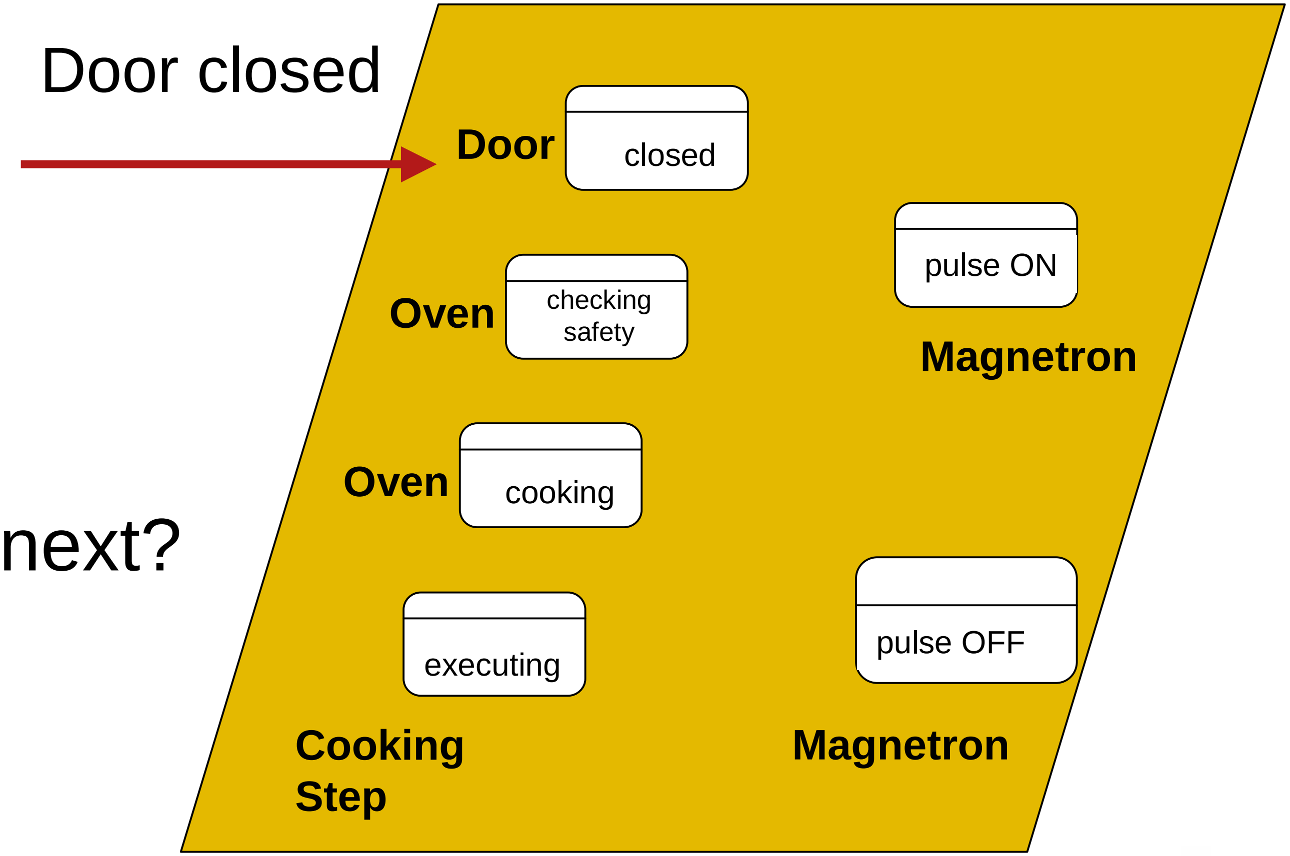
The basic unit of allocation is the activity.

Here, we allocate all activities to a single task.



Activities

Each activity is caused to execute on receipt of a signal.



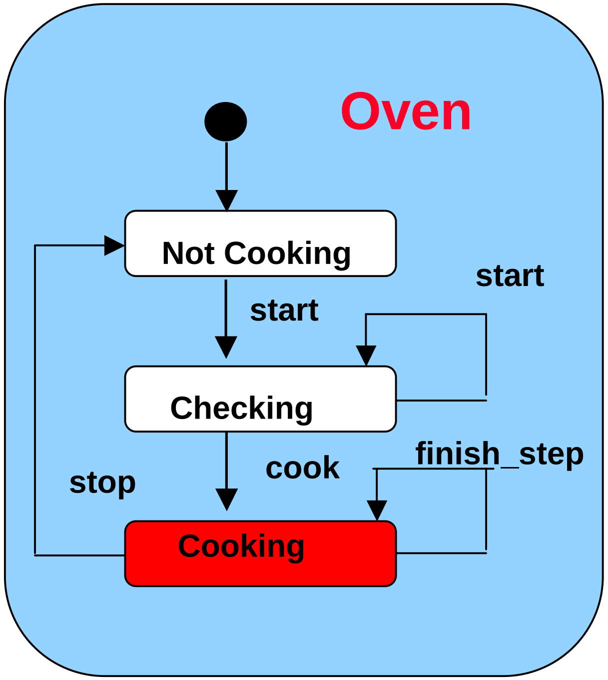
What happens next?

Dequeuing Signals

When a signal is received, we:

- Determine what to do next based on the name of the signal and the current state
- Do it

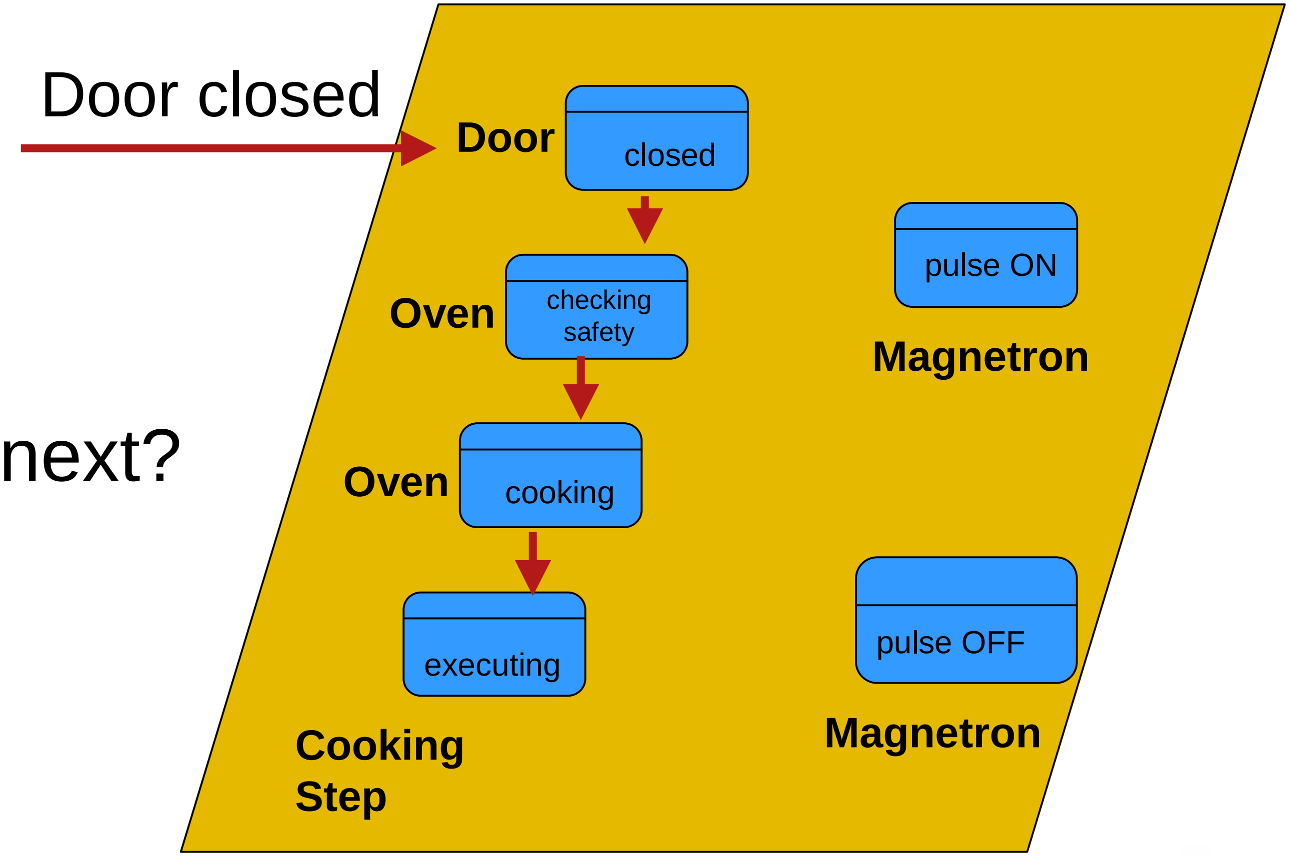
Signal / State	start	cook	step_done	stop
Not Cooking	**Checking**	**CANT HAPPEN**	**CANT HAPPEN**	**IGNORED**
Checking	**Checking**	**Cooking**	**CANT HAPPEN**	**IGNORED**
Cooking	**CANT HAPPEN**	**CANT HAPPEN**	**Cooking**	**Not Cooking**



Activities

An activity may send a signal.

