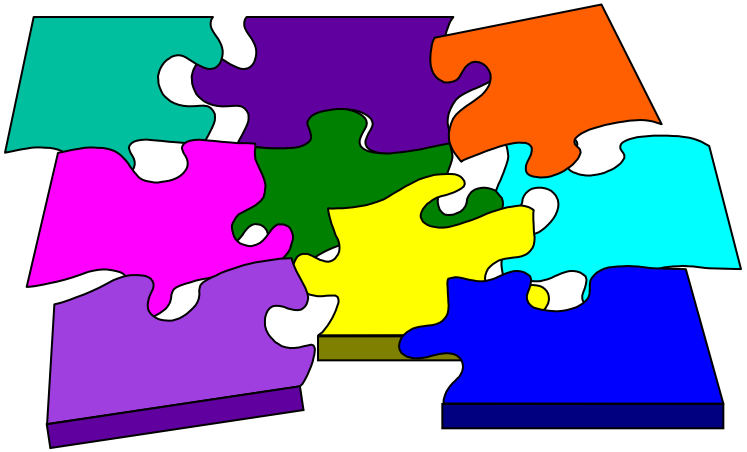
Coding with UML

Stephen J. Mellor

What's the problem?

1



A simple embedded system



- Set the time,
- Close the door,
- Press ‘cook’
- It’s that easy!

But…..

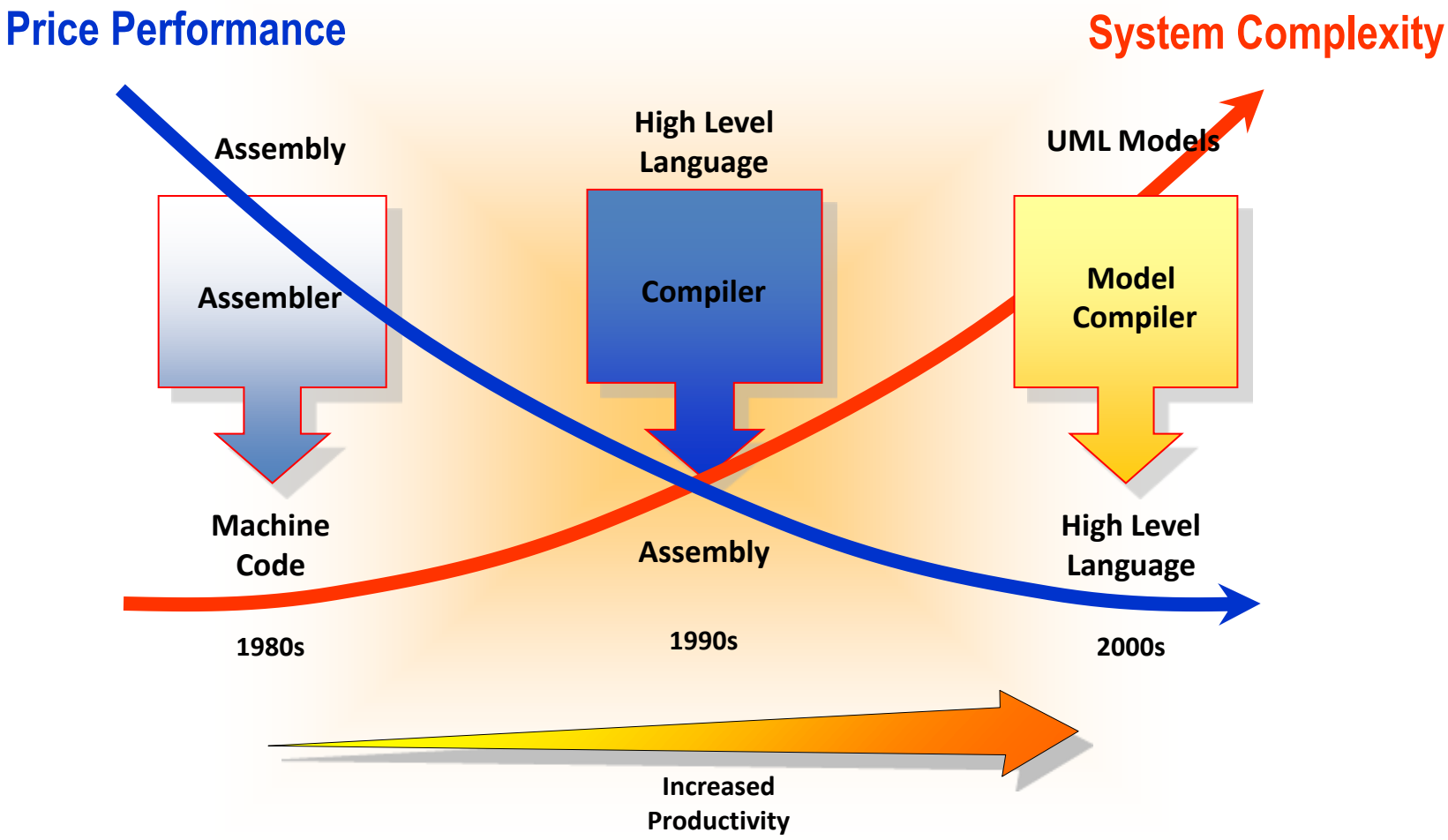
- For safety, the door *must* be closed
- If the door is opened, cooking *must* stop.
- There are some details about timers
- And power levels……
- And…. And…. And….

Complex Systems

- We want to be able to:
  - Develop concepts rapidly
  - Test ideas with confidence
  - Reduce required code by identification of cross-cutting commonality
  - Understand performance before implementation
  - Adapt easily to hardware and platform changes
  - Maximize parallel development

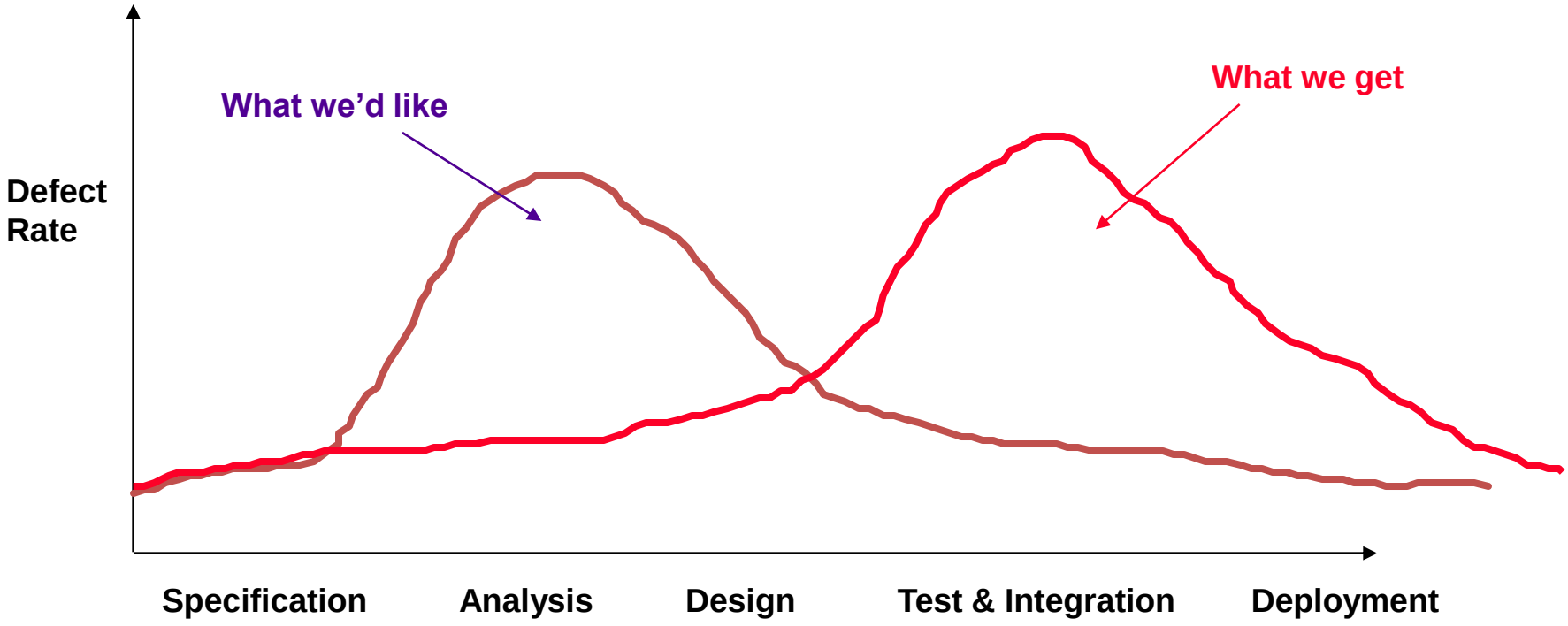


Abstraction



Testing

- Systems are difficult to test—and it occurs too late.
  - Wasted effort in constructing wrong code
  - Late discovery of problems and delayed delivery



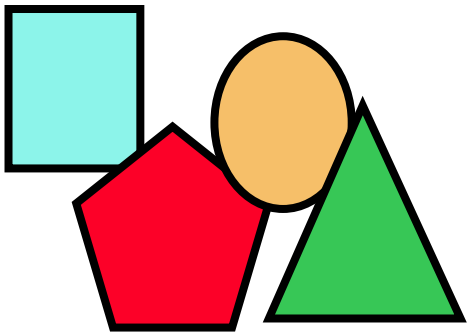
Skills

- Large designs require specialization of skills.



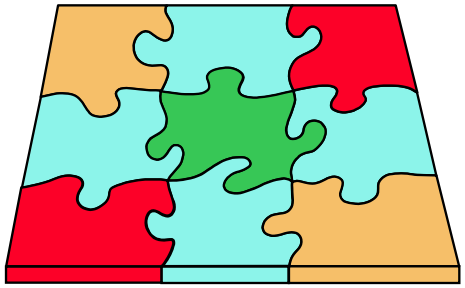
A solution is…

- 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





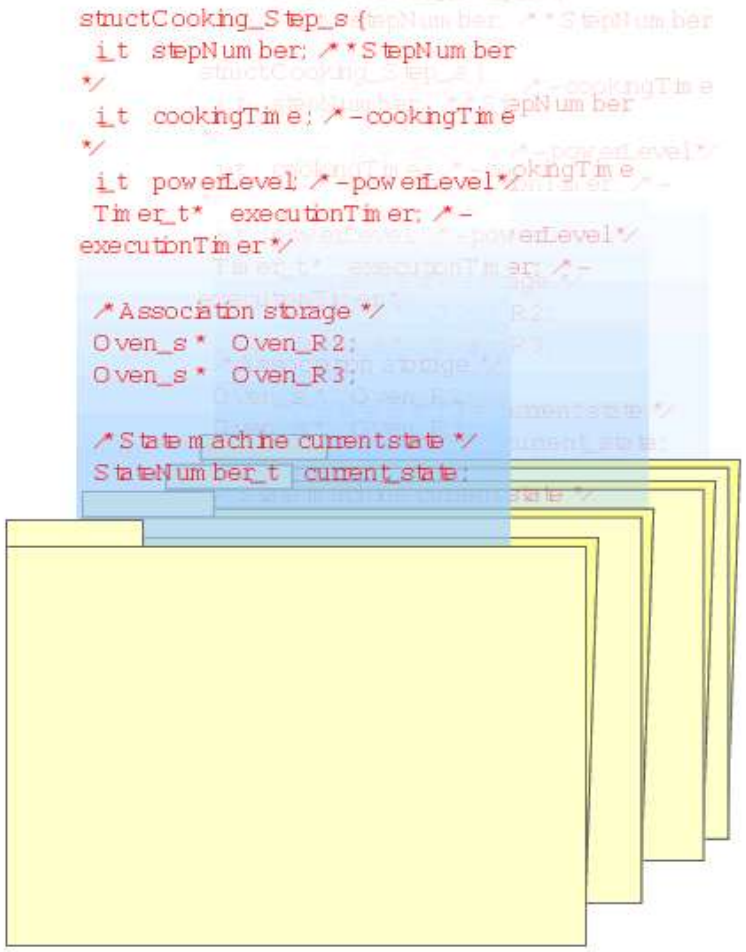
The software

Why?

- Why are concurrent specifications necessary?
  - Because our problems *are* concurrent
  - Concurrency problems exist in the problem as well as the implementation.
  - So we can establish priorities among tasks
  - So we can maximize precious system resources when many things can happen at once.

Coding Data in UML

2



Code for a Microwave Oven

```
struct Oven_s {
  UniqueID_t    OvenID;  /* * OvenID *

  /* Association storage */
  Door_s    * Door _R1;
  Cooking_Step_s    * Cooking_Step _R2
  Cooking_Step_s    * Cooking_Step_R3;
  Magnetron_s *    Magnetron_R4;

  /* State machine current state */
  StateNumber_t    current_state;
};
```



Another Piece of Code

```
struct Cooking_Step_s {
  i_t     stepNumber;  /* * StepNumber */
  i_t     cookingTime;  /* - cookingTime */
  i_t     powerLevel;  /* - powerLevel */
  Timer_t *     executionTimer;  /* - executionTimer
*/

  /* Association storage */
  Oven_s *     Oven_R2;
  Oven_s *     Oven_R3;

  /* State machine current state */
  StateNumber_t     current_state;
};
```



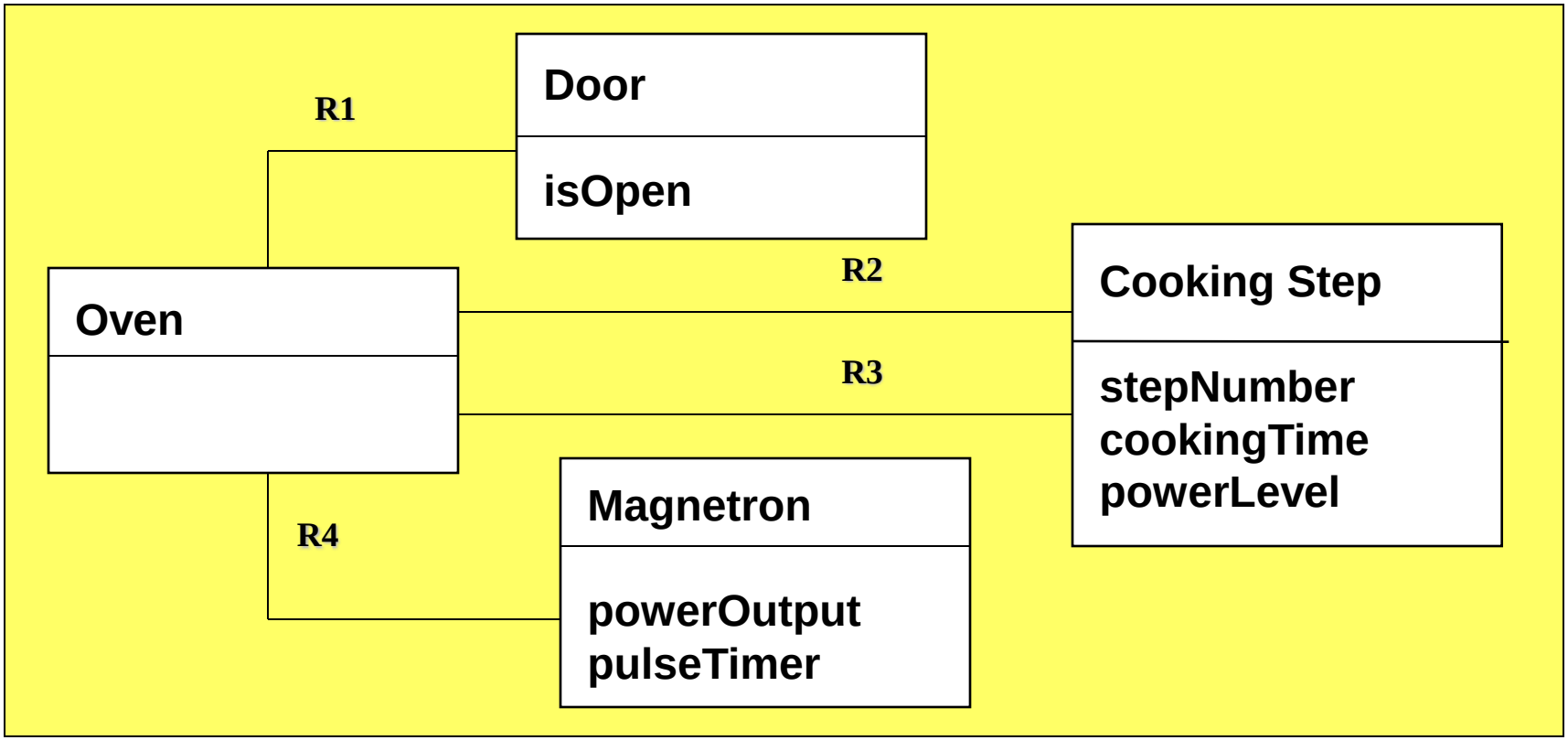
The Code Follows Rules

```
struct <class name>_s {
  <attr1 type>    <attr1 name>; /*  <attr1 description> */
  <attr2 type>    <attr2 name>; /*  <attr2 description> */
  <attr2 type>    <attr2 name>; /*  <attr2 description> */
       . . .
/* Association storage */
  <ref1 class name>_s *   < ref1 class name >_<assoc1 number>;
  <ref2 class name>_s *   < ref2 class name >_<assoc2 number>;
       . . .
 /* State machine current state */
  StateNumber_t    current_state;
};
```

Class Diagram

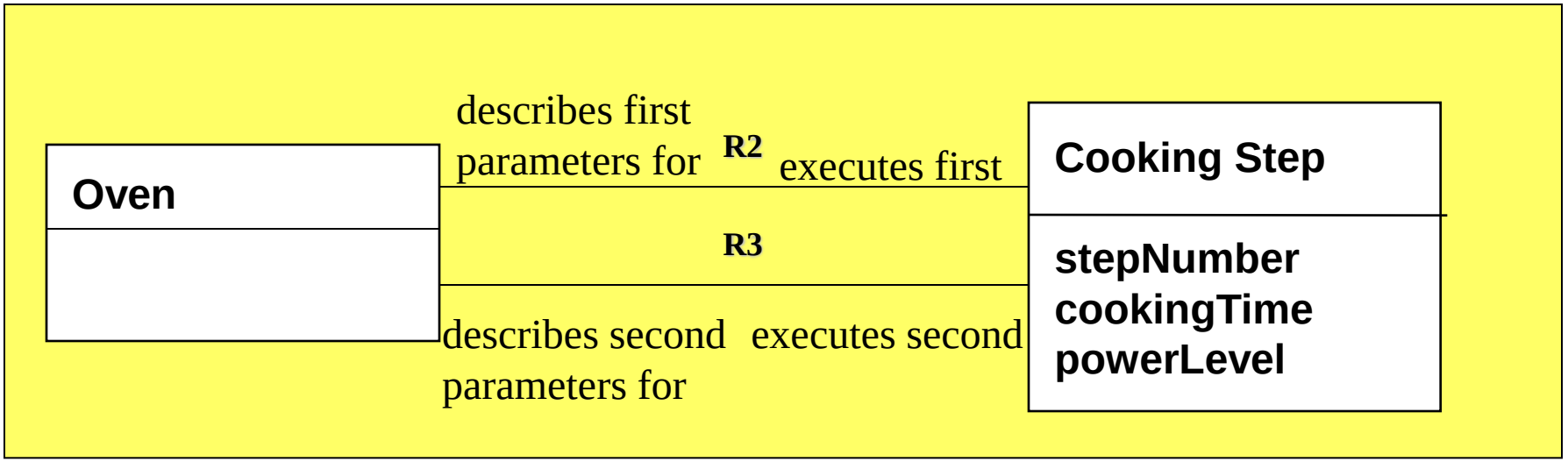
Abstract classes based on both:

- Data
- Behavior



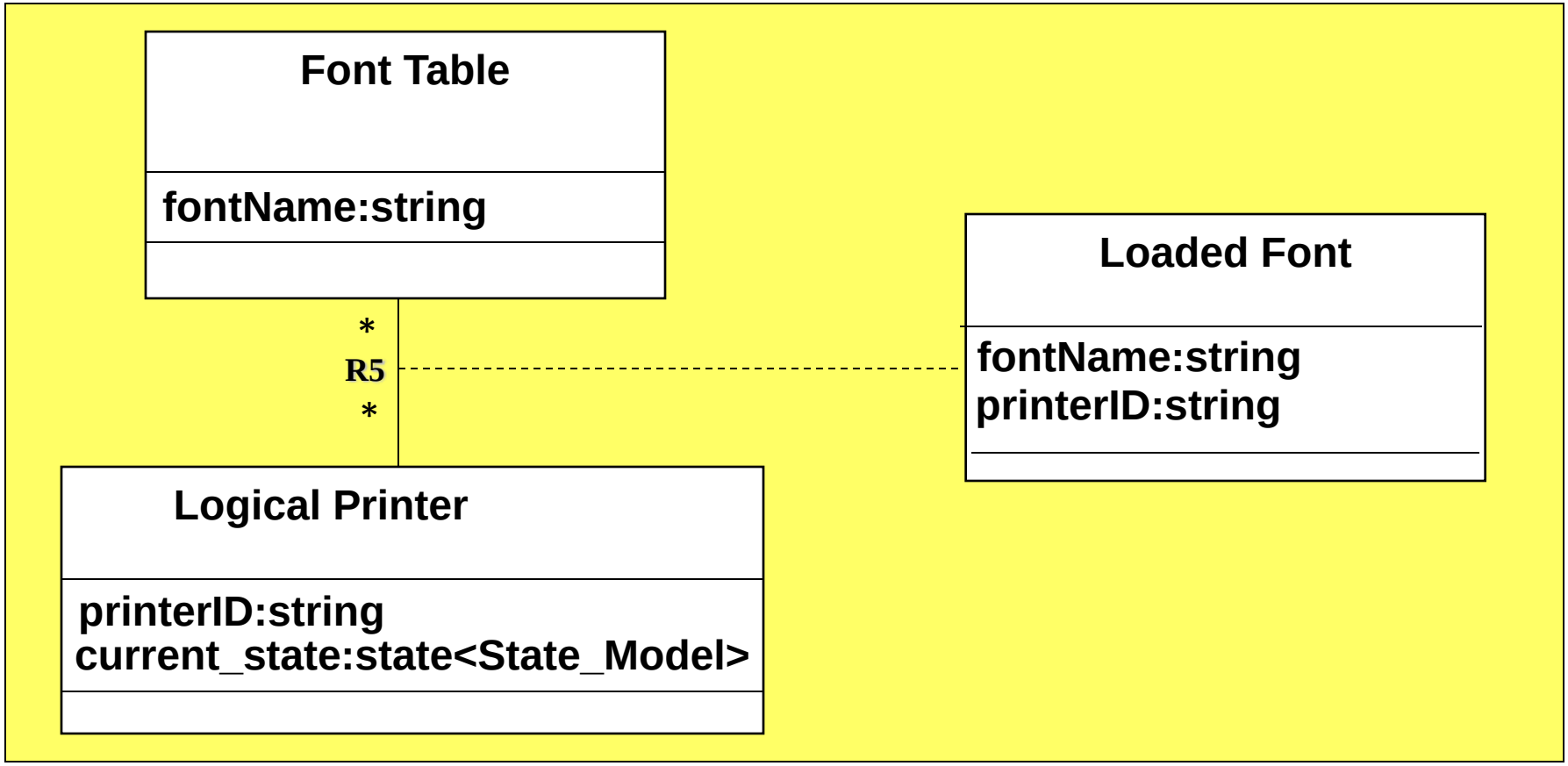
Associations

- Classes may be associated to capture relationships in the subject matter.
- We number the relationships to ensure uniqueness.