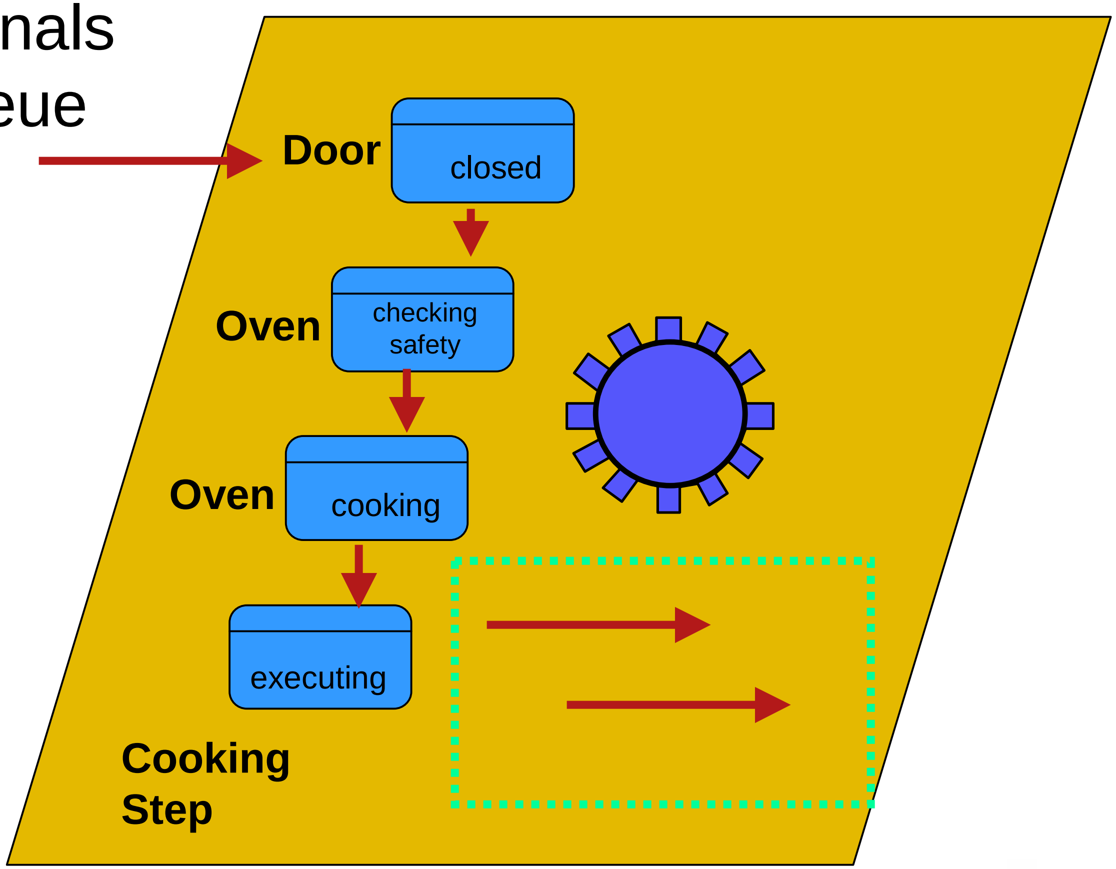
What happens next?



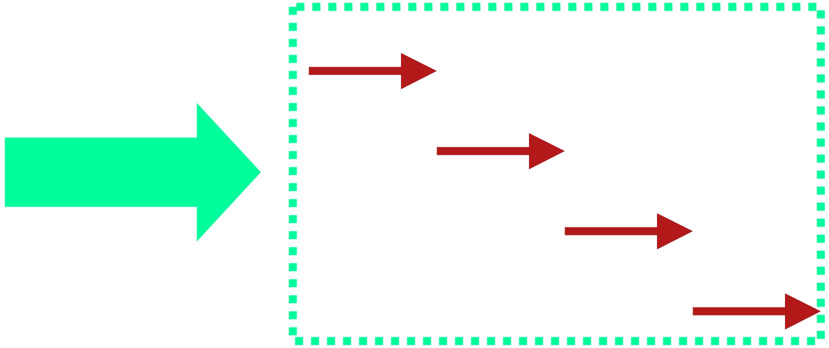
Communication

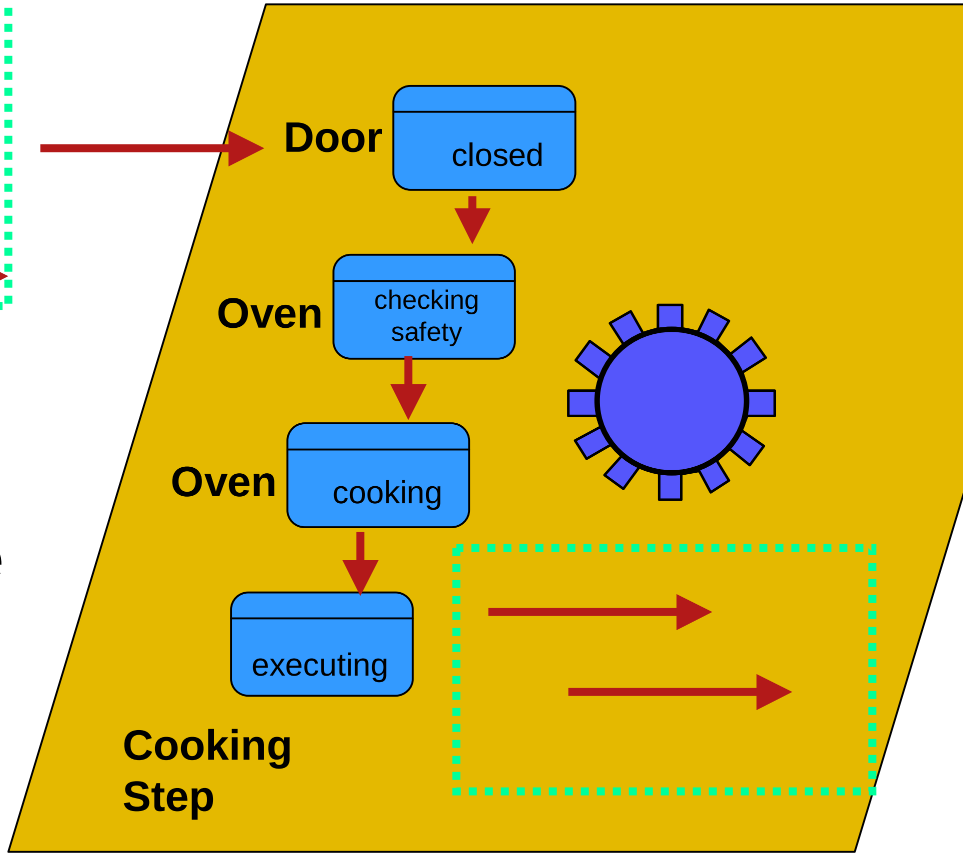
If actions generate signals to other objects, we:

- Store the new signals on an internal queue
- Pick them off one by one
- When the queue is empty, wait for another signal