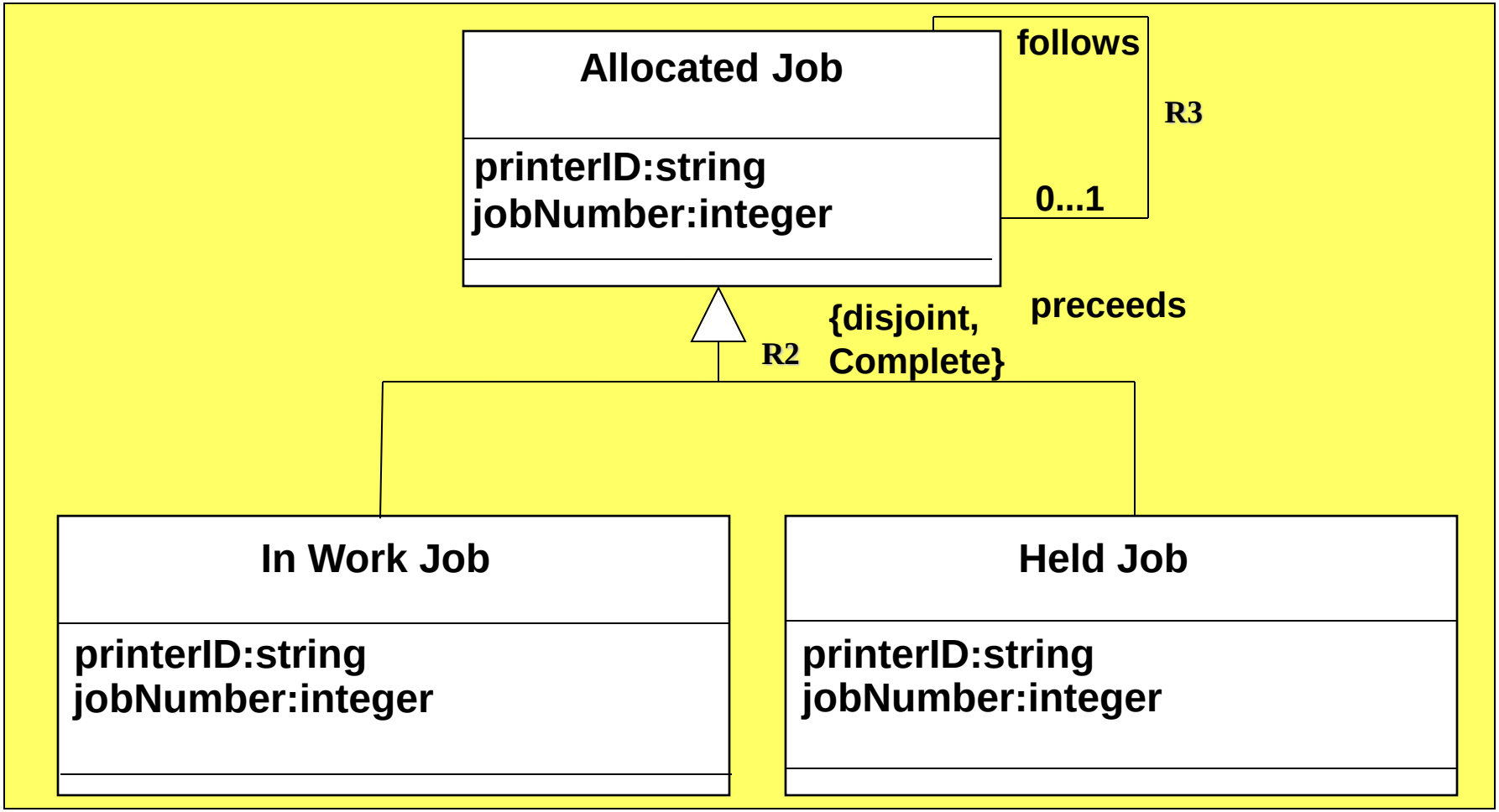
Association Classes

- Some associations give rise to other classes.



Generalization

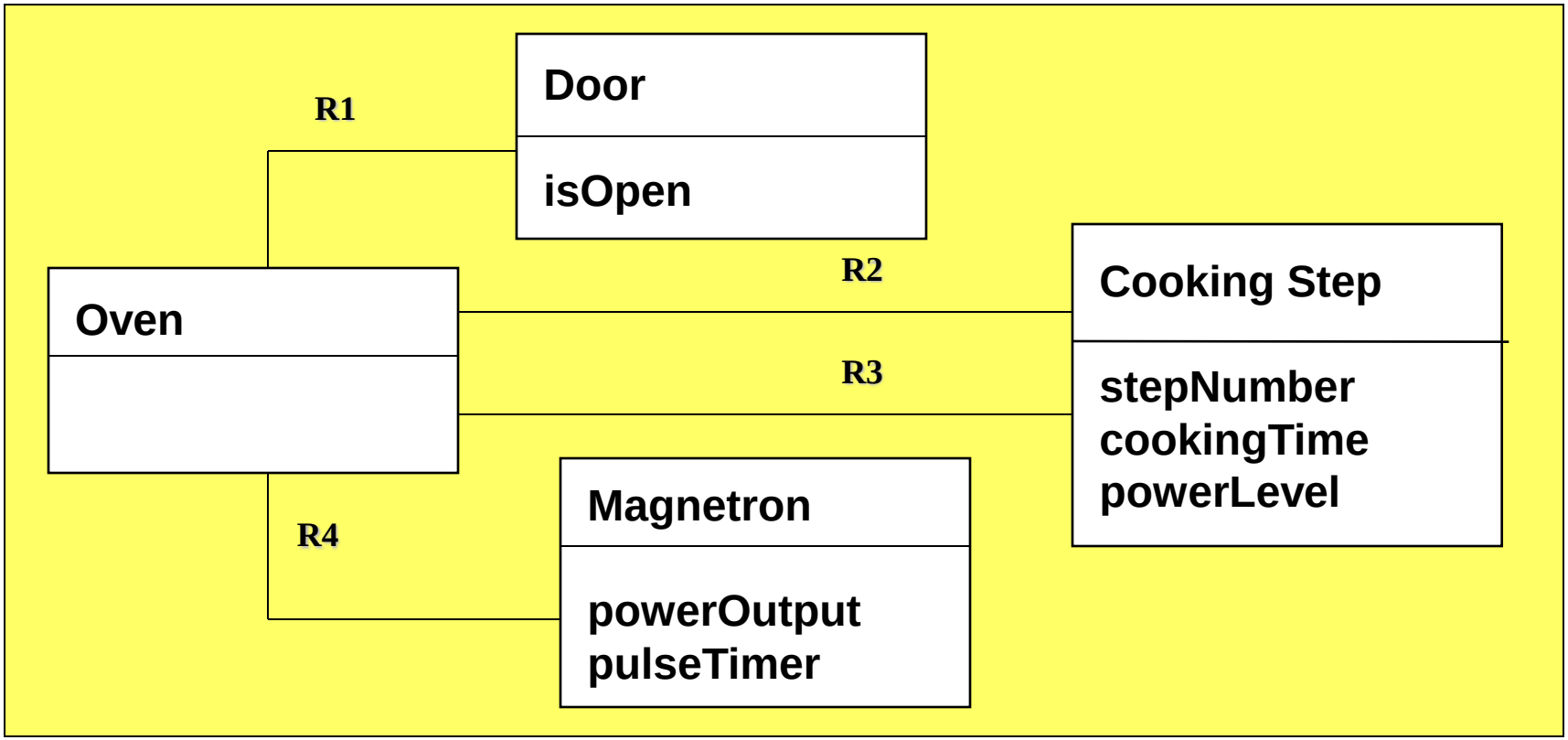
- One class may be a supertype of others.



Class Diagram

Abstract classes based on both:

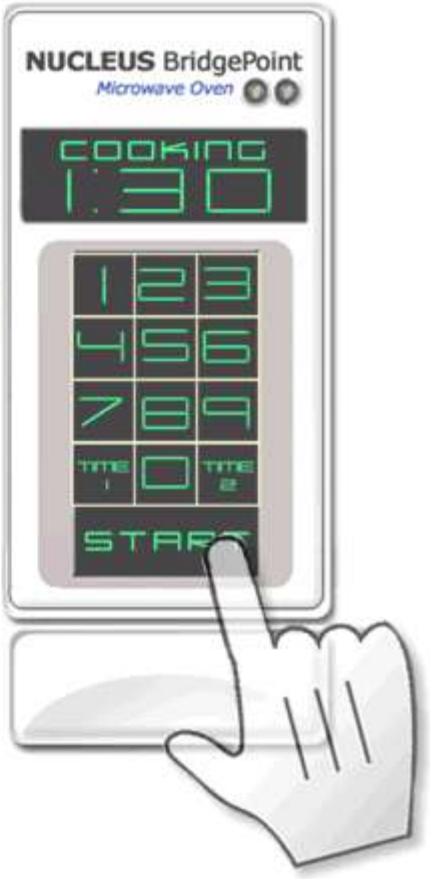
- Data
- Behavior



Coding Behavior in UML

3

Start
Cook
Wait
Cook
Start
Wait
Stop
Stop



State Event Matrix for the Oven

```
static const StateNumber_t Oven_StateEventMatrix[3][4] =
{
    /* Row 1: Oven_State_1 'Not_Cooking' */
    { Oven_State_2,  CANT_HAPPEN,  IS_IGNORED,  IS_IGNORED },
    /* Row 2: Oven_State_2 'Checking_Safety' */
    { Oven_State_2,  Oven_State_3,  IS_IGNORED,  IS_IGNORED },
    /* Row 3: Oven_State_3 'Cooking' */
    { CANT_HAPPEN,  CANT_HAPPEN,  Oven_State_3,  Oven_State_1 }
};
```

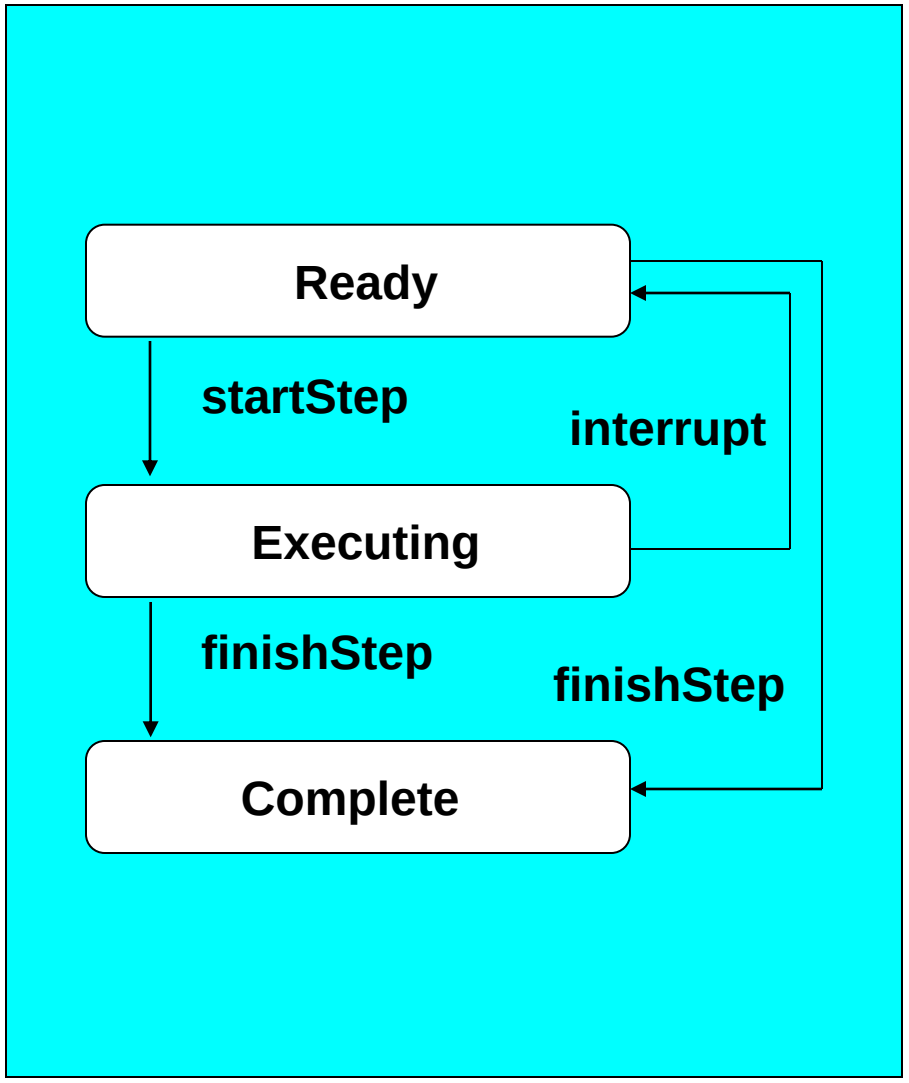
…and the Cooking Step

```
static const StateNumber_t Cooking_Step_StateEventMatrix[3][3] =
{
    /* Row 1: Cooking_State_1 'Ready' */
    { Cooking_State_2,  IS_IGNORED,  Cooking_State_3 },
    /* Row 2: Cooking_State_2 'Executing' */
    { IS_IGNORED,  Cooking_State_1,  Cooking_State_3 },
    /* Row 3: Cooking_State_3 'Complete' */
    { IS_IGNORED,  IS_IGNORED,  IS_IGNORED }
};
```

Lifecycles

Build a lifecycle model for each class.

Lifecycle for *Cooking Step*



State Event Matrix for Oven

State \ Signal	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**

Event dispatching code for the Oven

```
        . . .
state = instance->current_state;
next_state = Oven_StateEventMatrix[ state ][ event_number ];
if ( next_state <= 3 )
{
    /* Update state and execute the state action */
    instance->current_state = next_state;
    ( *Oven_Actions[ next_state ] )( instance, event );
}
        . . .
```

… and for Cooking Step

```
        . . .
state = instance->current_state;
next_state = Cooking_Step_StateEventMatrix
        [ state ][ event_number ];
if ( next_state <= 3 ) {
    /* Update state and execute the state action */
    instance->current_state = next_state;
    ( *Cooking_Step_Actions[ next_state ] )( instance, event );
}
        . . .
```

..and the rule to produce it

```
        . . .
state = instance->current_state;
next_state = <class name>_StateEventMatrix
        [ state ][ event_number ];
if ( next_state <= <class num_states> )
{
    /* Update state and execute the state action */
    instance->current_state = next_state;
    ( *<class name>_Actions[ next_state ] )( instance, event );
}
        . . .
```

Constructing the State Event Matrix

```
static const State_Number_t <class name>_StateEventMatrix
       [<class states>][<class events>] =

{

   /* Row <state1 number>:  <state1 name> */

   { <state1event1 next_state number>,
      <state1event2 next_state number>, . . .  }

   /* Row <state2 number>:  <state2 name> */

   { <state2event1 next_state number>,
      <state2event2 next_state number>, . . .  }

      . . .

};
```

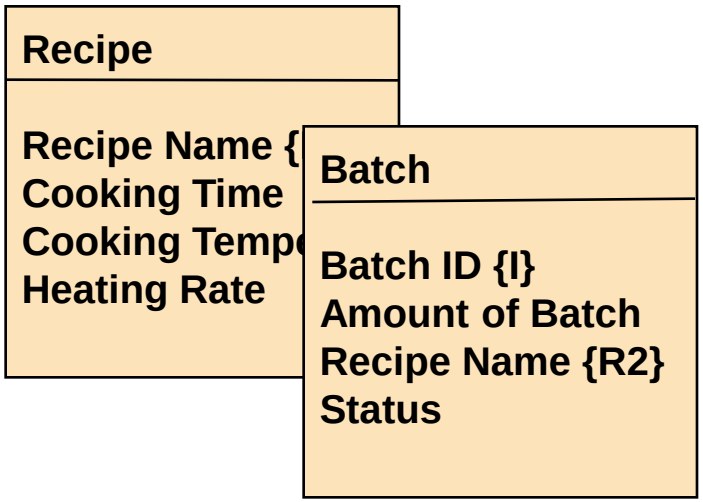
Coding Communication in UML

4



Instances

An executable model operates on data about instances.



Recipe

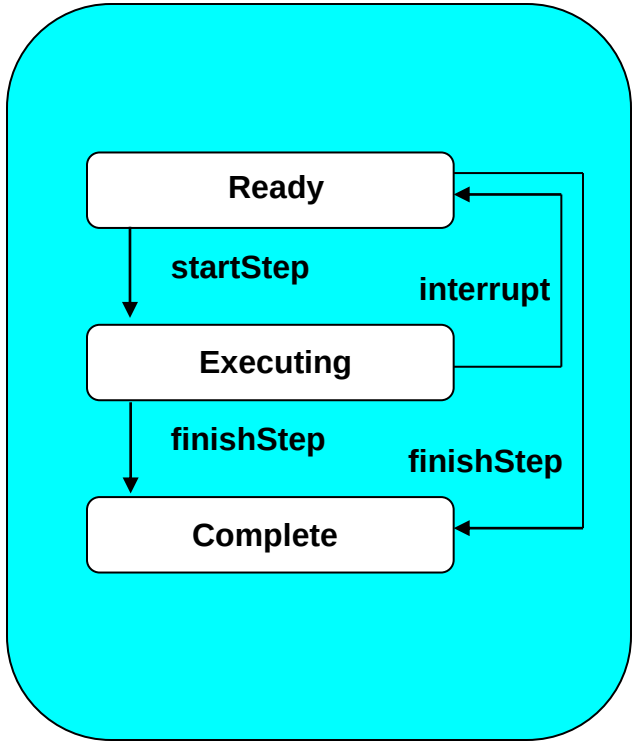
Recipe Name	Cooking Time	Cooking Temp	Heating Rate
Nylon	23	200	2.23
Kev			
Stu			

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

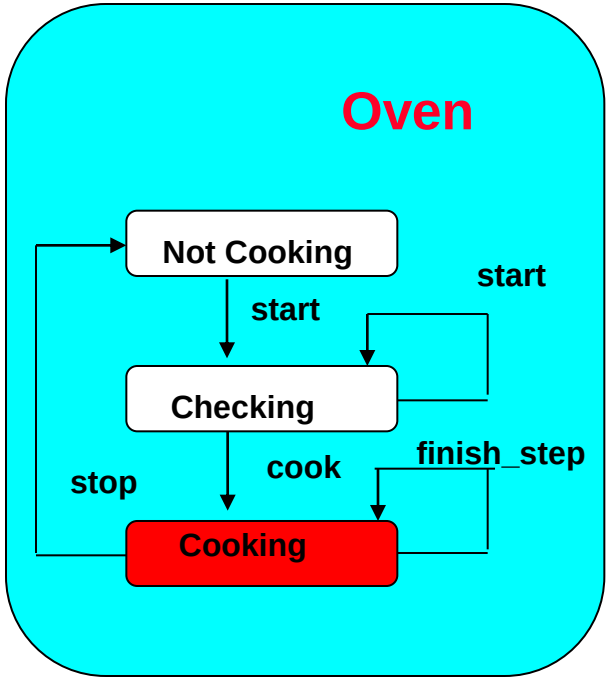
Executing the Model

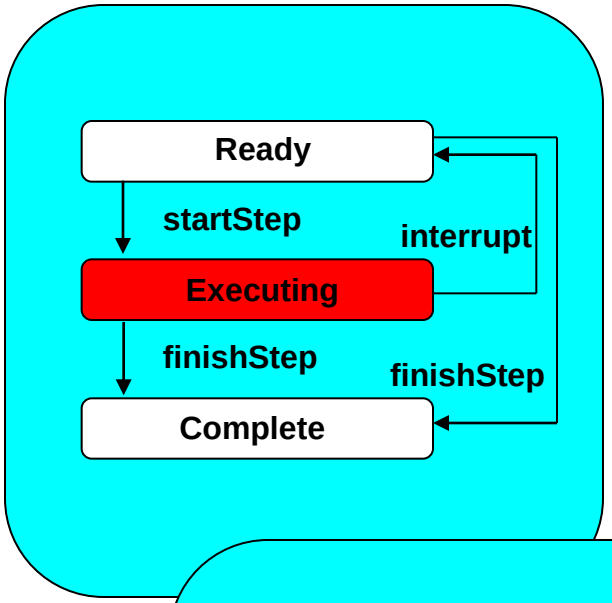
- The model executes in response to signals from:
  - the outside,
  - timers
  - other instances as they execute

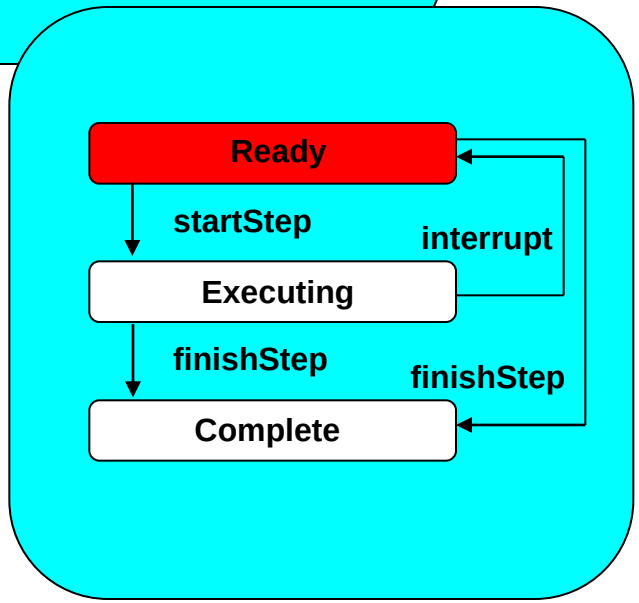


Instances

- An executable model operates on instances.

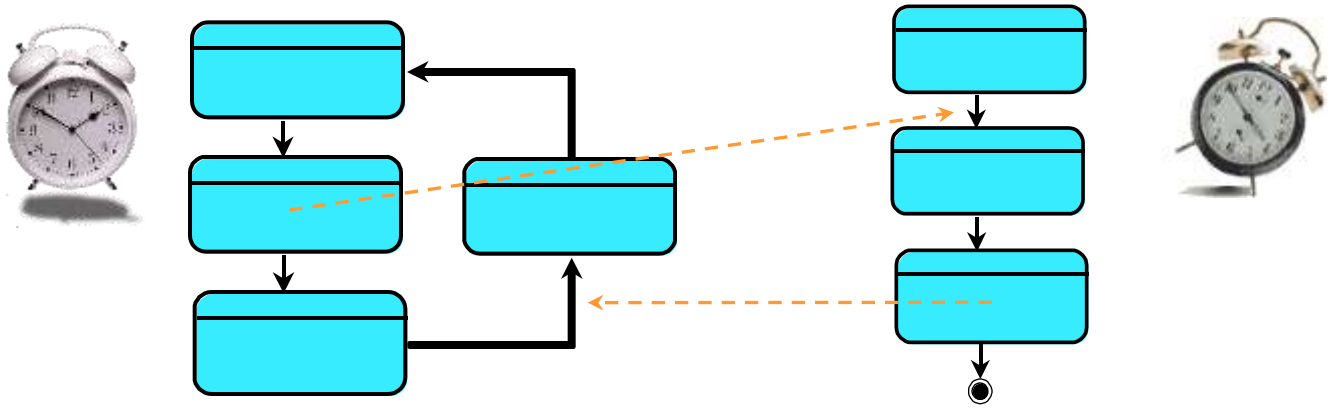






How does time work….?

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

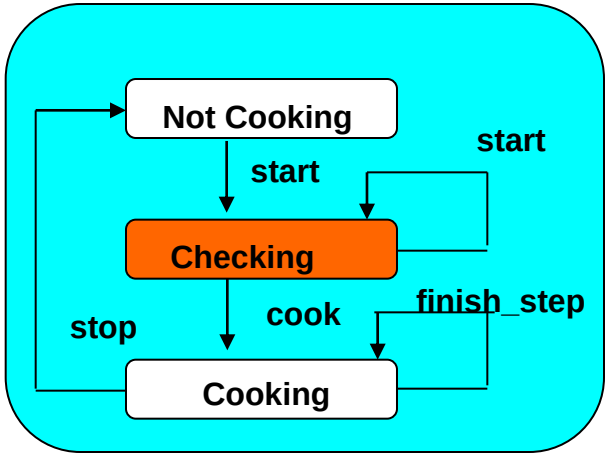


© 2005 Model Integration, LLC

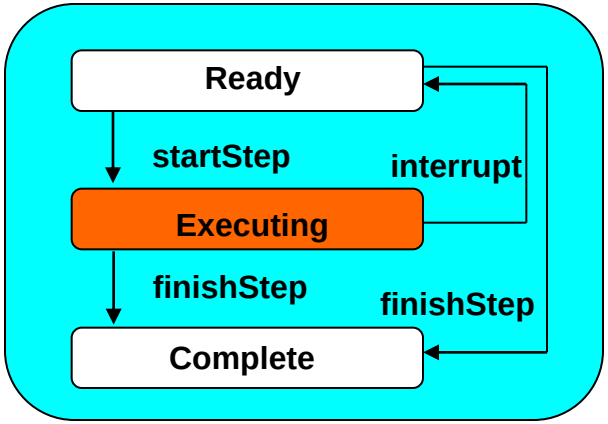
Concurrent Execution

All instances execute concurrently.

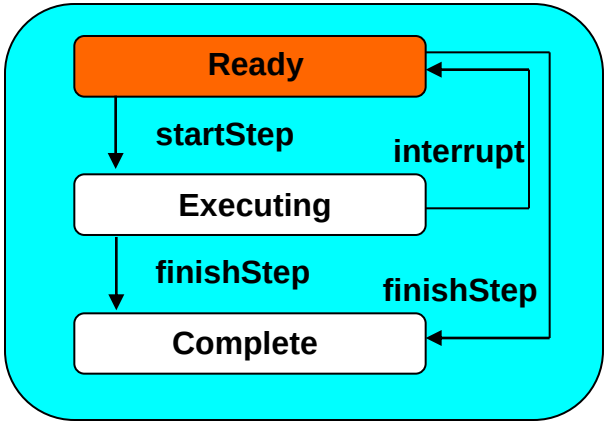
Oven



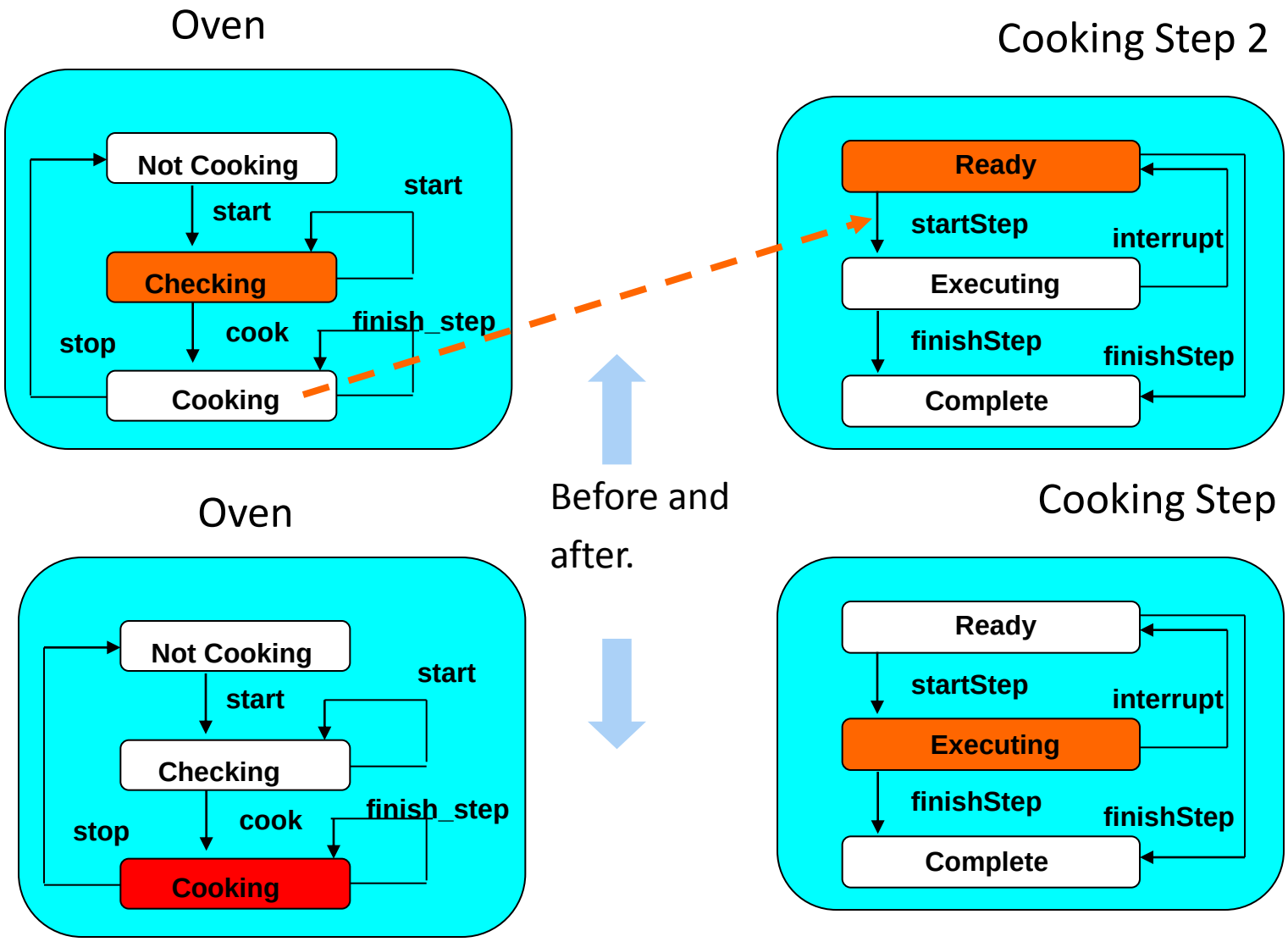
Cooking Step 1



Cooking Step 2

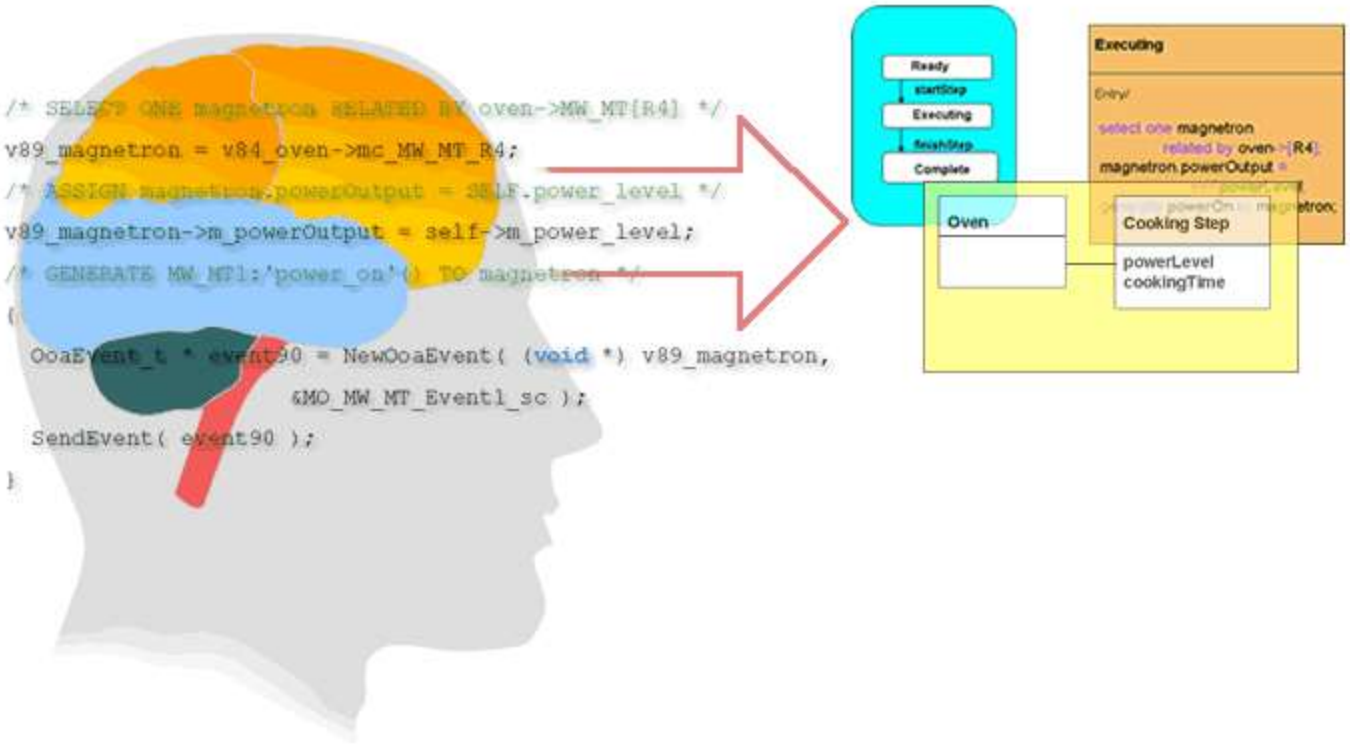


Communication



Coding Logic in UML

5

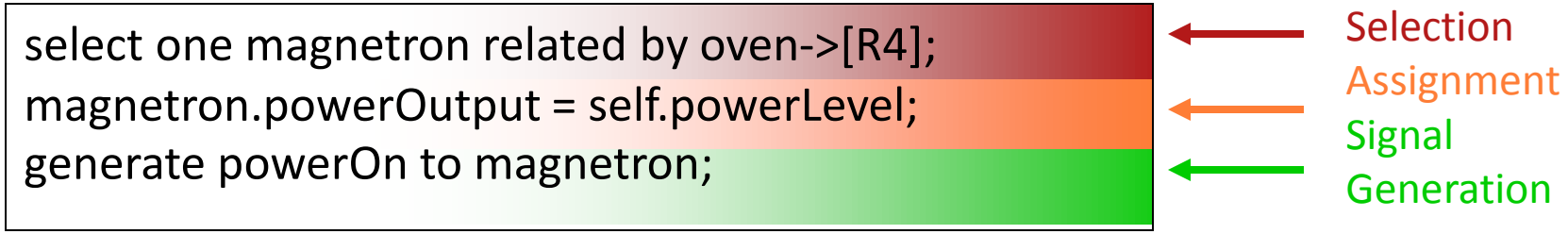


Coding Logic in UML

```
/* SELECT ONE magnetron RELATED BY oven->MW_MT[R4] */
v89_magnetron = v84_oven->mc_MW_MT_R4;
/* ASSIGN magnetron.powerOutput = SELF.power_level */
v89_magnetron->m_powerOutput = self->m_power_level;
/* GENERATE MW_MT1:'power_on'() TO magnetron */
{
  OoaEvent_t * event90 = NewOoaEvent(
          (void *) v89_magnetron, &MO_MW_MT_Event1_sc );
  SendEvent( event90 );
}
```

Activity

- Progression of the state machines causes execution of an activity.
- The activity is written in an *action language*.



- Each statement may be translated into code in different ways, depending on choice of data and control structure.

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

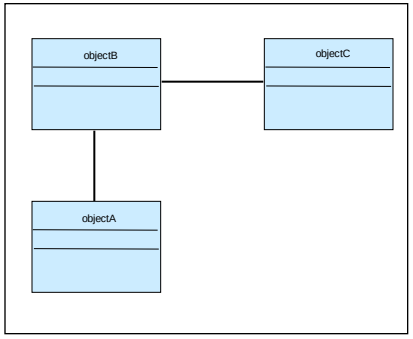
This is commonly called *run-to-completion* semantics

Executable elements

- Data
- Control
- Processing

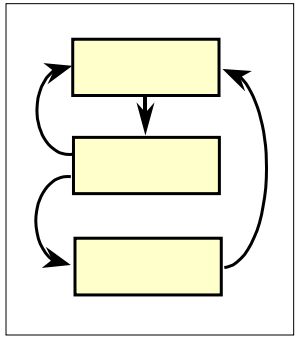


Class Model



Class Diagram

State Models



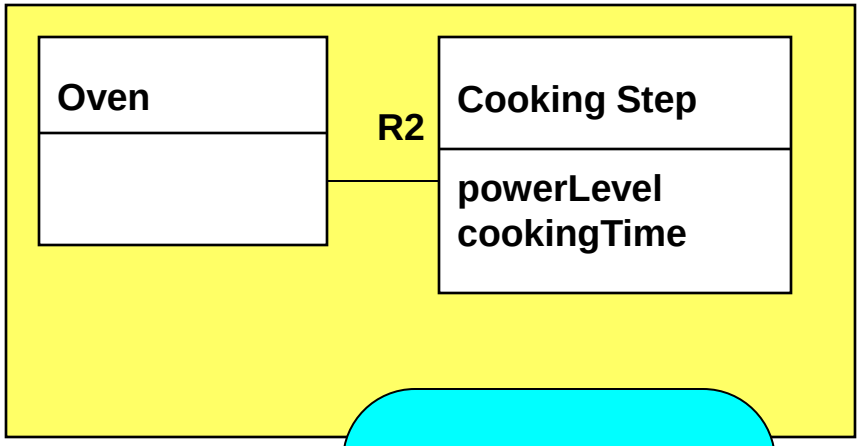
State Charts

Activity Models

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

Action Language

Three Primary Diagrams

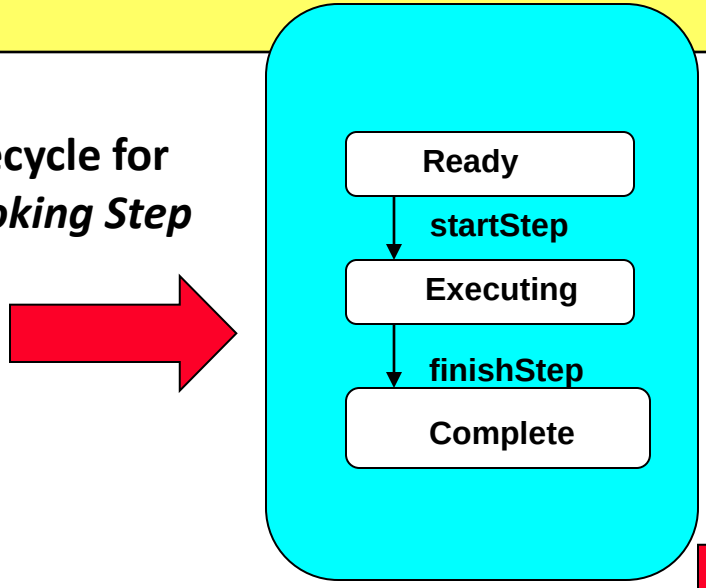


Class Diagram

Statechart Diagram

Action Language

Lifecycle for *Cooking Step*

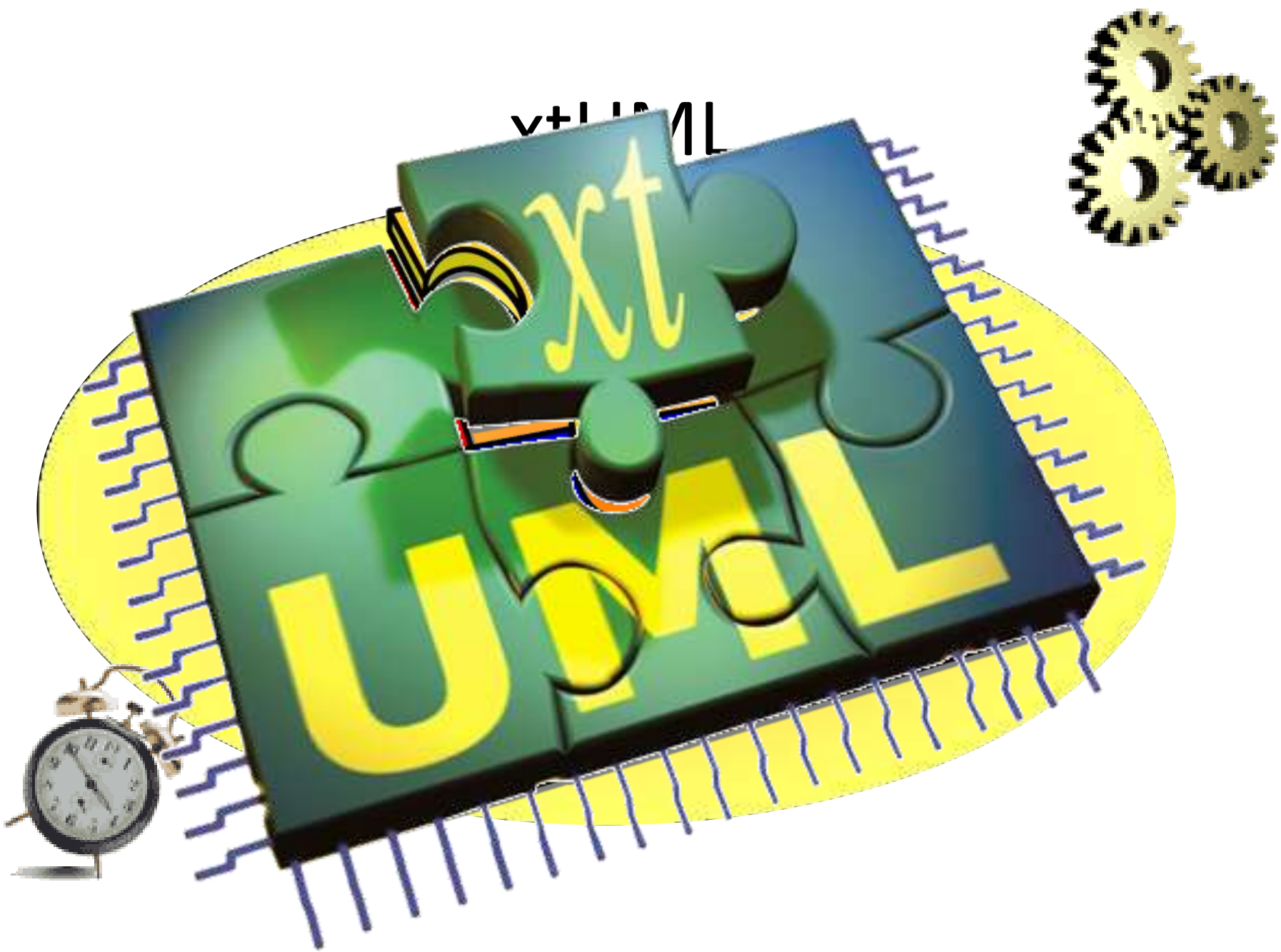


Action for state *Executing*

Executing

Entry/

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


xt
UML

xtUML

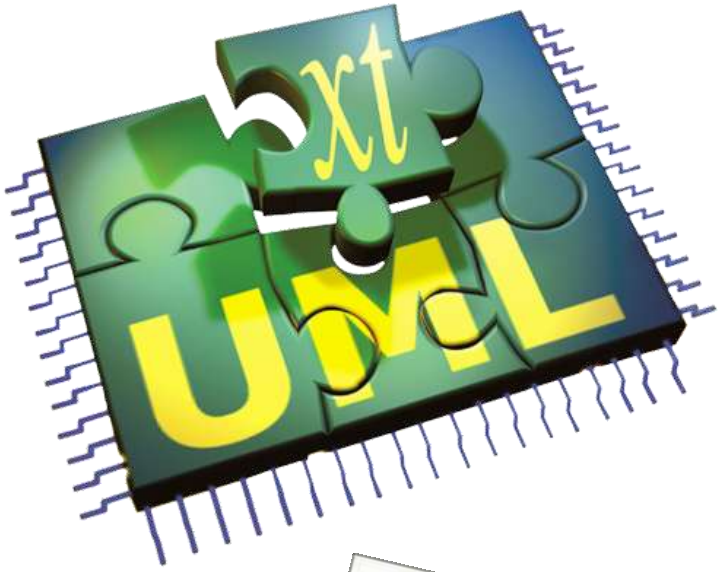
xtUML is a streamlined subset of the UML industry standard that:

(x) Executes models

- Allows for early verification
- Integration of legacy code

(t) Translates models

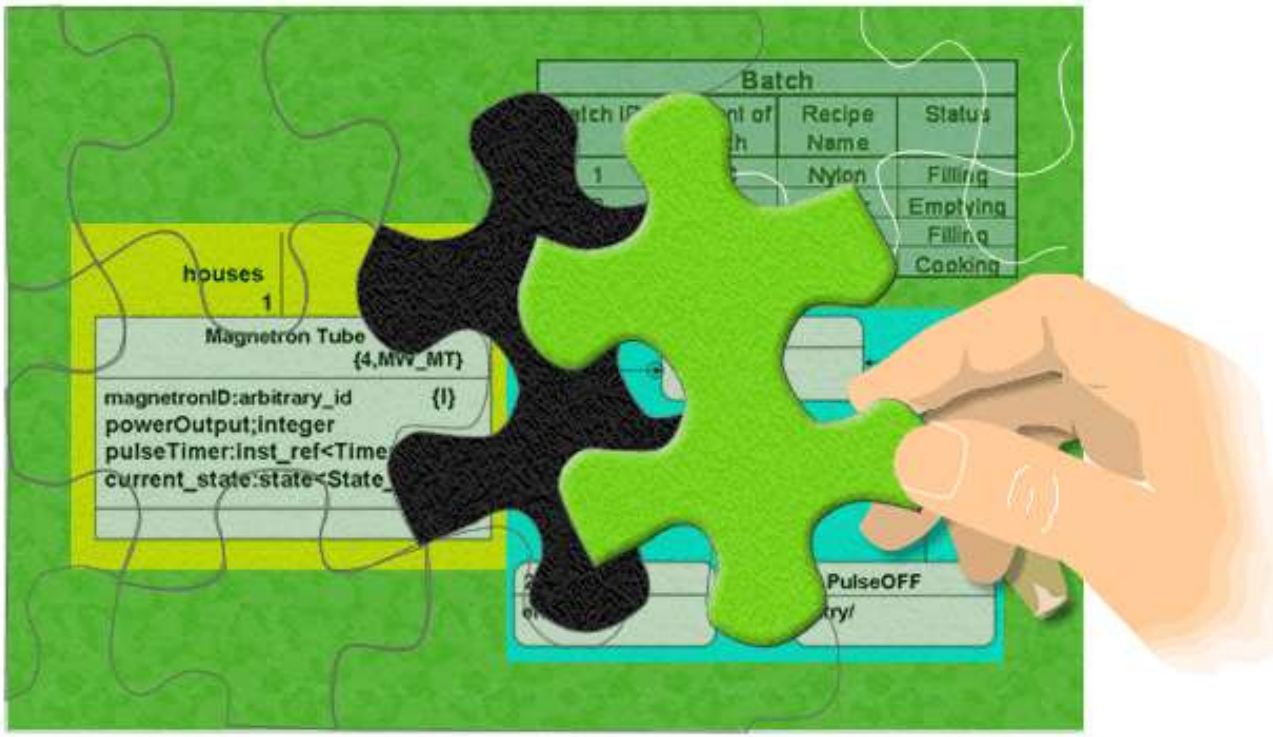
- Complete code generation from models
- Customizable compilation rules
- Optimized code





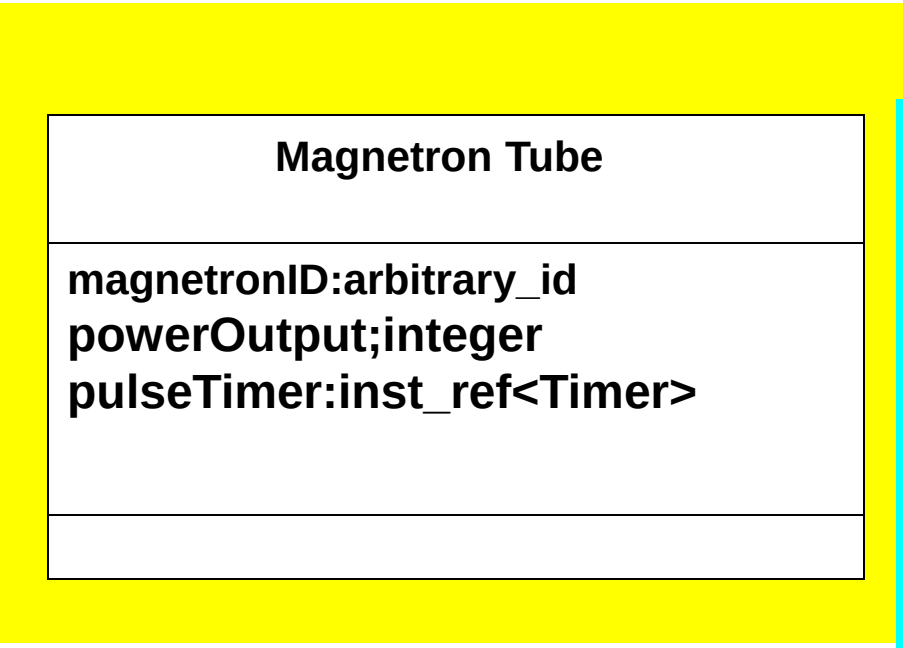
Completing the Models

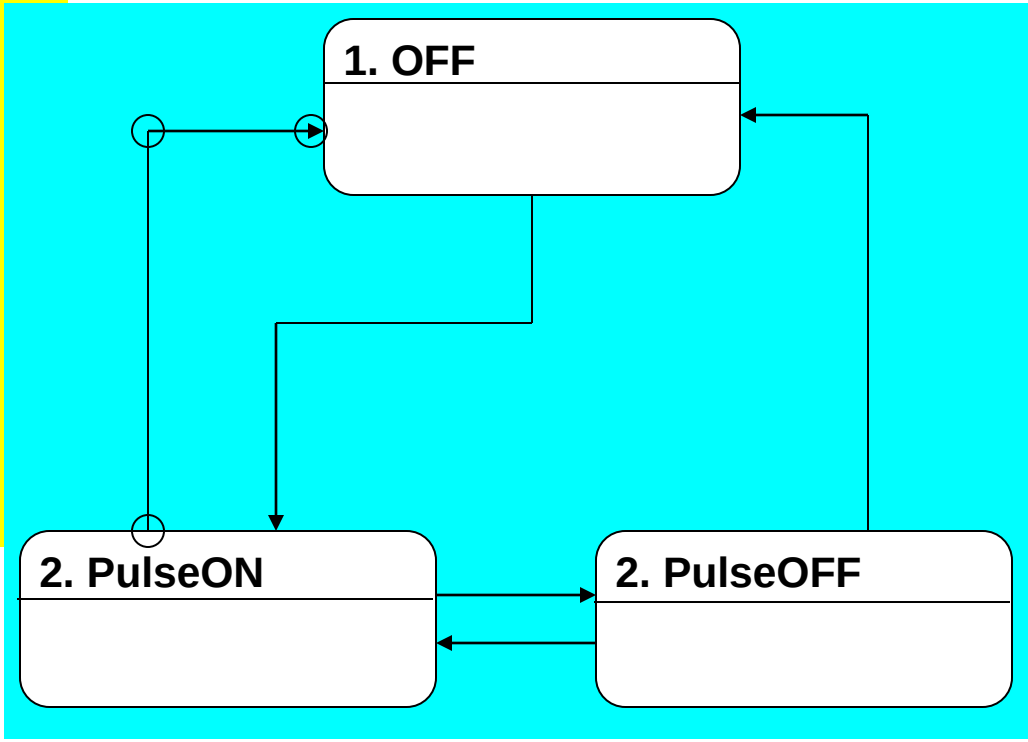
6



Timers

To build an implementation, we will use timers.





Timers

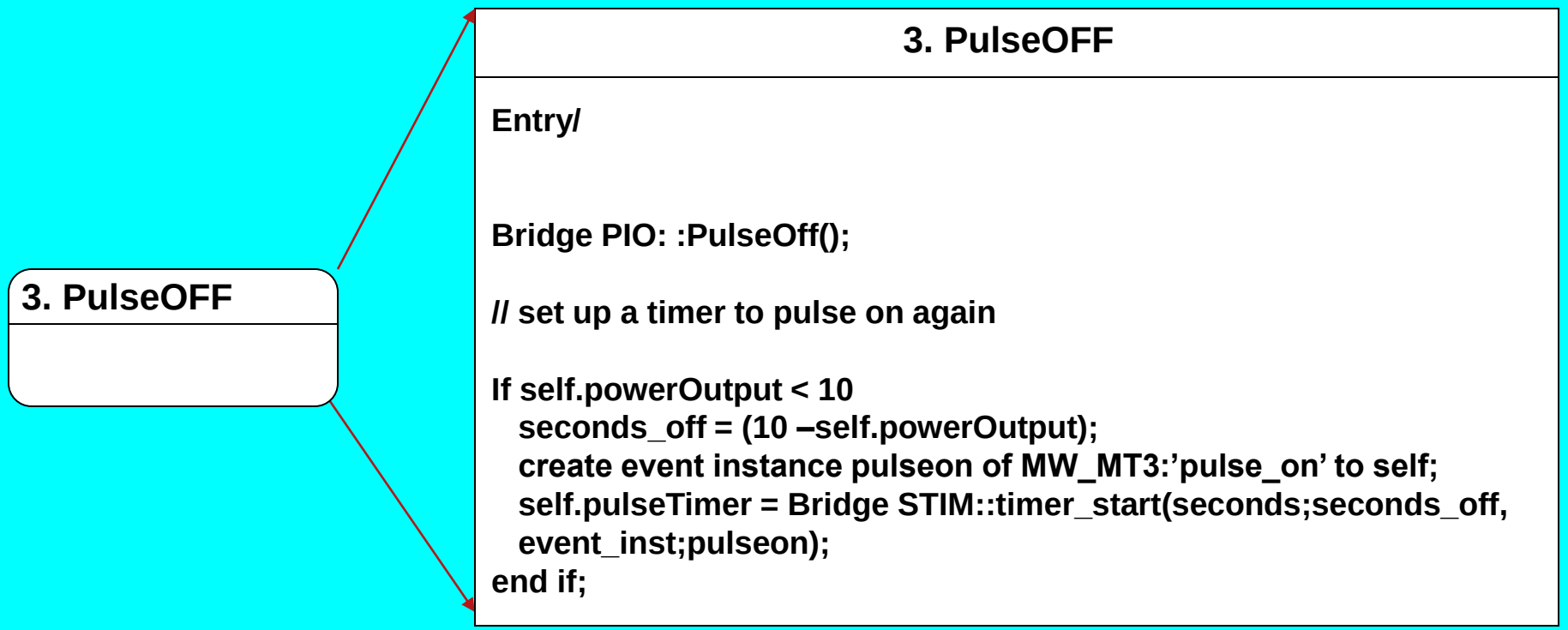
Timers are generally provided.

<<external entity>>
SecondsTimer

<operation>
timer_start(seconds : integer, event_inst : inst<Event>) : inst_ref <Timer>
timer_cancel(timer_inst_ref : inst_ref<Timer>) : boolean
timer_remaining_time(timer_inst_ref : inst_ref<Timer>) : integer

<<operation>>
current_date : date
create_date(second : integer,minute : integer,hour : integer,day : integer,month : integer, year
get_second(date : date) : integer
get_minute(date : date) : integer
get_hour(date : date) : integer
current_clock : timestamp
timer_start(microseconds : integer, event_inst : inst<Event>) : inst_ref<Timer>
timer_reset_time(timer_inst_ref : inst_ref<Timer>, microseconds : integer) : boolean
timer_add_time(timer_inst_ref : inst_ref<Timer>, microseconds : integer) : boolean
timer_cancel(timer_inst_ref : inst_ref<Timer>) : boolean

Tying an instance of an Event to a Timer



Bridge to Hardware

We must also bridge to the hardware.

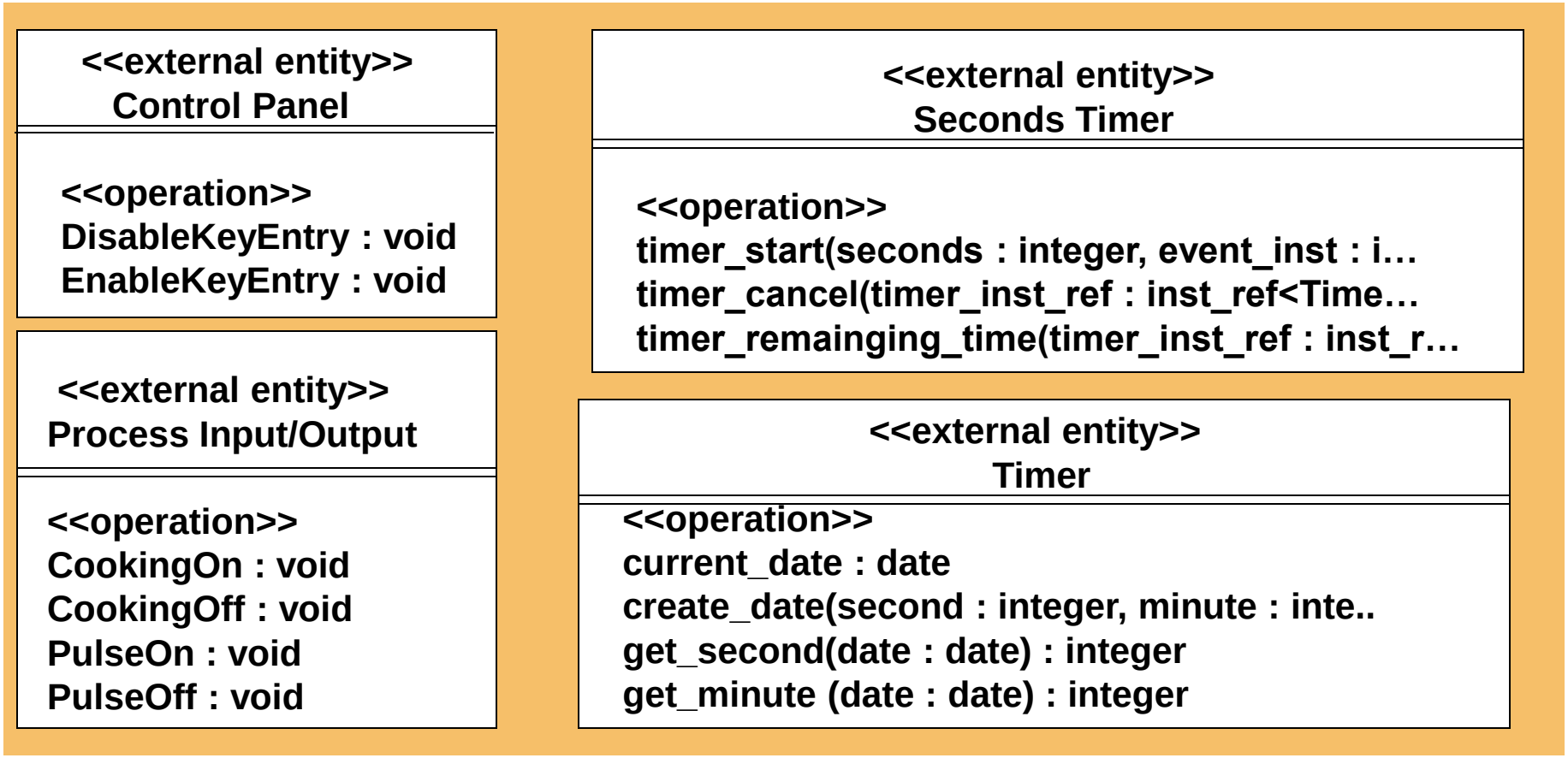
```
Bridge PIO::PulseOff();

// set up a timer to pulse on again

If self.powerOutput < 10
    seconds_off = (10 - self.powerOutput);
    create event instance pulseon of W_MT3:'pulse_on' to self;
    self.pulseTimer = Bridge STIM::timer_start(
                    seconds:seconds_off, event_inst:pulseon);
end if;
```

External Entities

The operations allow bridging from one domain to another.



Initialization

```
select any oven from instances of MW_OV;
If (empty oven)
    create object instance oven of MW_OV;

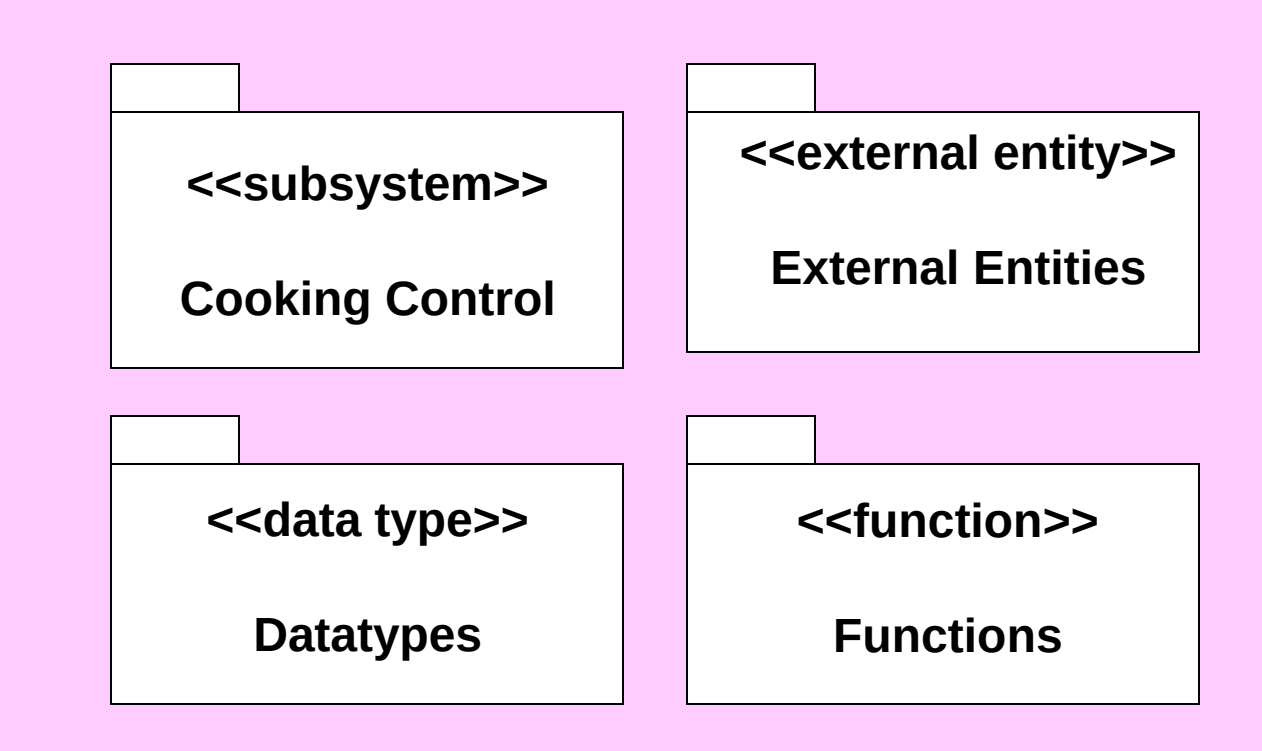
    create object instance door of MW_DR;
    relate door to oven across R1;
    assign door.isLocked = false;

    create object instance tube of MW_MT;
    relate tube to oven across R4;
    assign tube.powerOutput = 10;

    Bridge CP::EnableKeyEntry();
    // now that system is initialized
end if;
```