External Communication

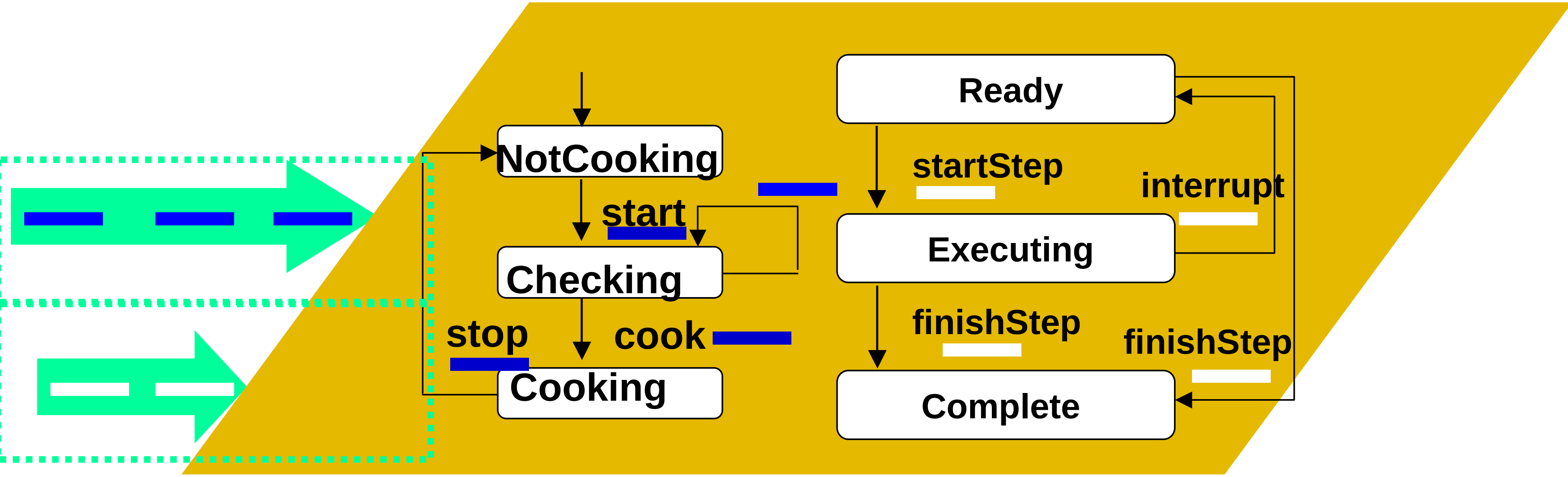




- Each task receives signals and queues them
- Each task executes all the processing associated with the receipt of a single signal
- Internal and external signals are the same

Priority in a Task

Events can be assigned a priority within a single task.



Single Tasking Concurrency

Concurrency is simulated, but:

- The “granularity of concurrency” is an activity
- An activity must wait for another to finish
- The waiting activity may be higher priority
- => Priority inversion

Multi-Tasking Concurrency

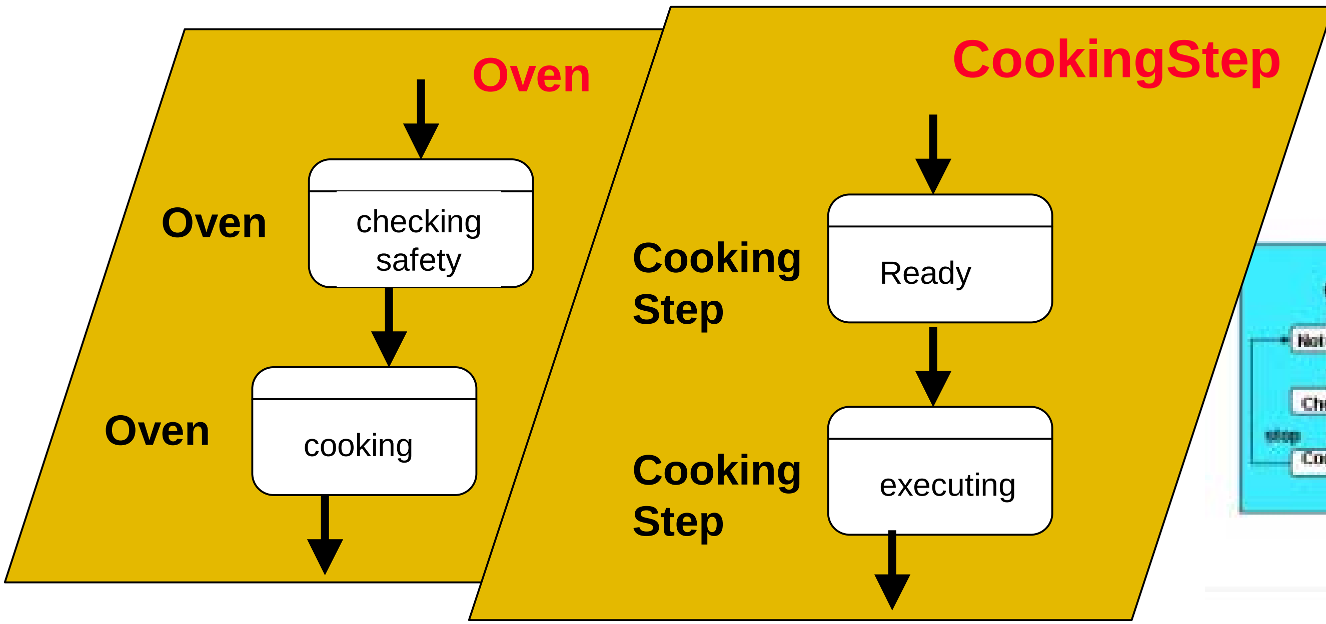
We can generate *n* tasks based on:

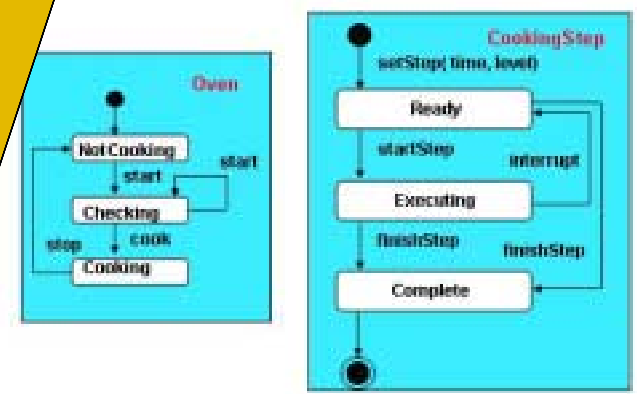
- Events
- Activities
- Classes
- Instances
- Threads of control
- Actions (even)

Single task ⟷ **One task per instance**

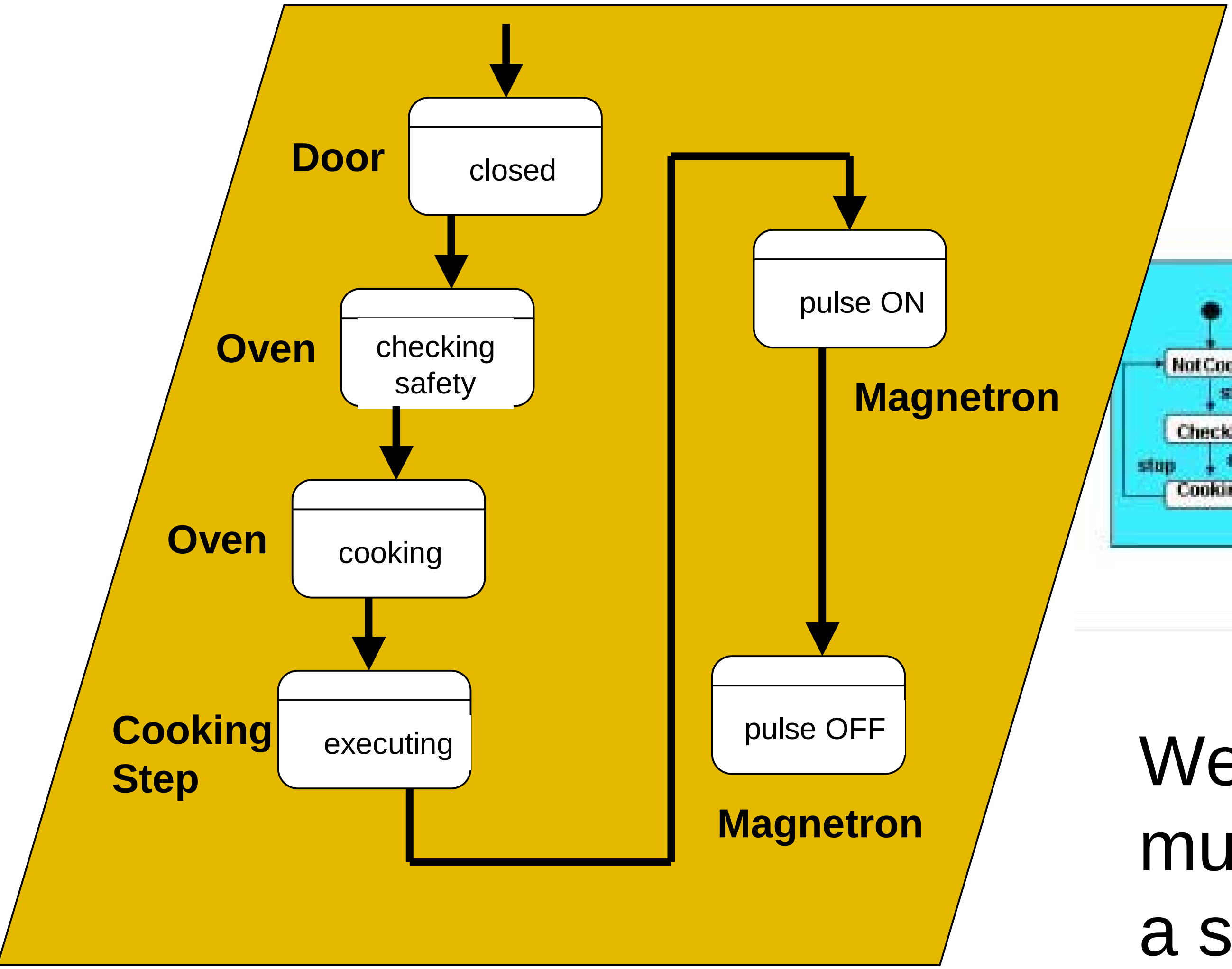
A Direct Mapping

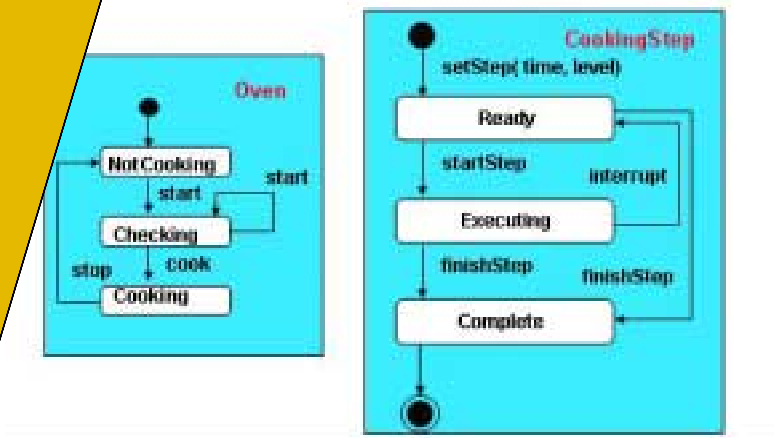
We may map each class to a separate task.





Map Multiple Classes

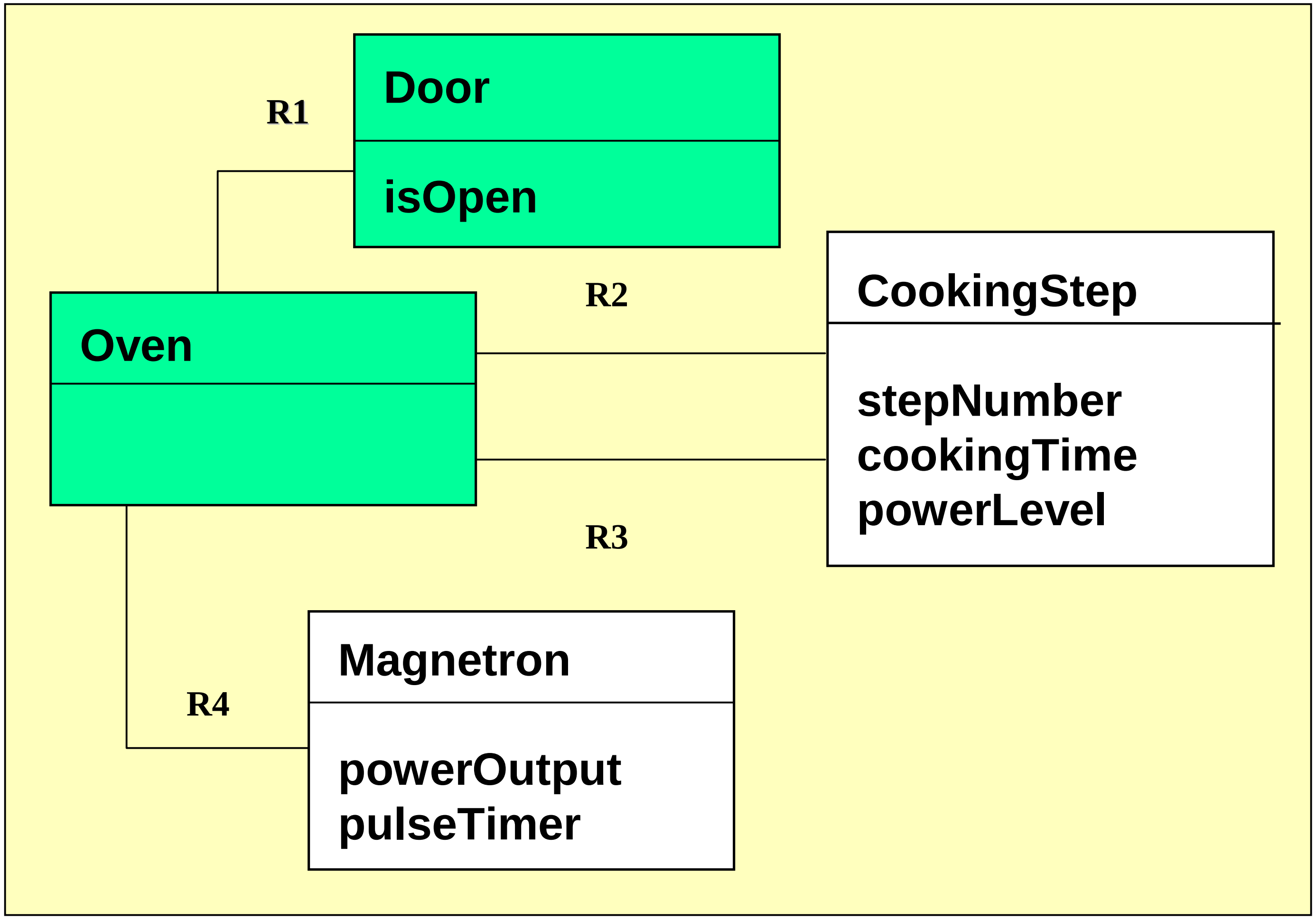




We may map multiple classes to a single task.