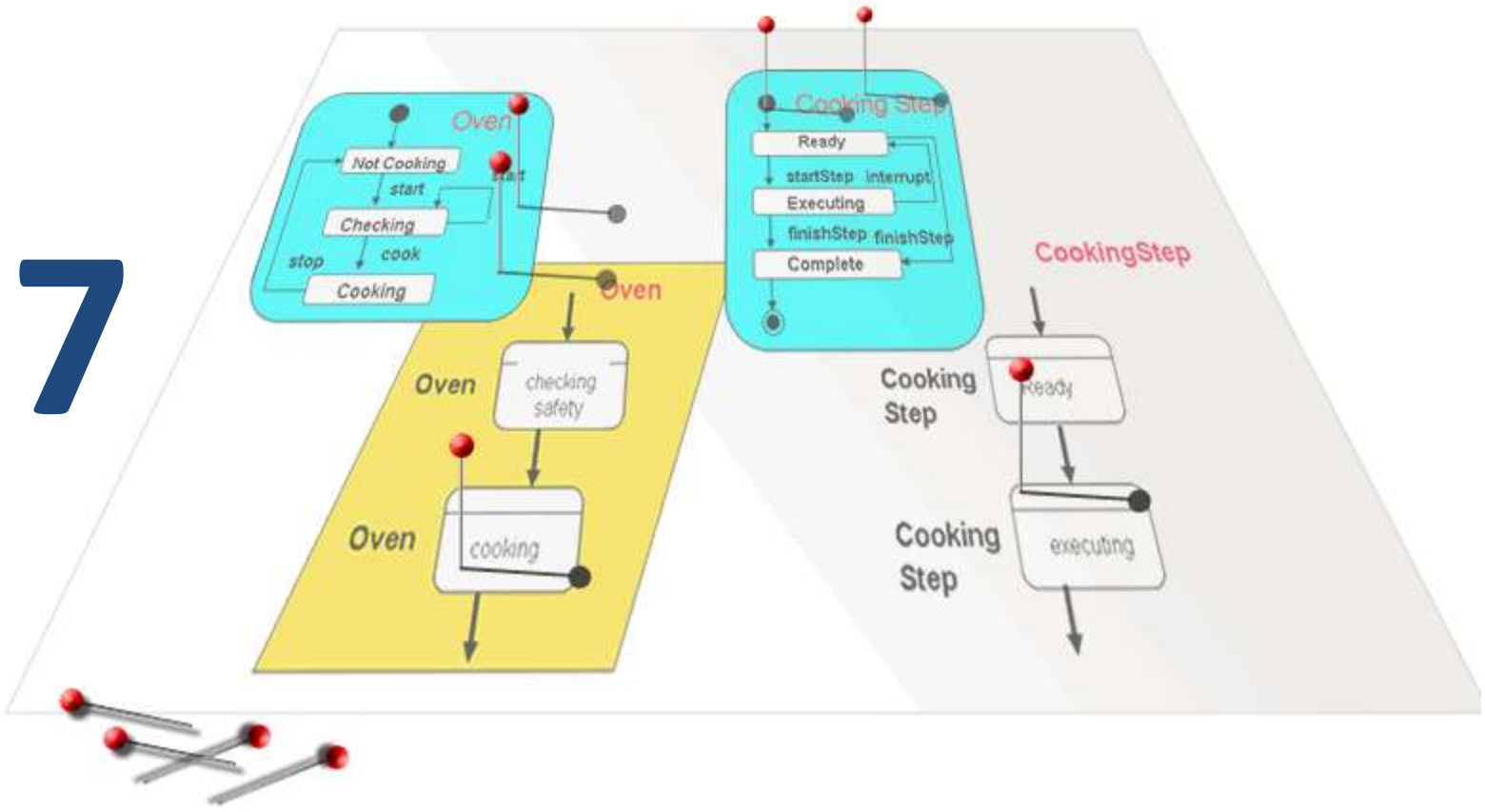
Domain Package Diagram

- Application
- Proxies for external entities
- Datatypes
- Functions



Mapping to an Implementation

7



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.

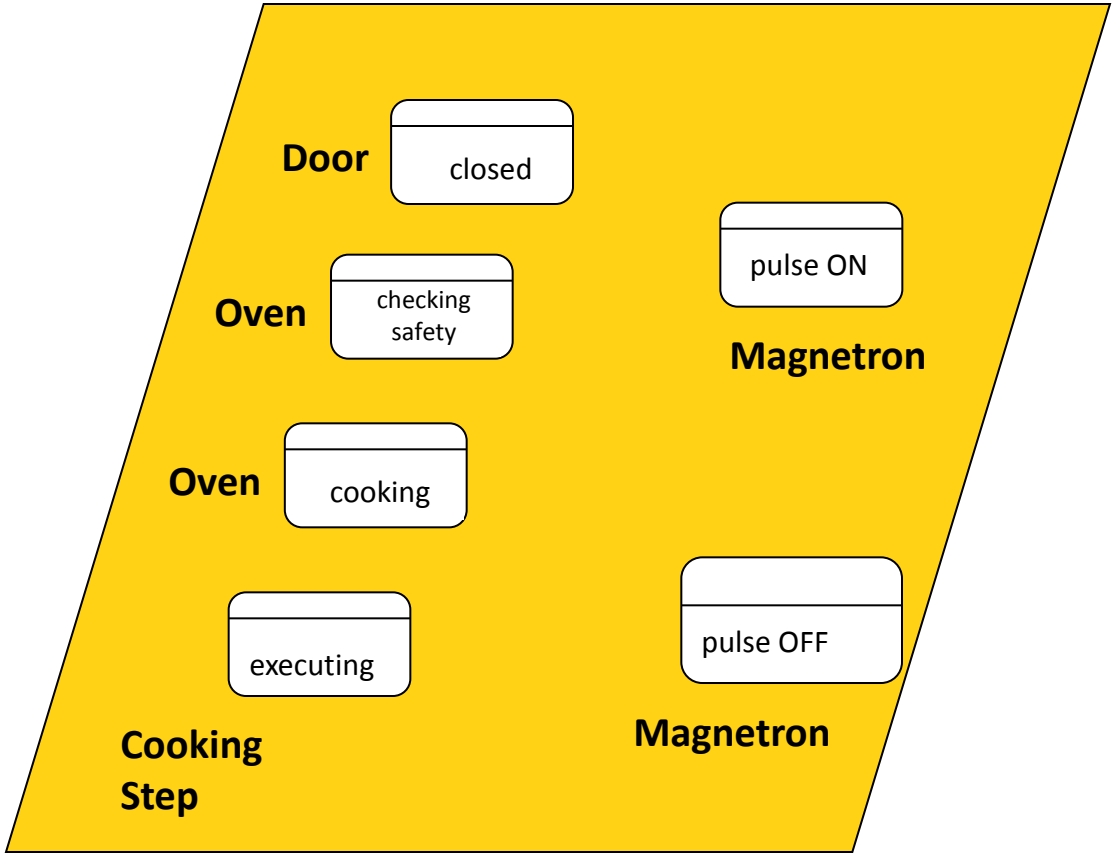
So, what should the structure of that implementation be?



Activities

- The basic unit of allocation is the activity.

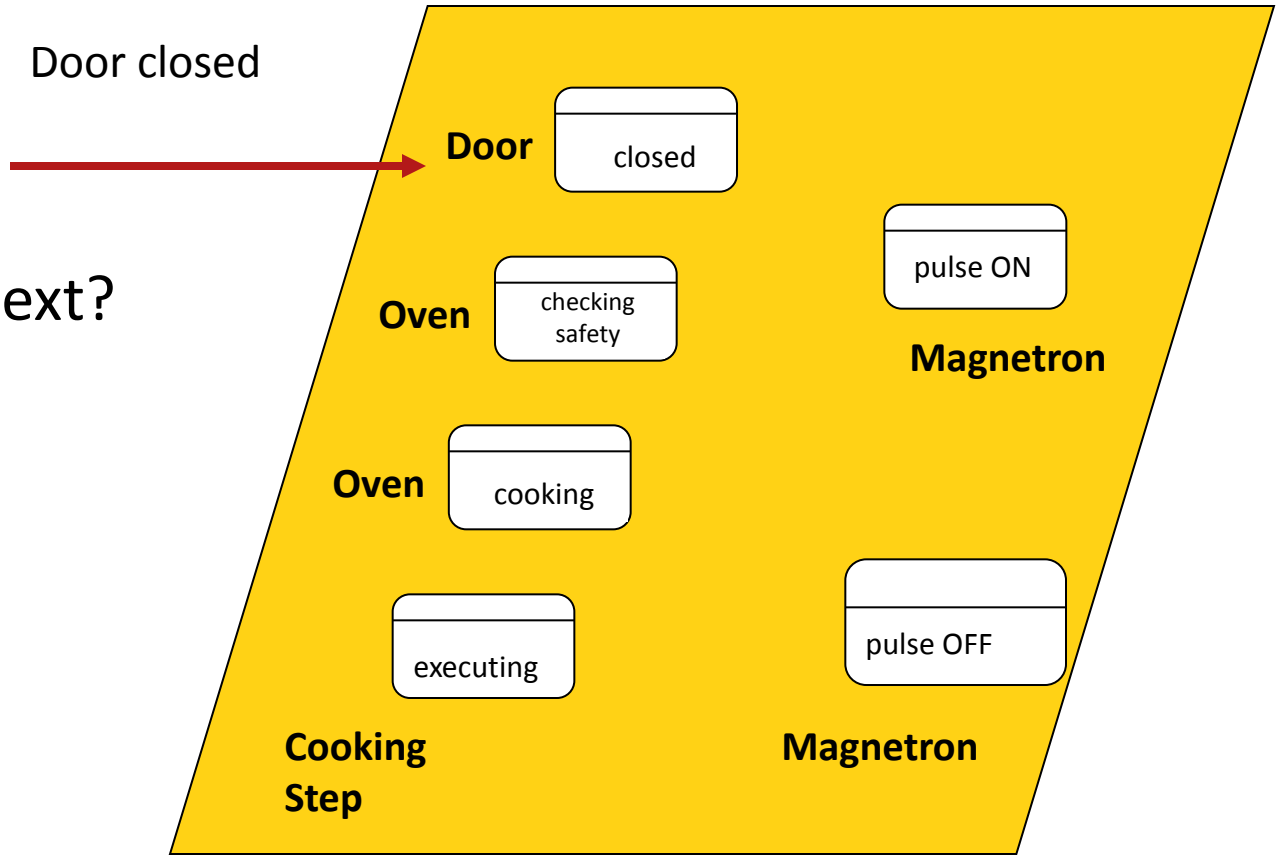
- Here, we allocate all activities to a single task.



Activities

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

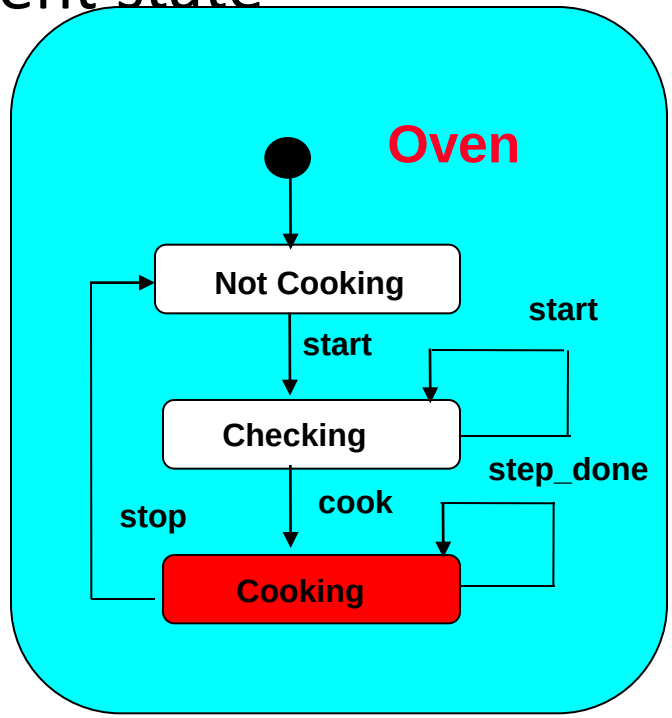
- What happens next?



Dequeuing Signals

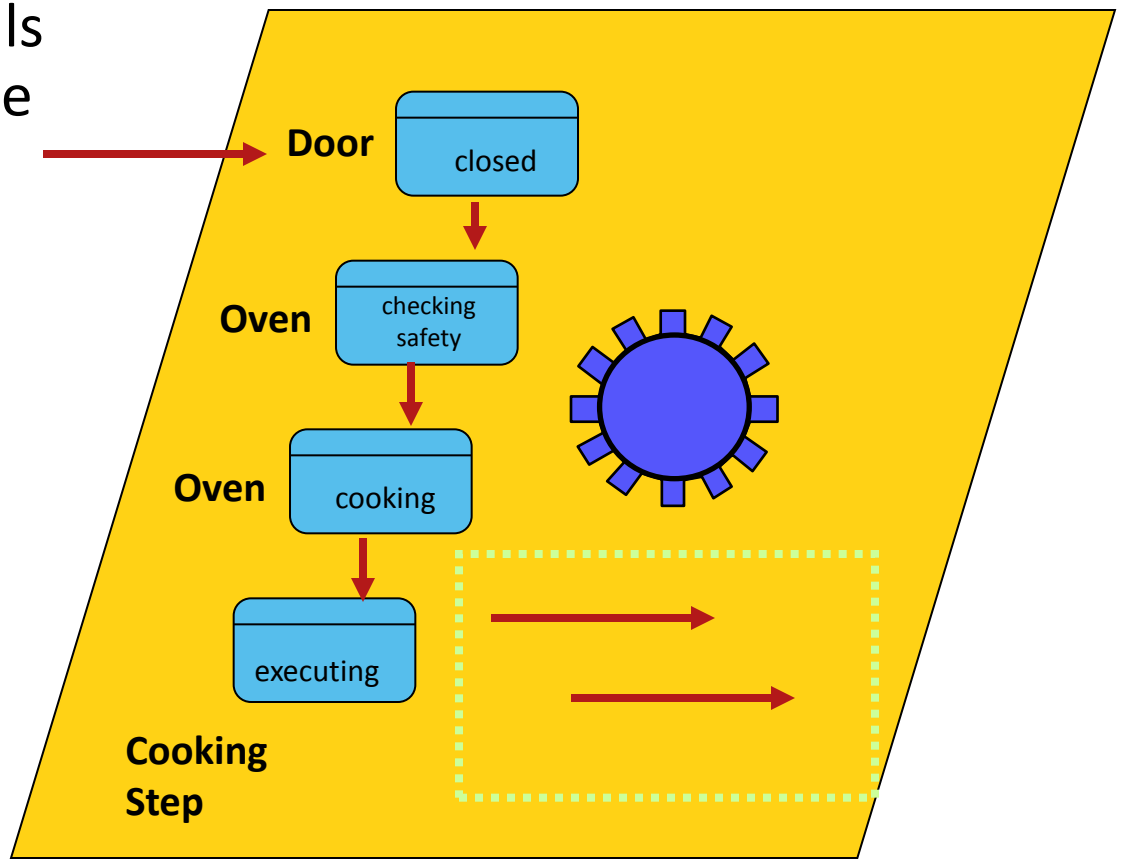
- When a signal is received, we:
  - Determine what to do next based on the name of the signal and the current state
  - Do it