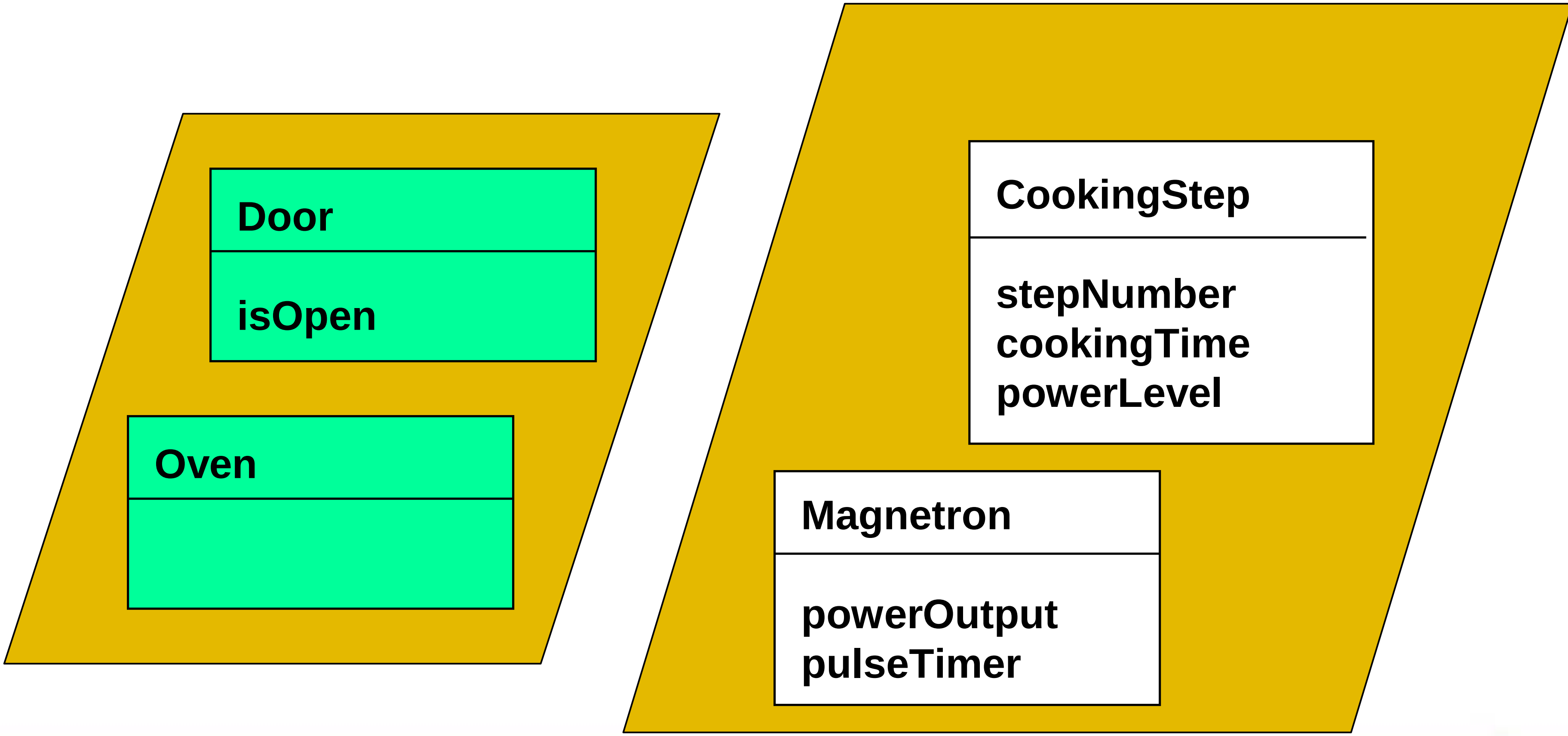
Multi-Tasking Concurrency

In this example we choose classes.



Multi-Tasking Concurrency

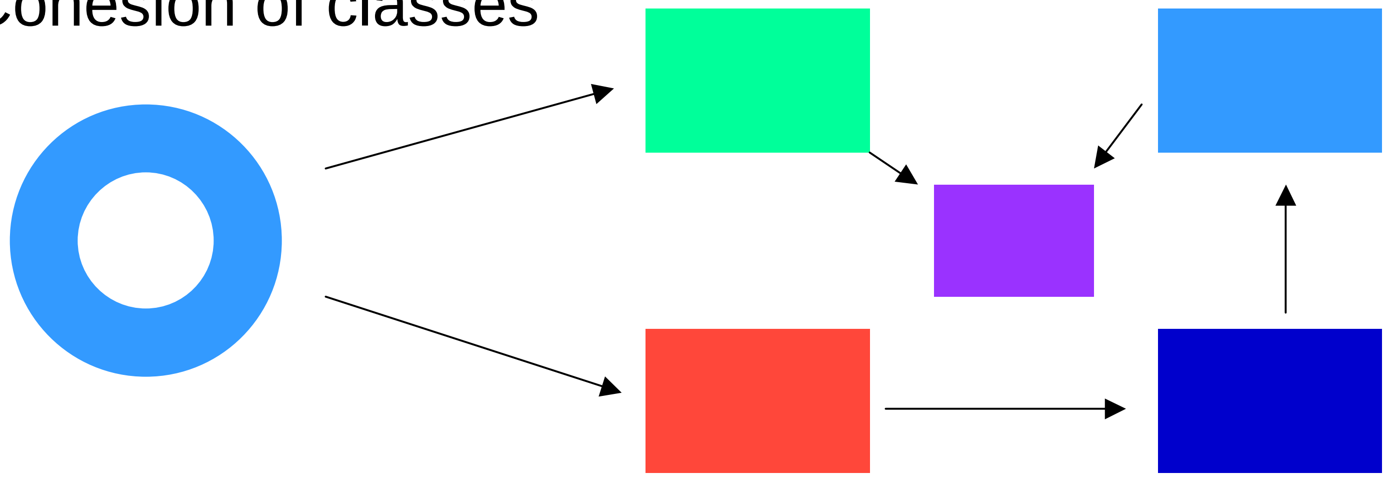
In this example we choose classes.



You Choose

You choose how to use concurrency based on:

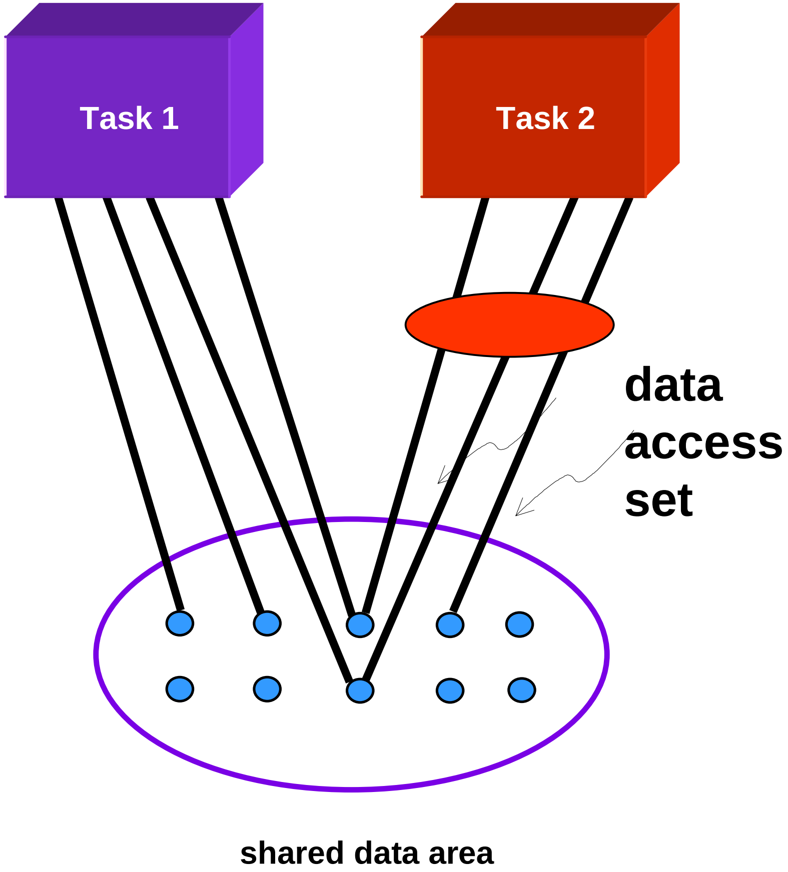
- I/O device latency
- Priority (and the avoidance of inversion)
- Coupling between classes (and especially data traffic between classes allocated to tasks)
- Cohesion of classes



You Choose Carefully

Concurrent tasks:

- Incur context switching overhead
- Risk data-access-set conflict



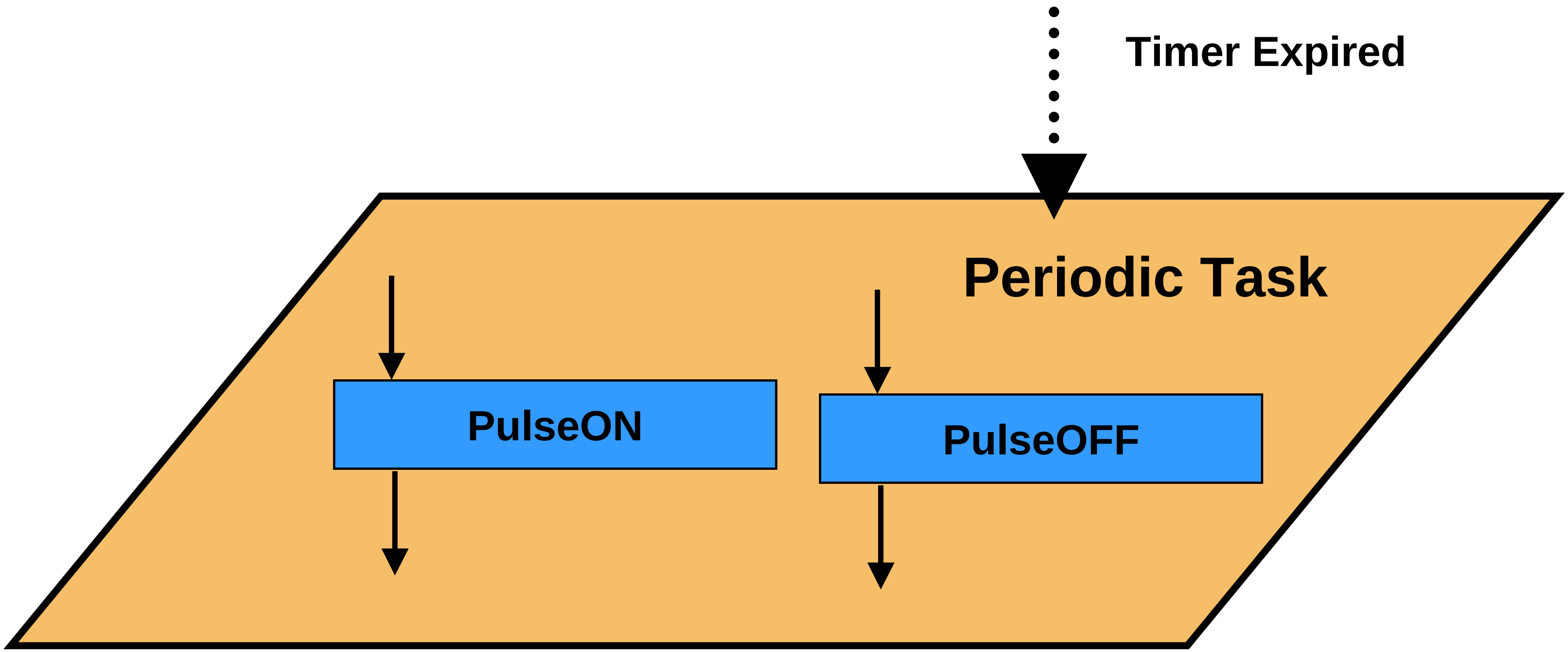
Where Have All The Cycles Gone?

The architectural overhead involved in making a transition every *x* microseconds could become intolerable if *x* is very small.

Is there a way to reduce architectural overhead in this example?

Where Have All The Cycles Gone?

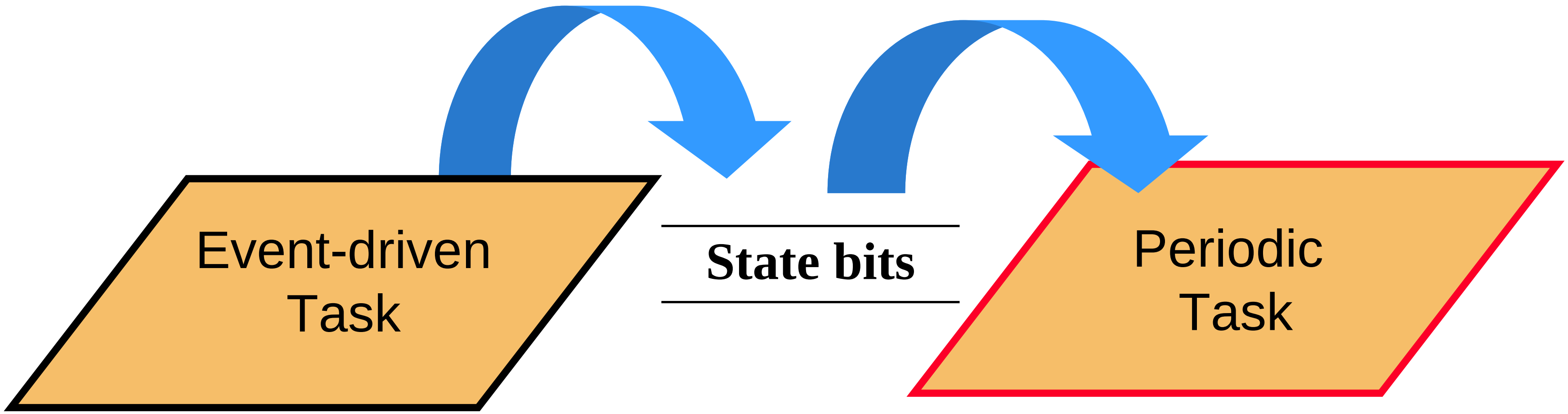
We could use a single task that executes periodically.



Description of Architecture

How to indicate transition to/from the periodic states?

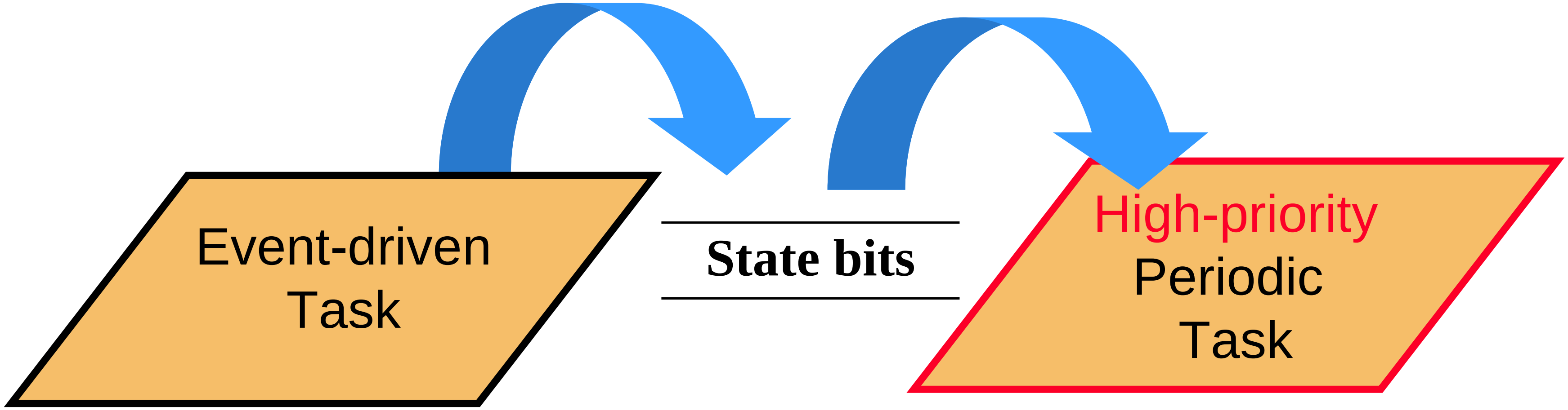
- use two bits per instance
  (one bit for PulseON/PulseOFF,
  the other for whether we are executing periodically)



Description of Architecture

The periodic task has to be able to execute when "it's time."