State \ Signal	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

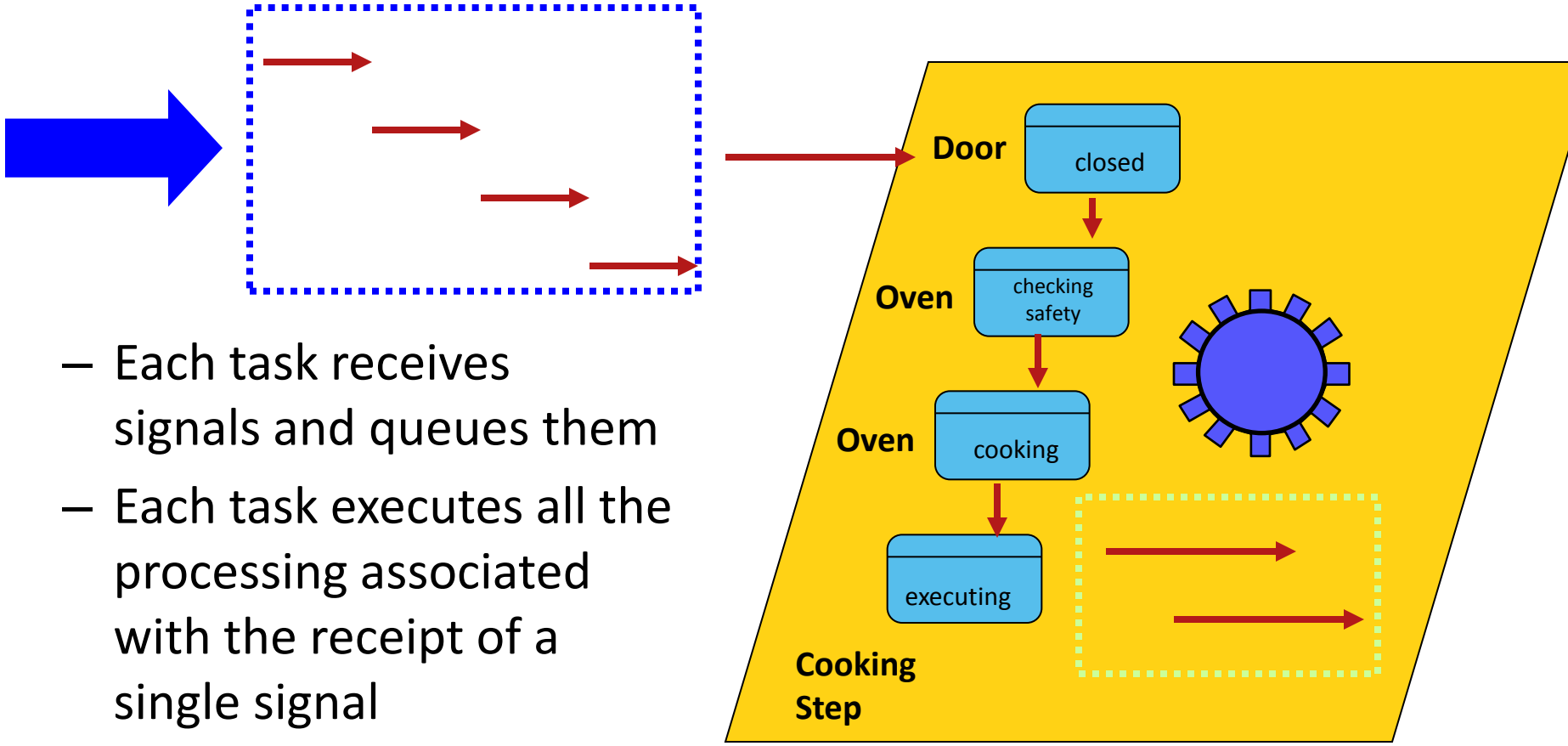


Communication

- 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



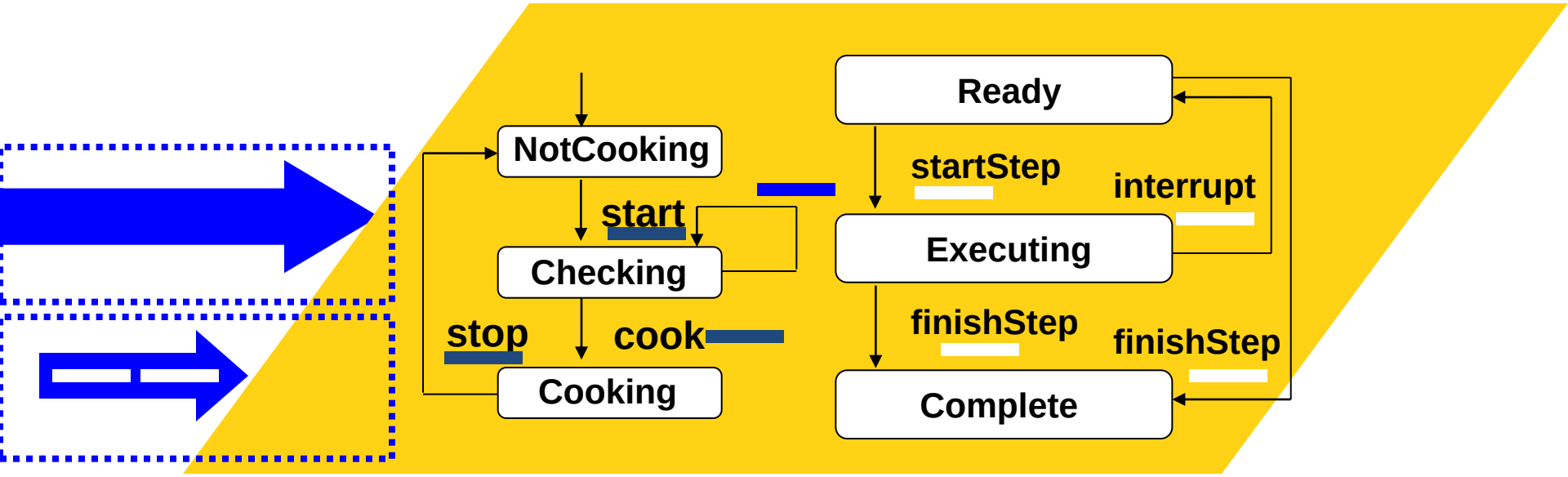
External Communication



- 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

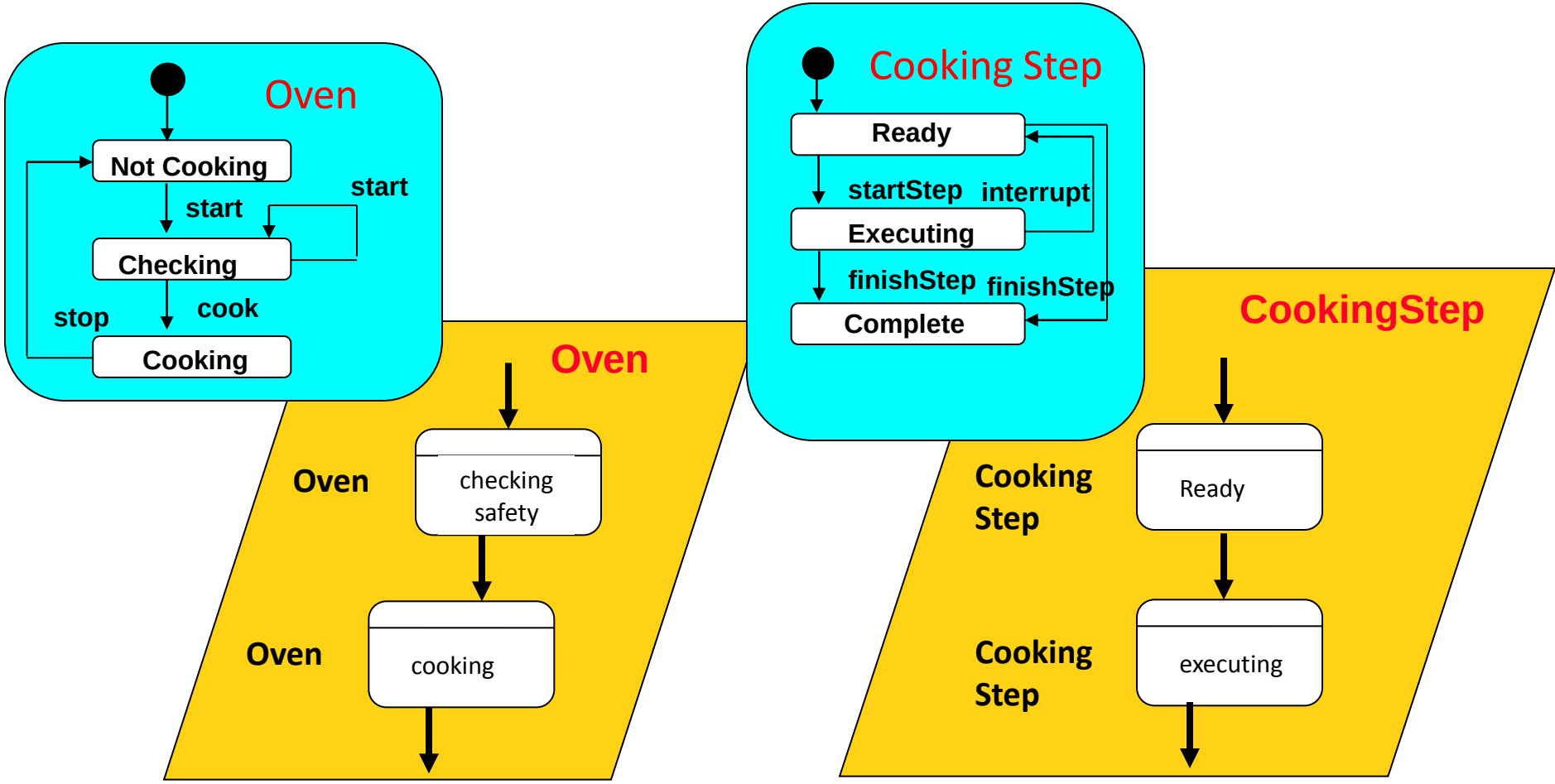
- Concurrency is simulated, but:
  - The “granularity of concurrency” is an activity
  - An activity must wait for another to finish
  - The waiting activity may be higher priority

  => Priority inversion

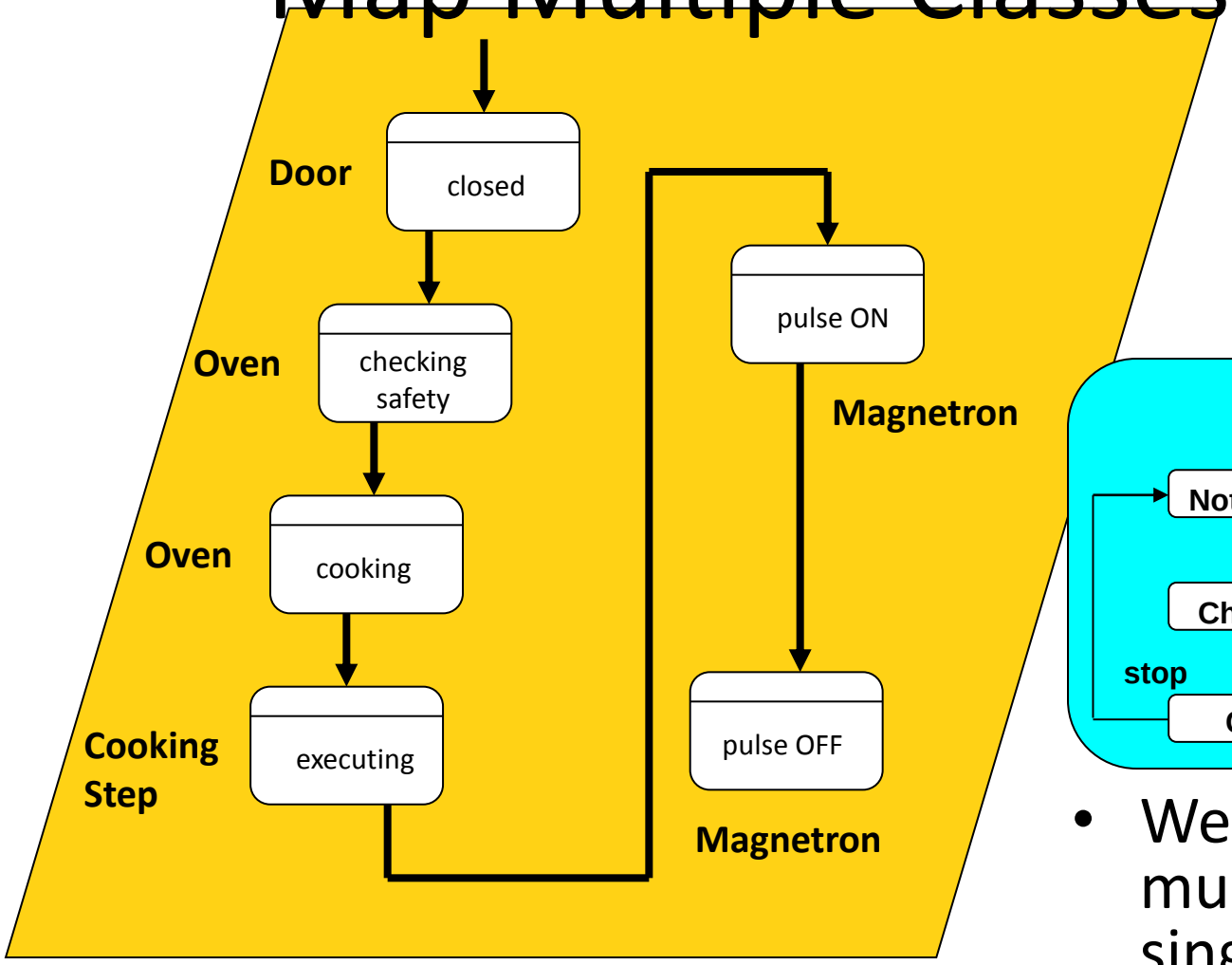


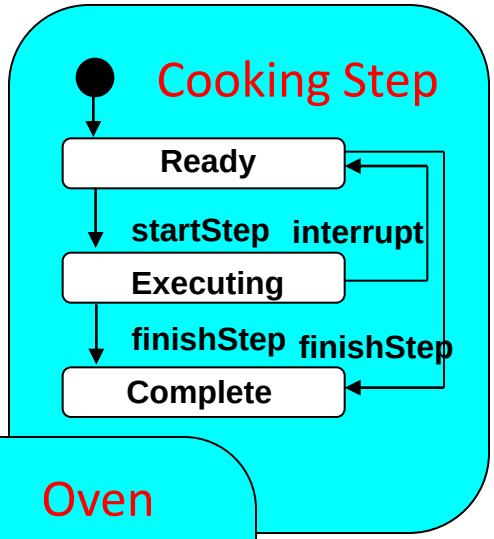
A Direct Mapping

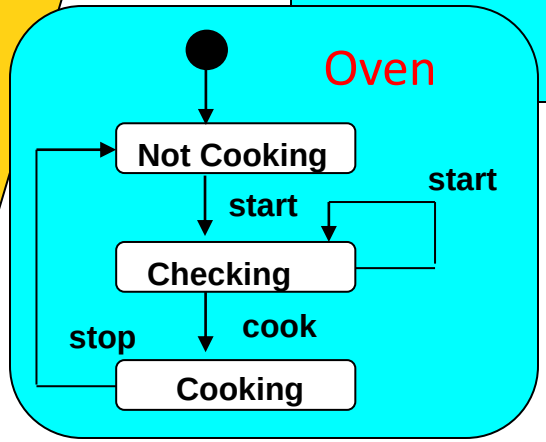
We may map each class to a separate task.



Map Multiple Classes



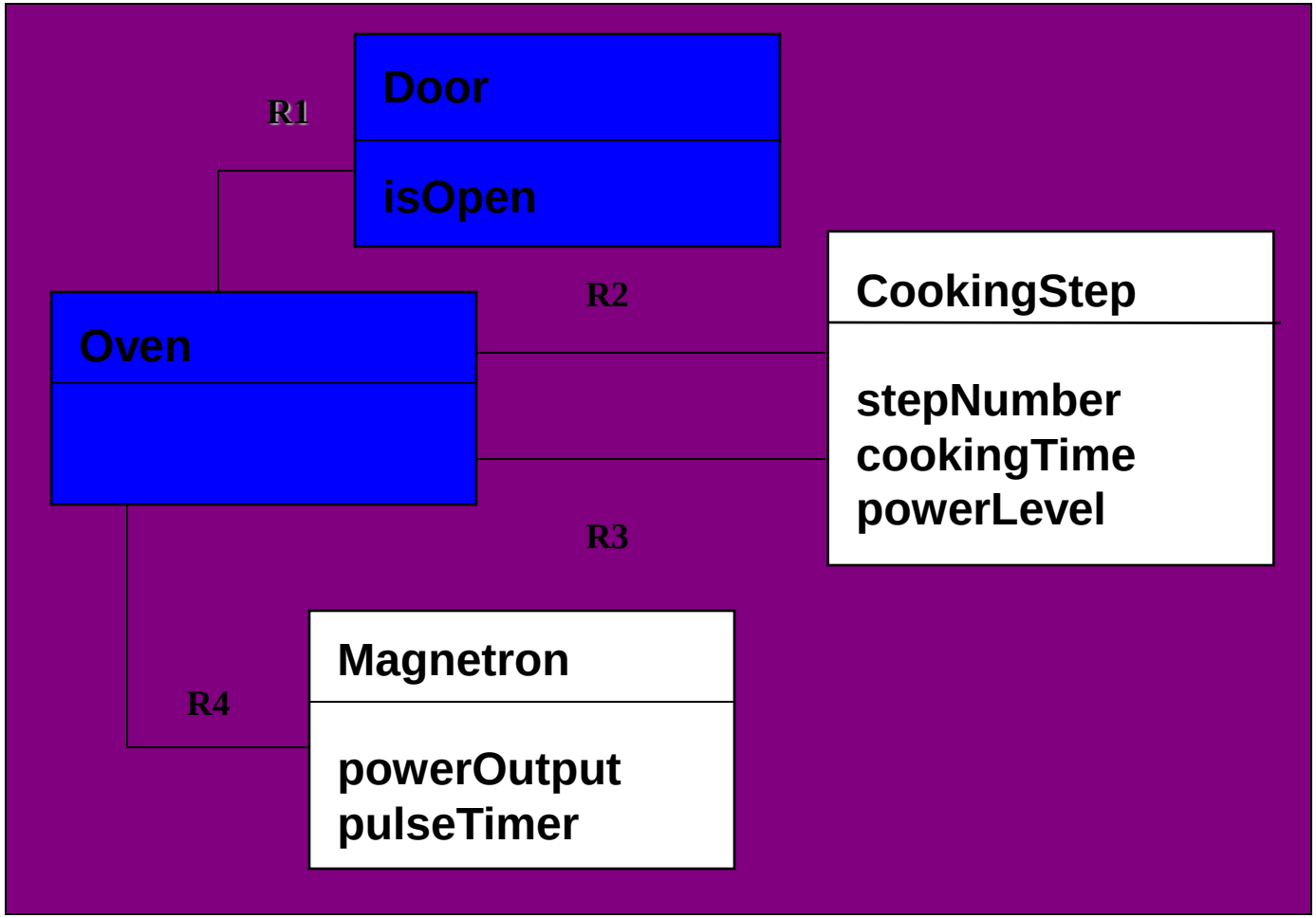




- We may map multiple classes to a single task.

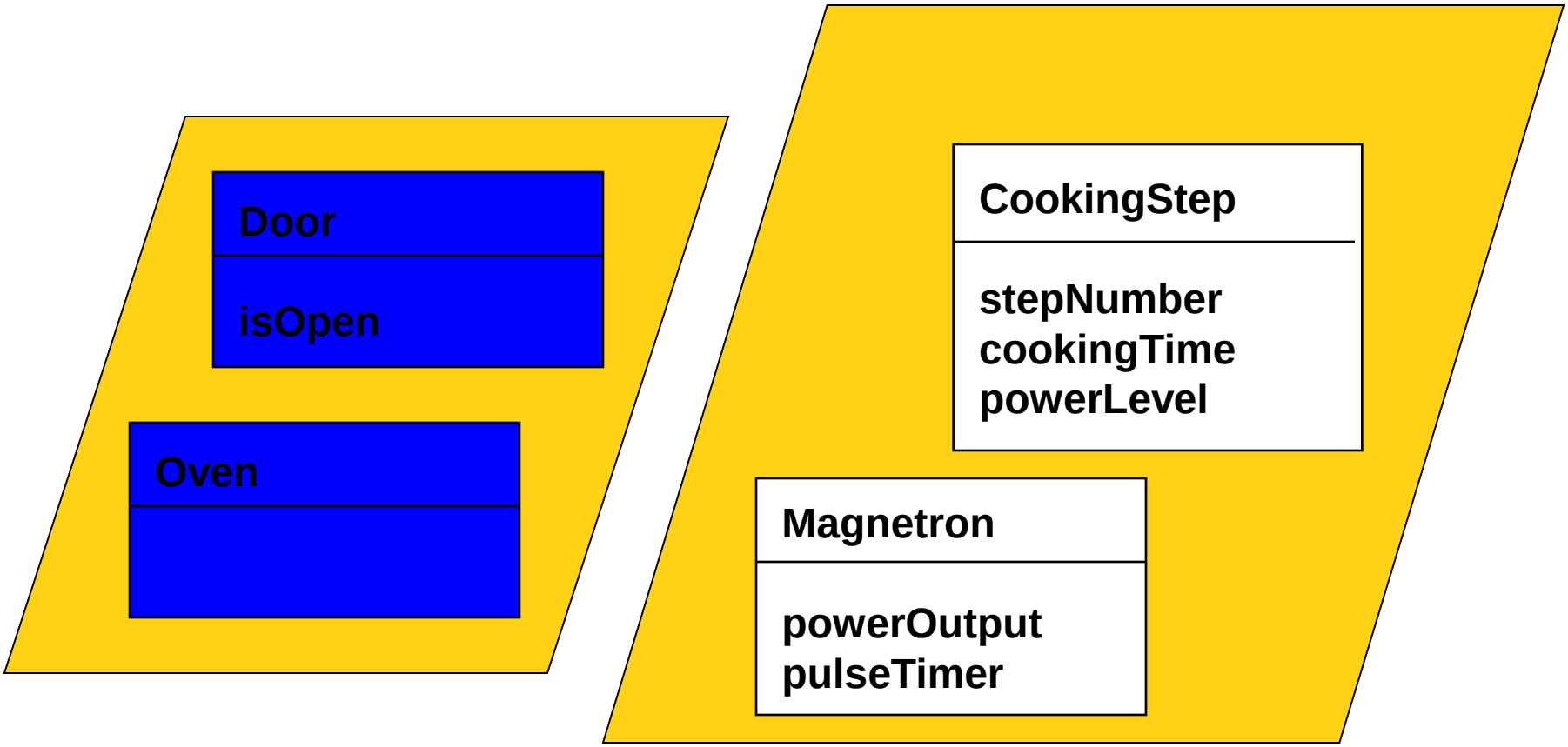
Multi-Tasking Concurrency

For this mapping, we choose to allocate classes.



Multi-Tasking Concurrency

For this mapping, we choose to allocate classes.



More information for translation

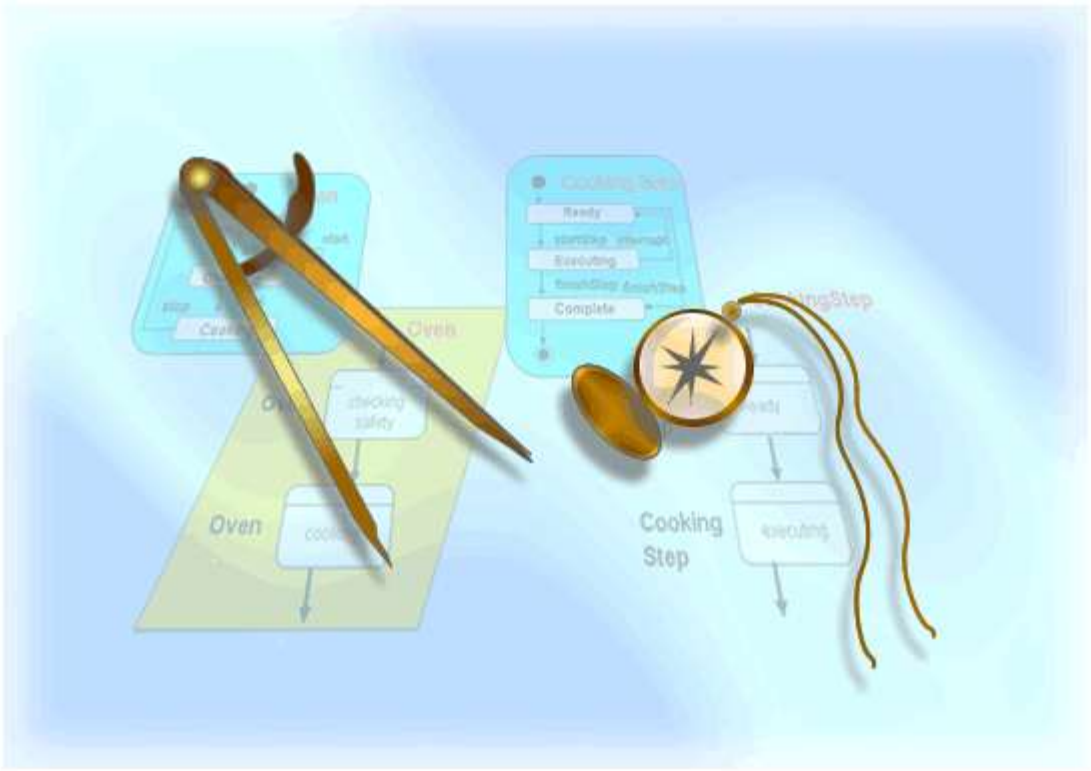
- We need also to state:
  - Priorities of tasks
  - Which classes go in which classes

- `.invoke EnableTasking( "Nucleus", 2)`
- `.invoke SetTaskPriority( 1, "low" )`
- `.invoke SetTaskPriority( 2, "high" )`

- `.invoke MapClassToTask( "OvenControl", "Door", 1 )`
- `.invoke MapClassToTask( "OvenControl", "Oven", 1 )`
- `.invoke MapClassToTask( "OvenControl", "Magnetron", 2 )`
- `.invoke MapClassToTask( "OvenControl", "CookingStep", 2 )`

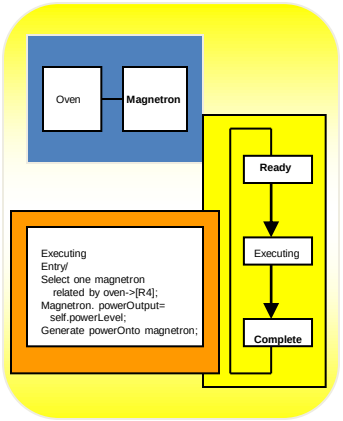
Mappings

8

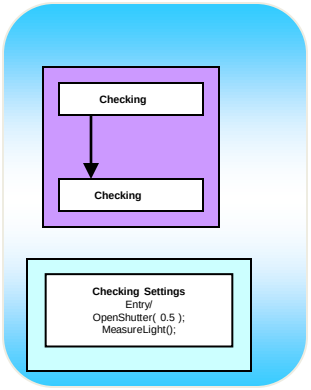


Separation

Application



- Design can be split between:
- Subject matter experts who understand the application, and
- Experts who understand the *application-independent* architecture
- Each evolves at its own pace.