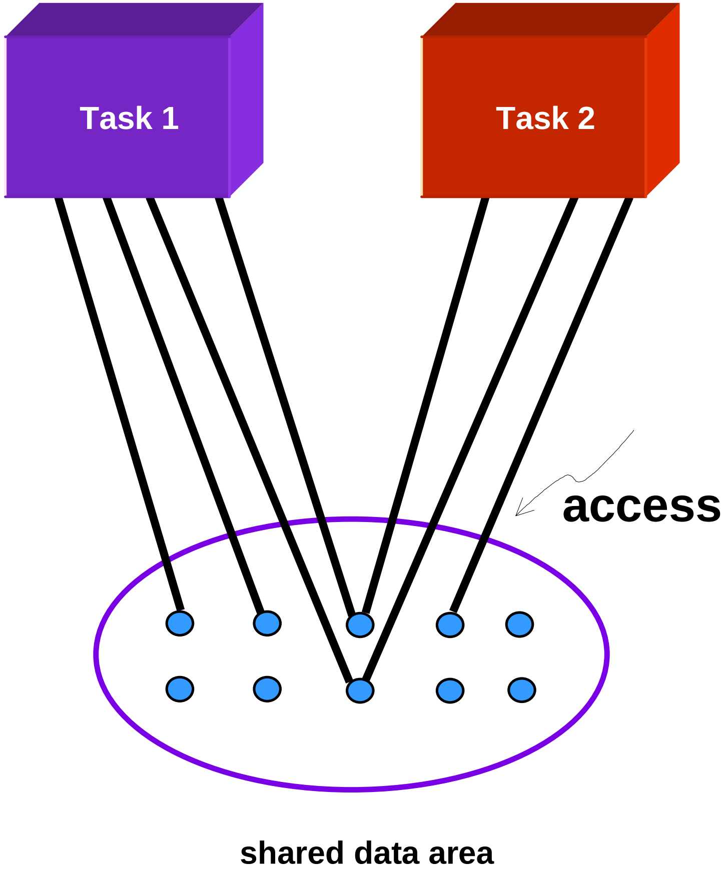
Give the periodic task higher priority



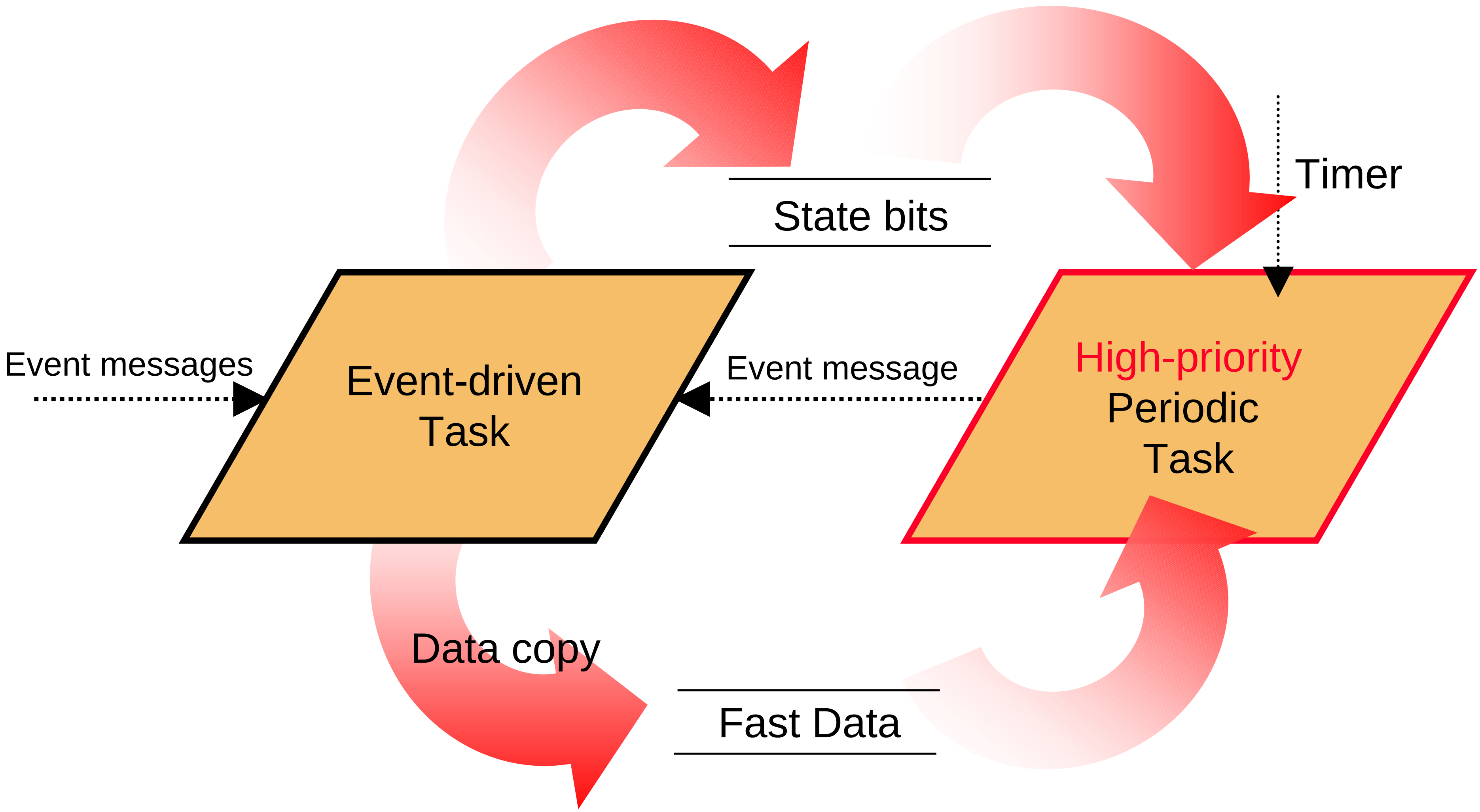
Description of Architecture

Priority tasking implies:

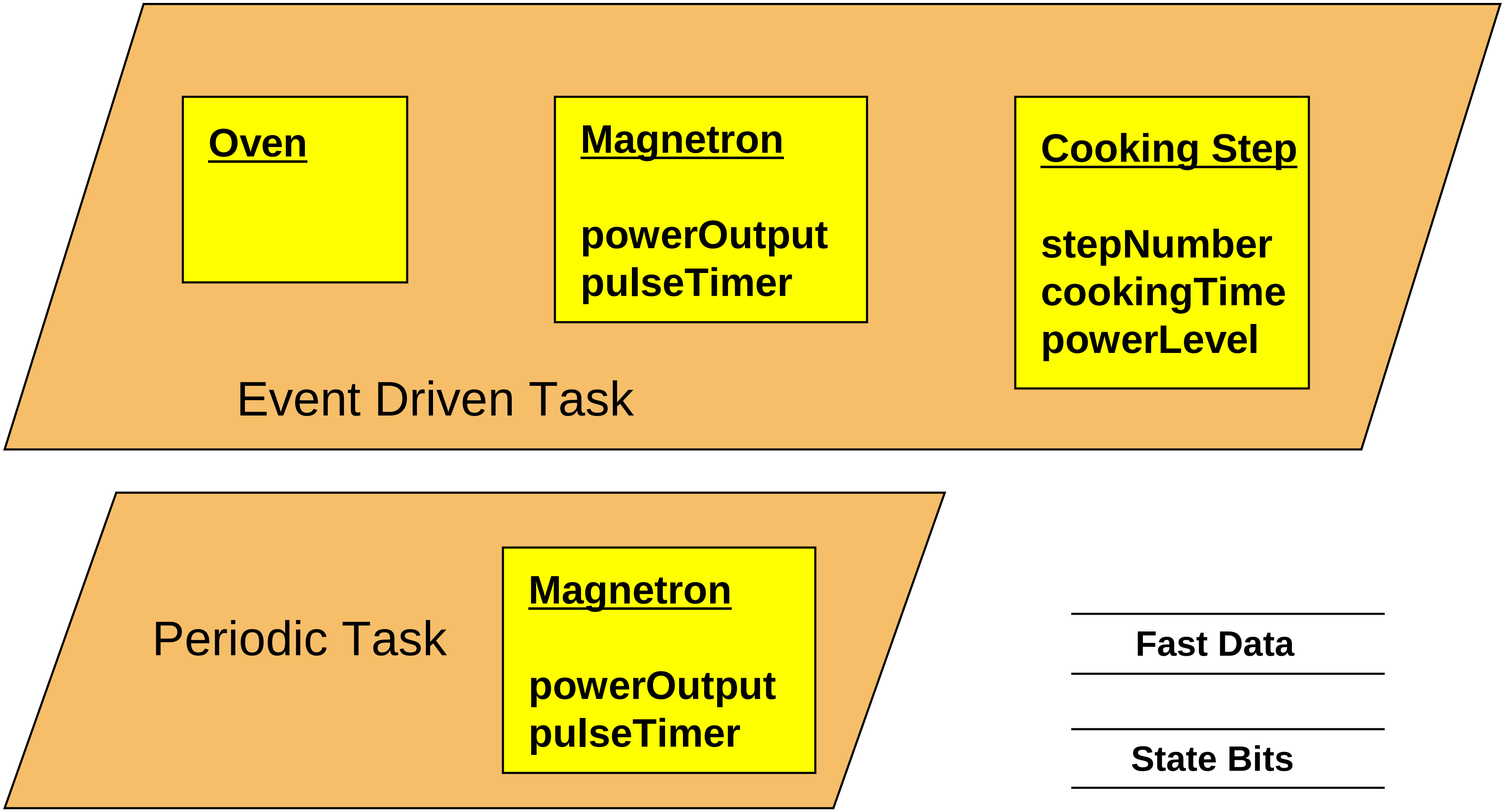
- data inconsistency could be caused if a higher priority task interrupts
- duplicate data needed for the control loop, and copy it over by the periodic task when required



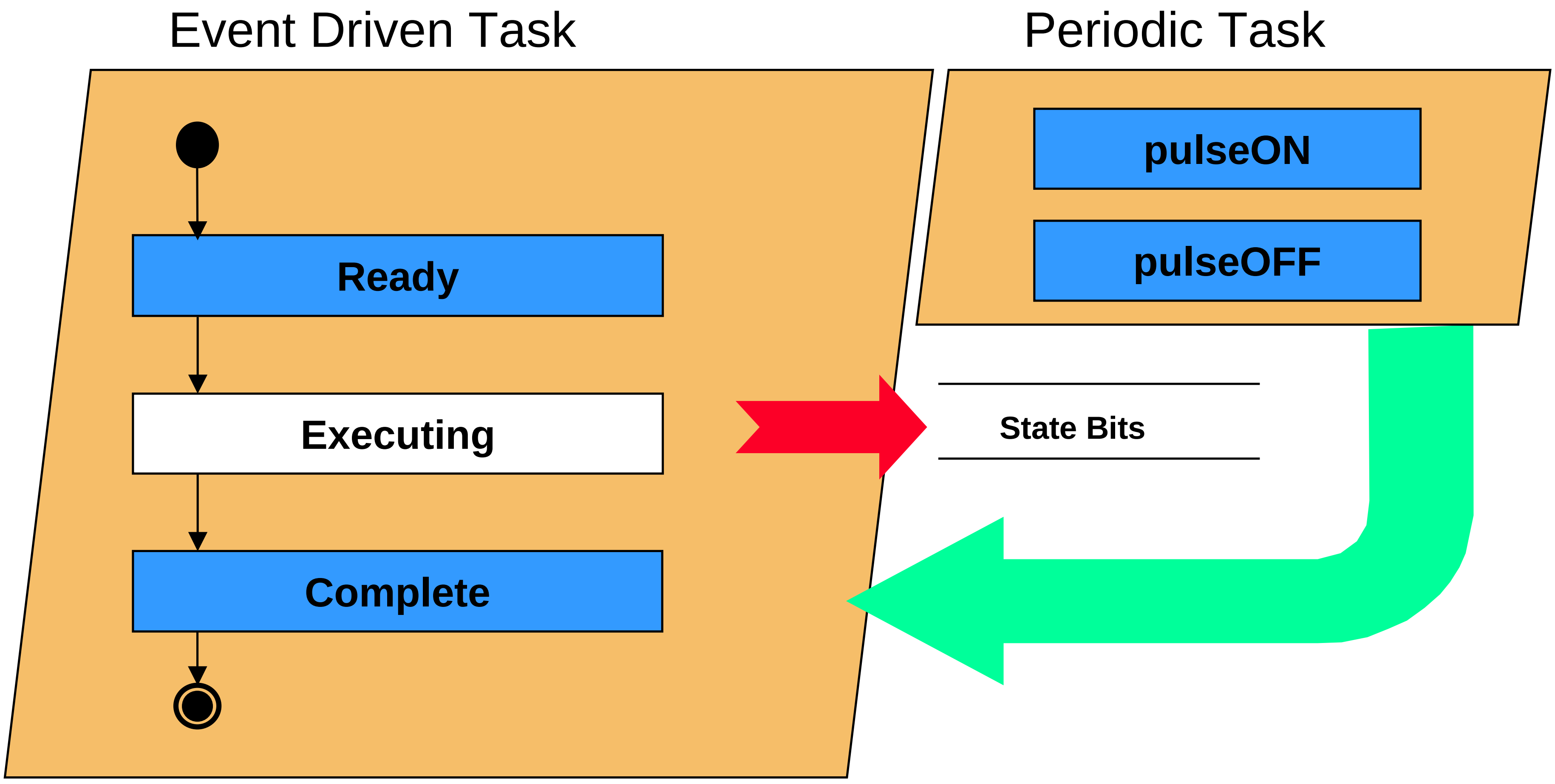
Sketch of Architecture



Application Mapping



Application Mapping



Extended Properties

State
Class ID {R14} State Number Name isFinal *isPeriodic*

To make certain distinctions, we need to mark elements of the meta-model.

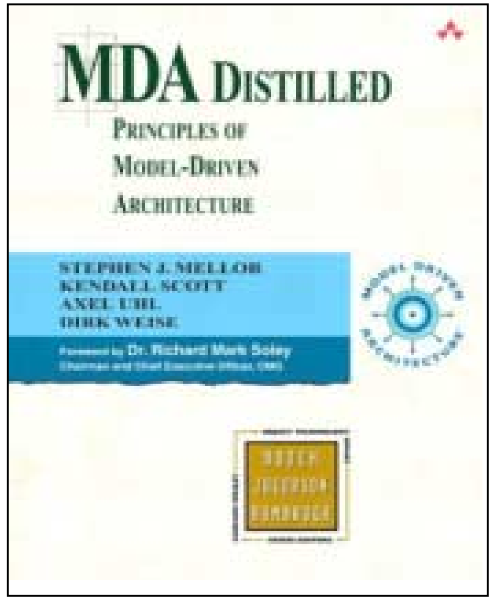
.function addPeriodicStateAction

…

StateBits[insNumber].activateActions();

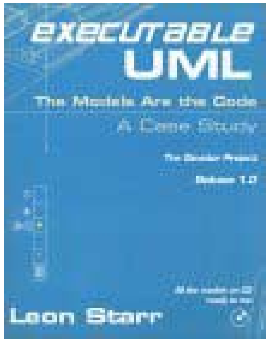
Want to learn more?

Executable UML: A Foundation for Model-Driven Architecture, Stephen J Mellor, Marc Balcer

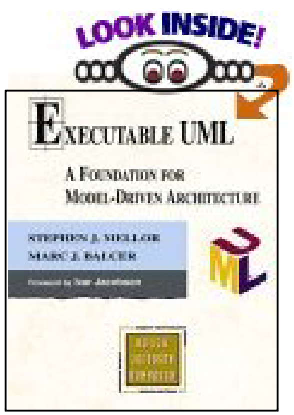


A description of how this technology fits in to Model-Driven Architecture

***Executable UML: A Case Study*,** Leon Starr



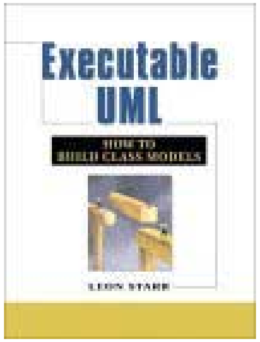
A complete set of models on CD that you can execute.



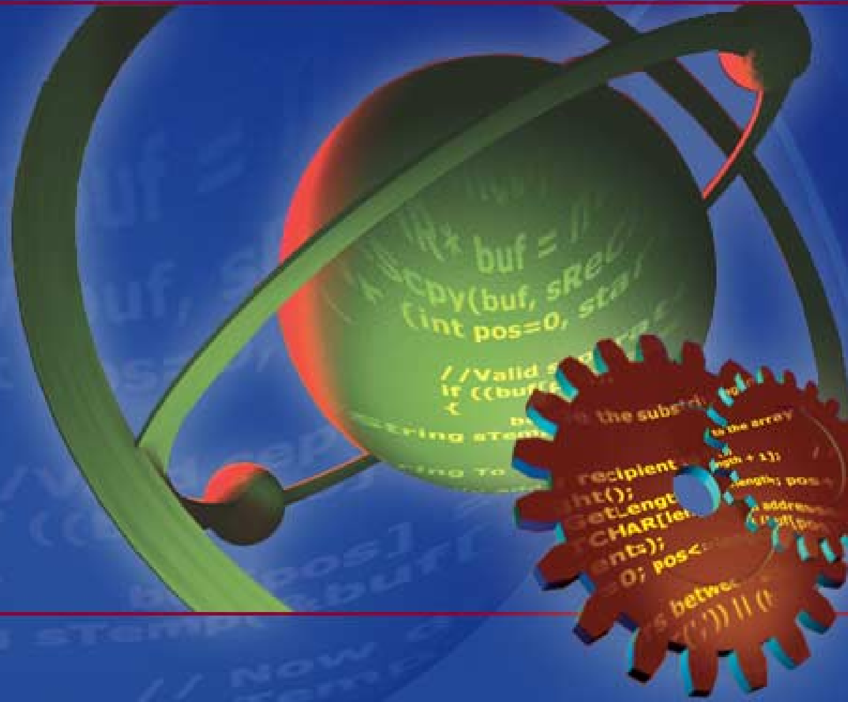
Comprehensive language introduction and reference

***Executable UML: How to Build Class Models*,** Leon Starr

Practical guide for model developers.



Questions?



www.acceleratedtechnology.com

Accelerated Technology®
A Mentor Graphics Division