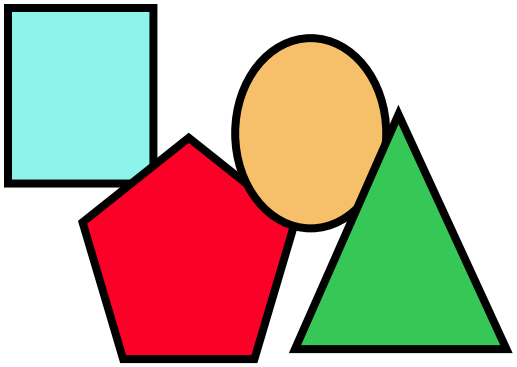


Architecture

Mappings

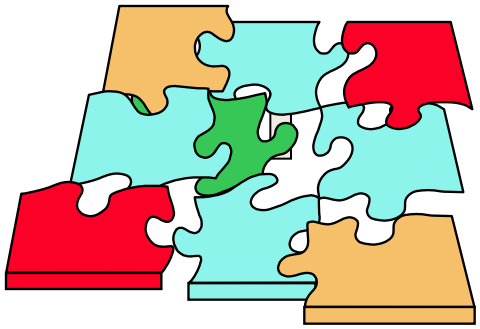
- Mapping by hand:
  - Is error prone
  - Not repeatable
  - Not scalable
  - Not predictable
  - Leads to integration problems





The knowledge

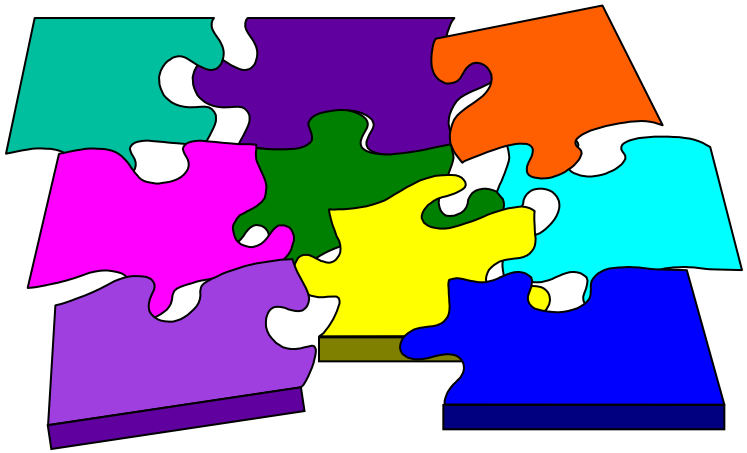




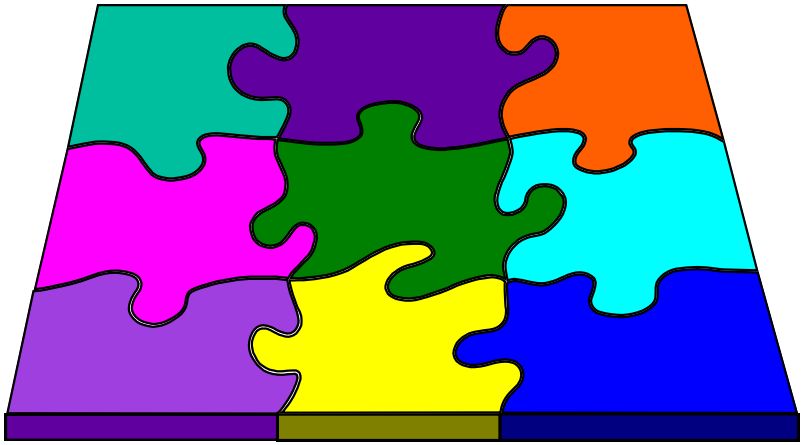
The software

Integration

- Large systems are difficult to integrate.

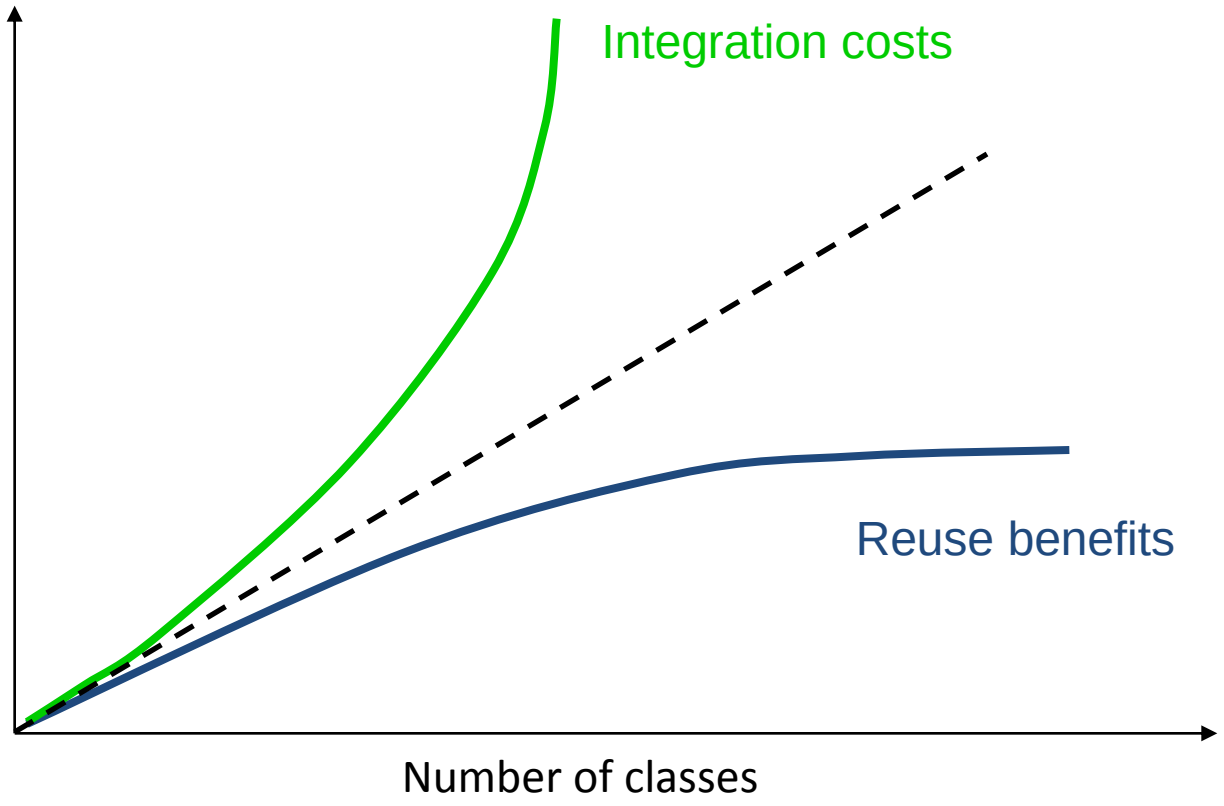


vs.



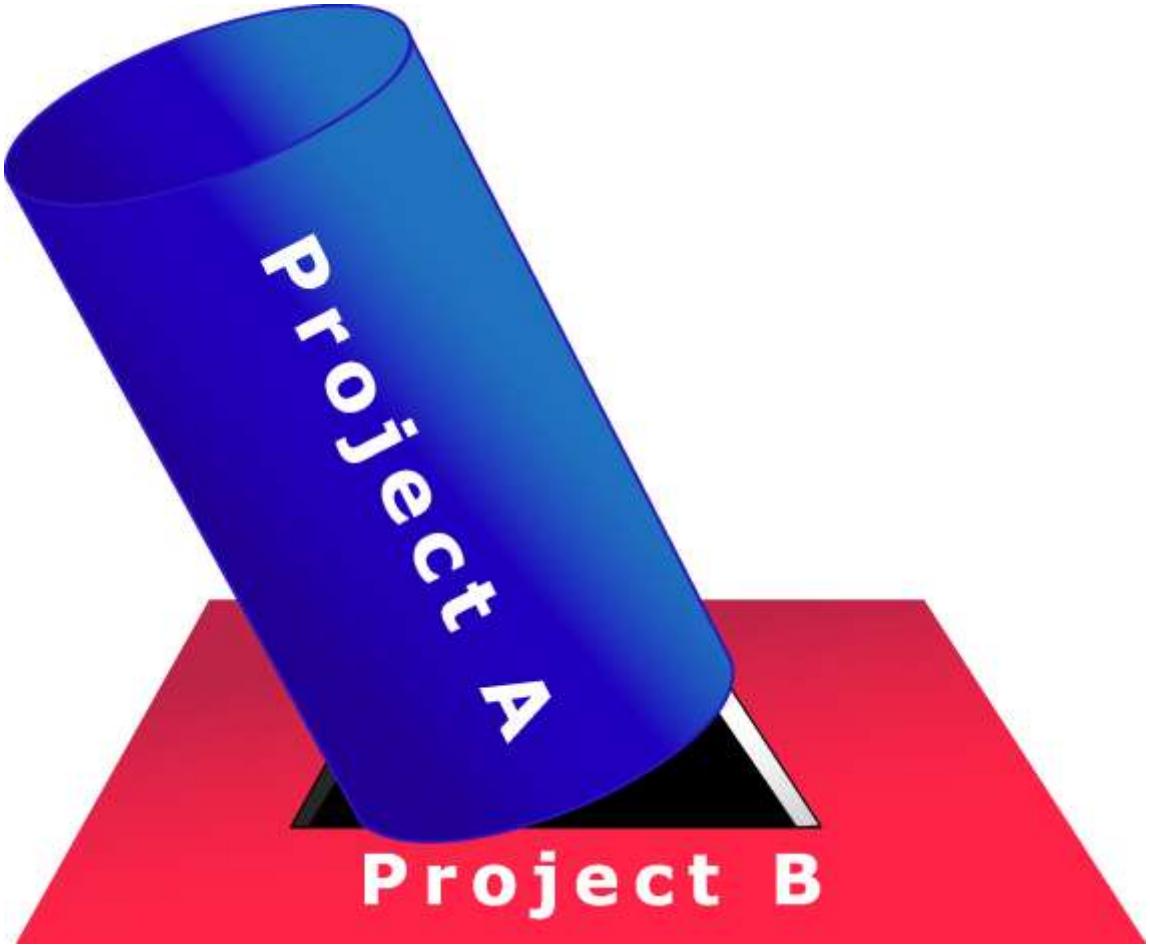
Scale

- The cost of a design rises steeply as system size increases.



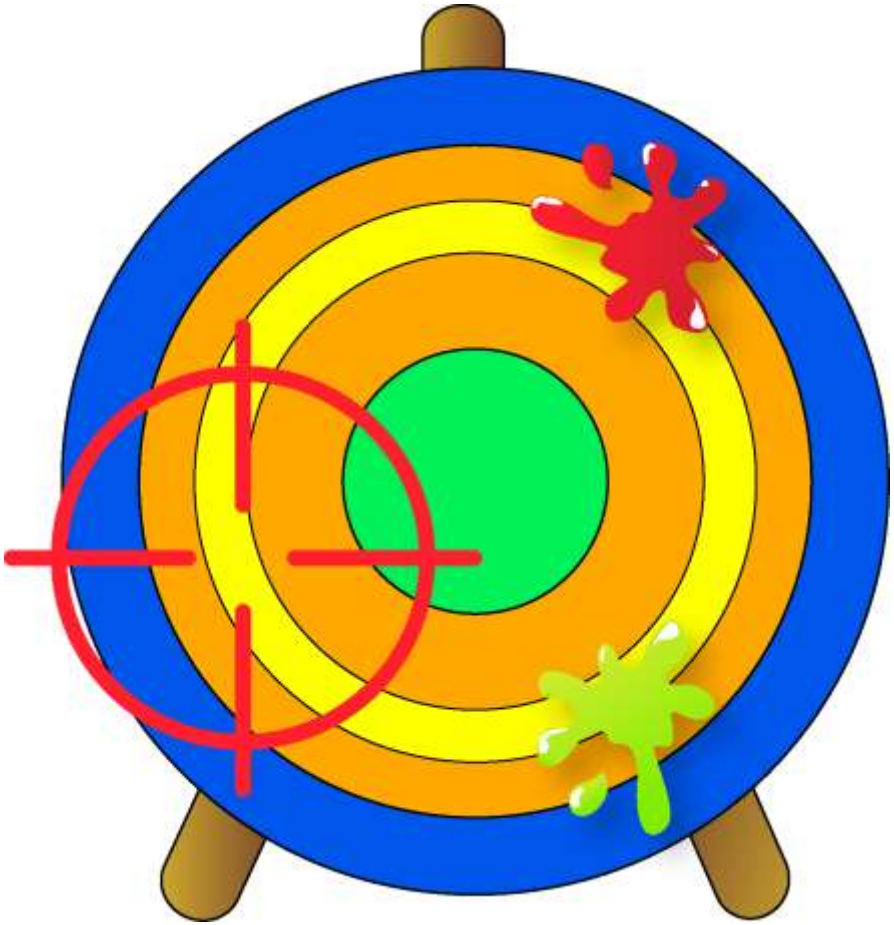
Reuse

- Designs are difficult to reuse.



Retargeting

- Designs are difficult to retarget.



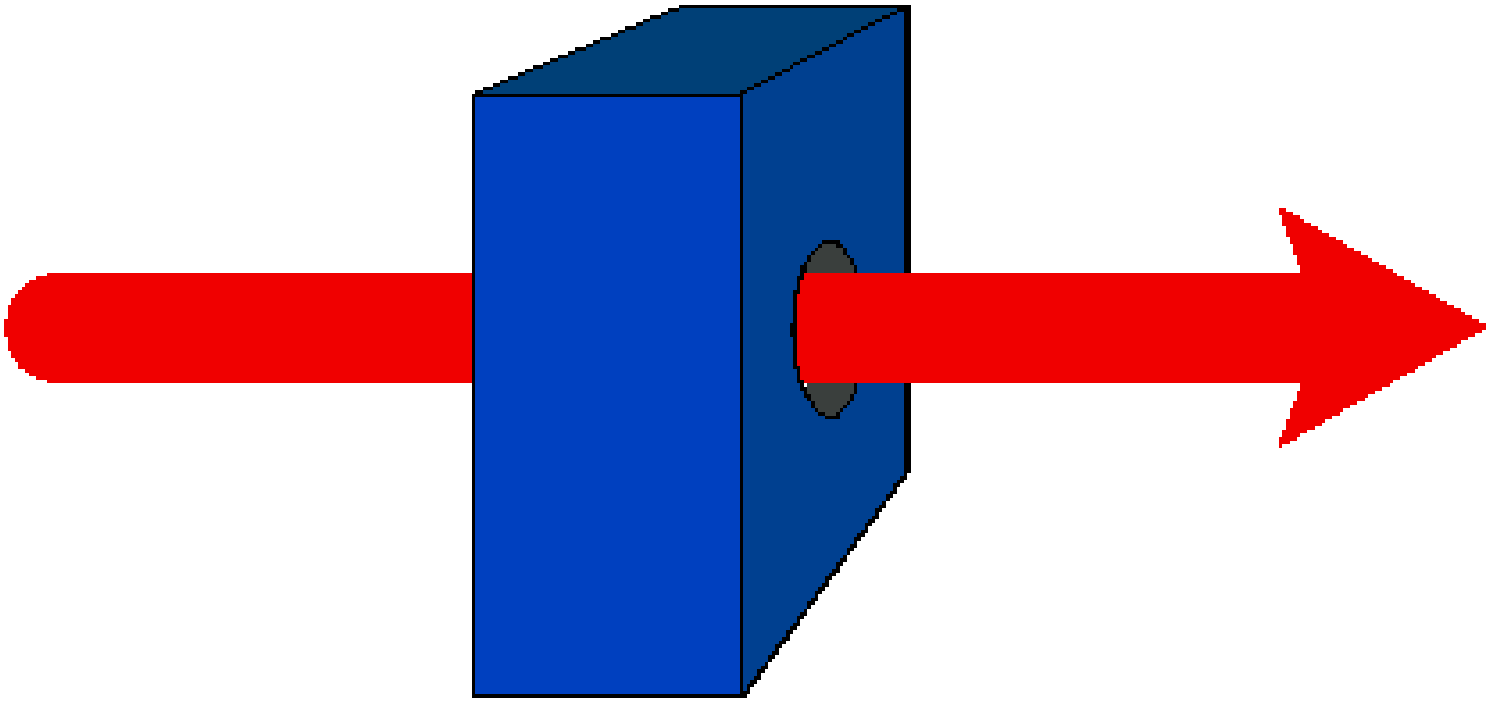
Instrumentation and Diagnosis

- Designs are difficult to instrument and diagnose.



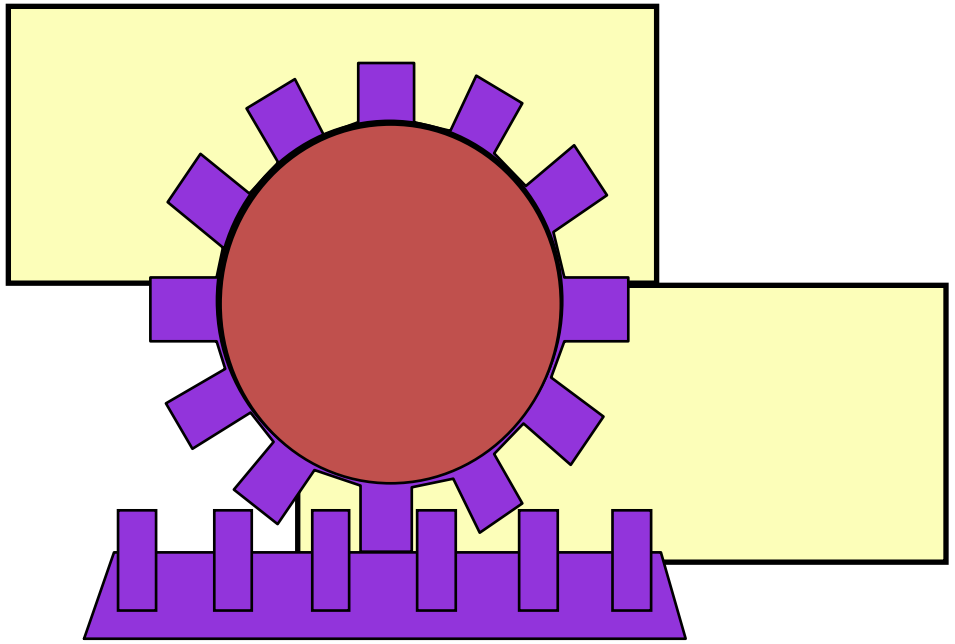
Optimization

- Large designs are difficult to optimize reliably.



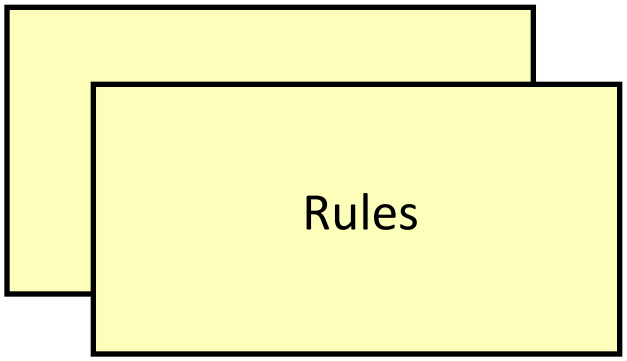
Model Compilers

9

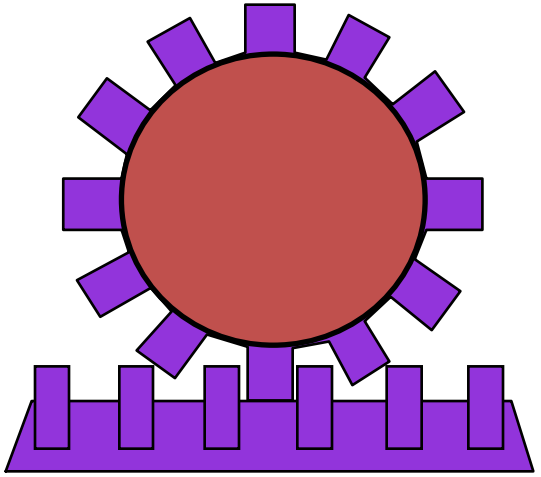


Model Compiler

- A set of rules: Translation rules interpreted by a generator against the repository



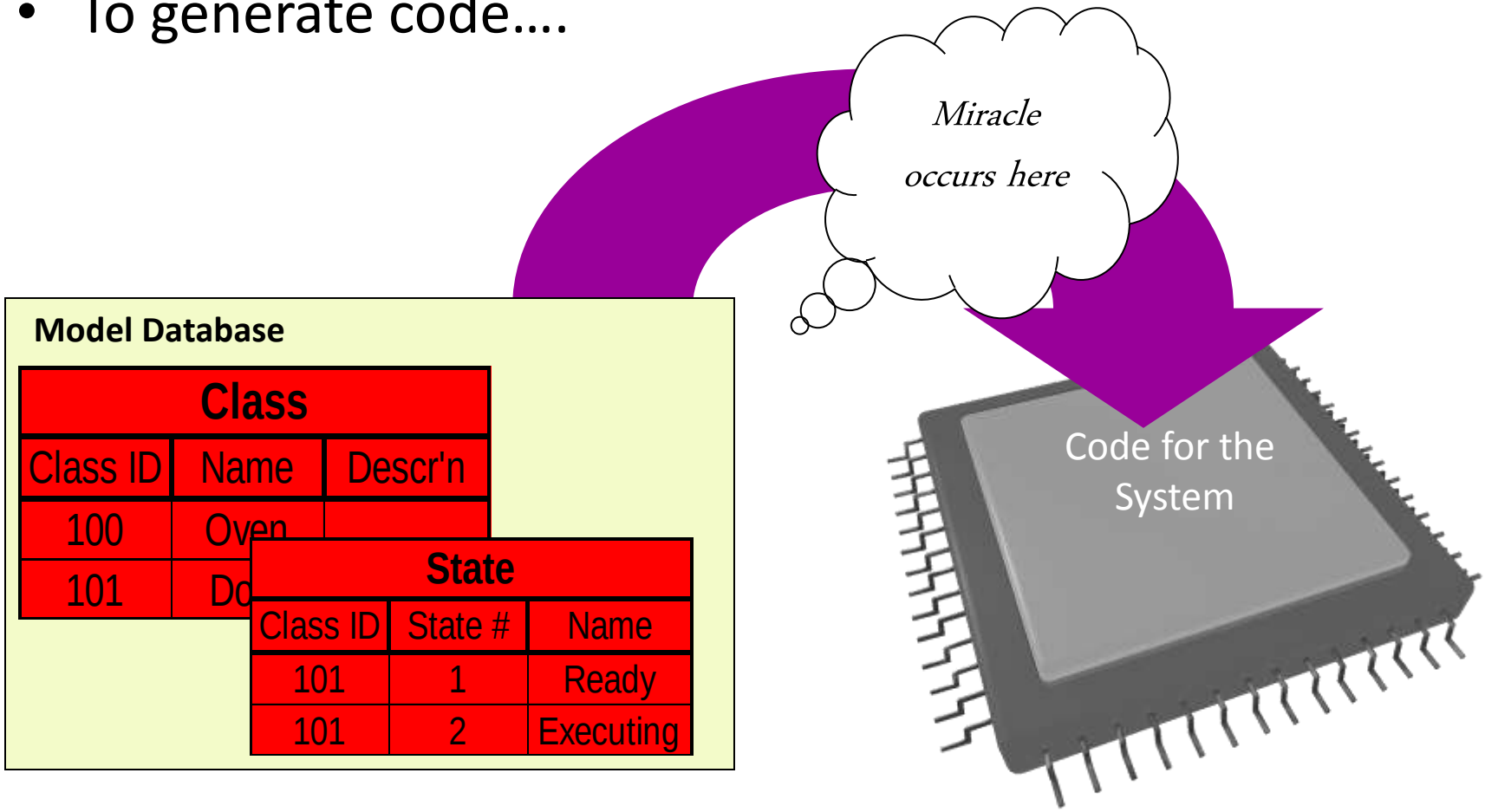
- An execution engine: A limited set of reusable components sufficient to execute Executable UML



Reusable library

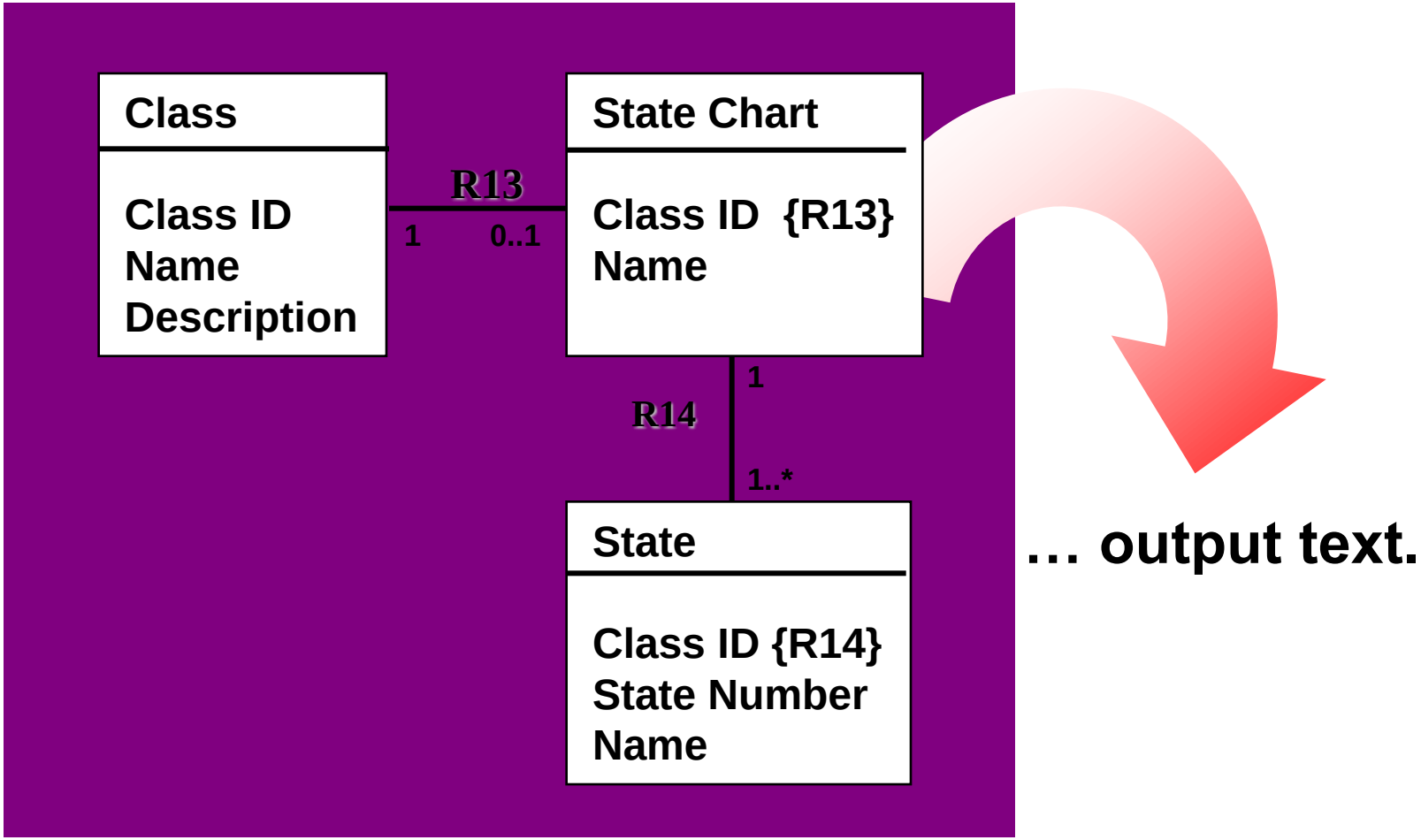
Purpose

- To generate code….



Purpose

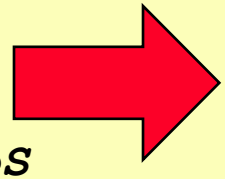
- ….traverse the model database and…



Example

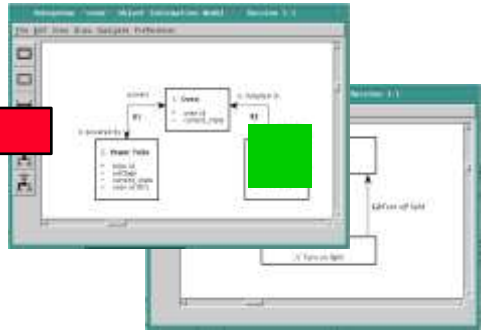
The rule language produces text.

```
.select many stateS related to instances of
        class->[R13]StateChart ->[R14]State
            where (isFinal == False)
public:
  enum states_e
    { NO_STATE = 0 ,
.for each state in stateS
        .if ( not last stateS )
            ${state.Name } ,
        .else
            NUM_STATES = ${state.Name}
        .endif
.endfor
};
```



```
public:
  enum states_e
    {  NO_STATE = 0 ,
       Ready ,
       Executing ,
       NUM_STATES = Complete
    };
```

Building the System



Application Models

Marks

Model Compiler

Rules

Model Database

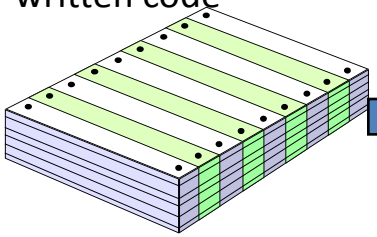
Class		
Class ID	Name	Descr'n
100	Ov	
101	Do	

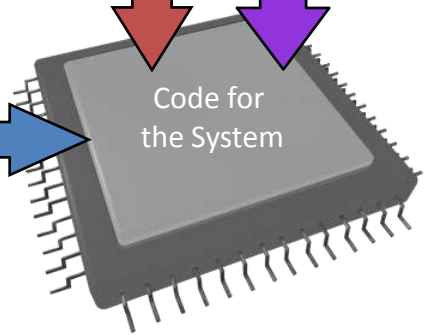
State		
Class ID	State #	Name
101	1	Ready
101	2	Executing

Generator

Reusable Library

- Libraries, Legacy or
- Hand-written code



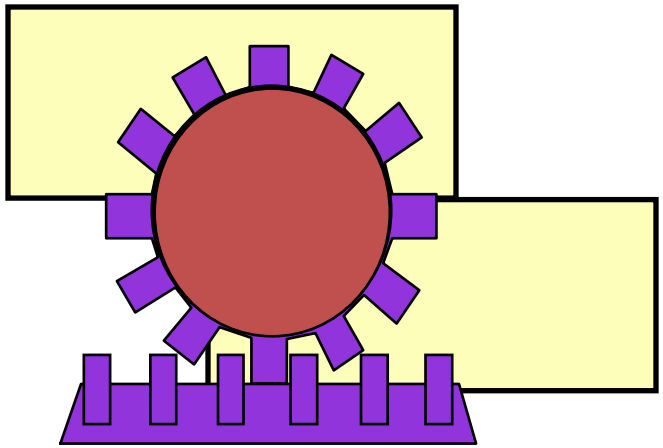


Model Compiler

- An architecture is an *Executable UML model compiler*.

  It translates a system specified in executable UML into the target programming language incorporating decisions made by the architect about:

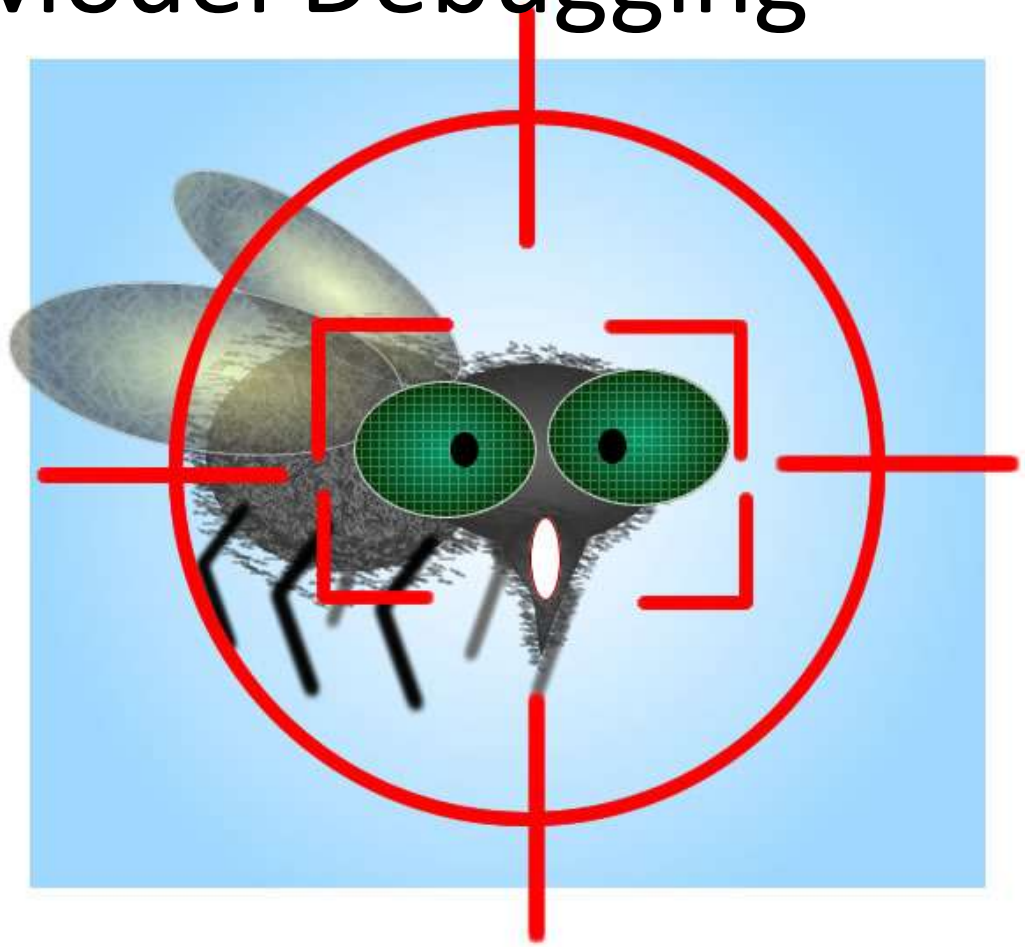
  - data
  - control
  - structures
  - time



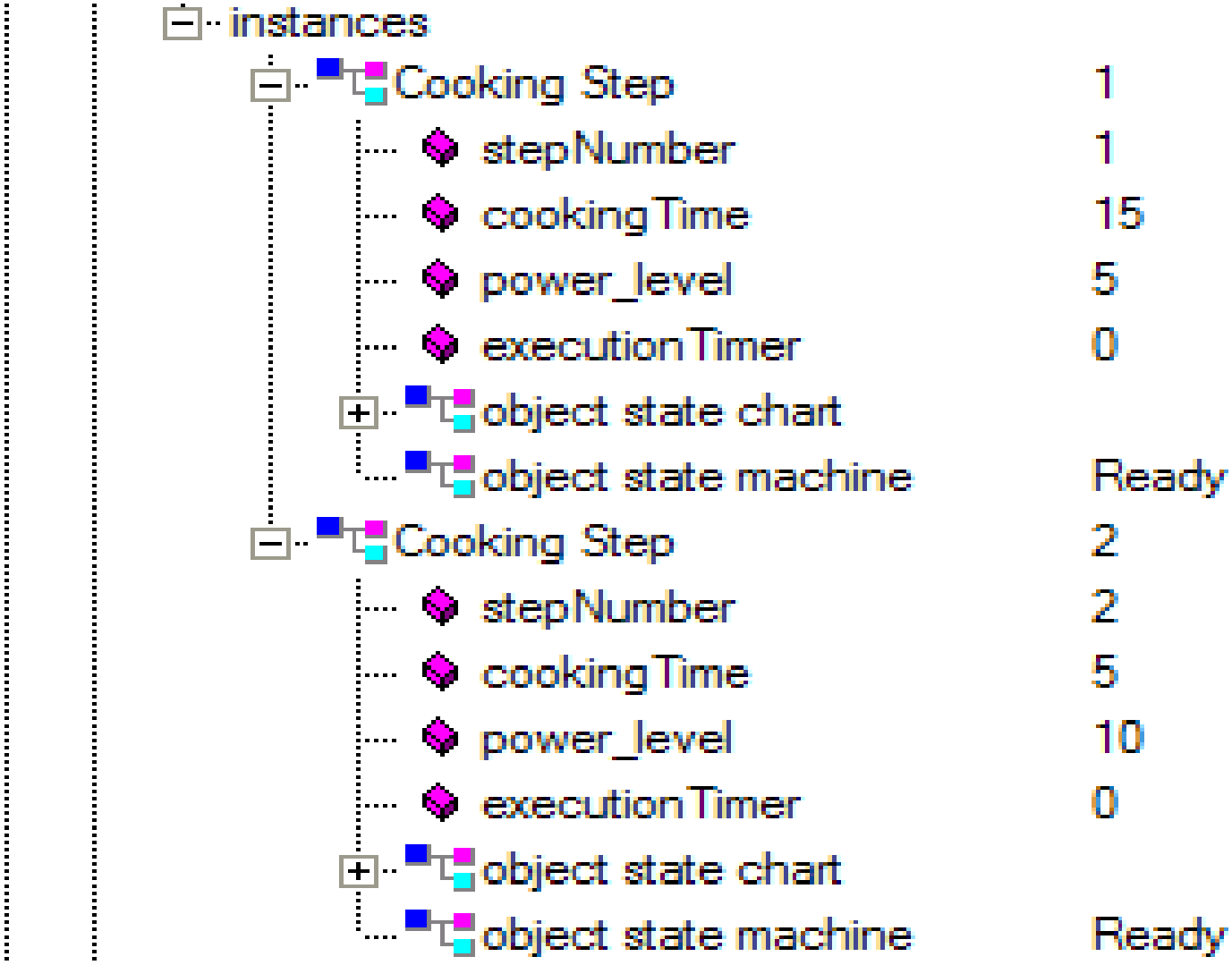
Model compilers, like programming languages compilers, can be bought.

Model Debugging

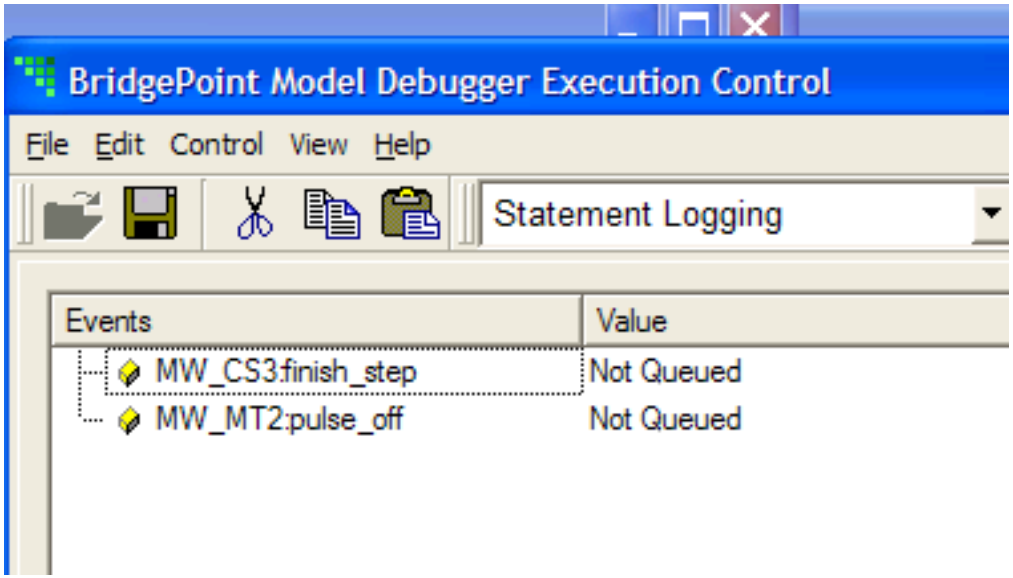
10



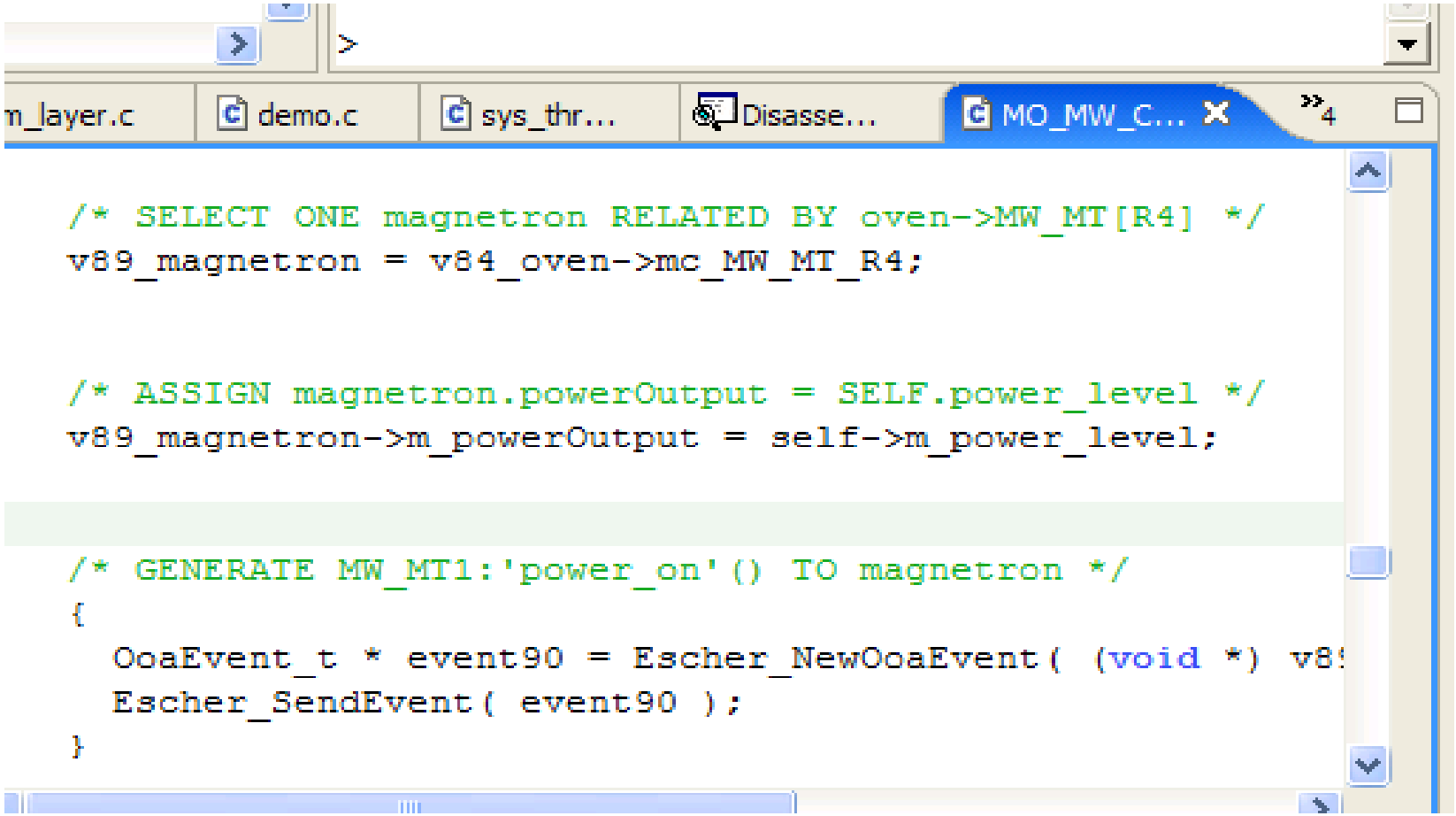
Models Execute on Instances

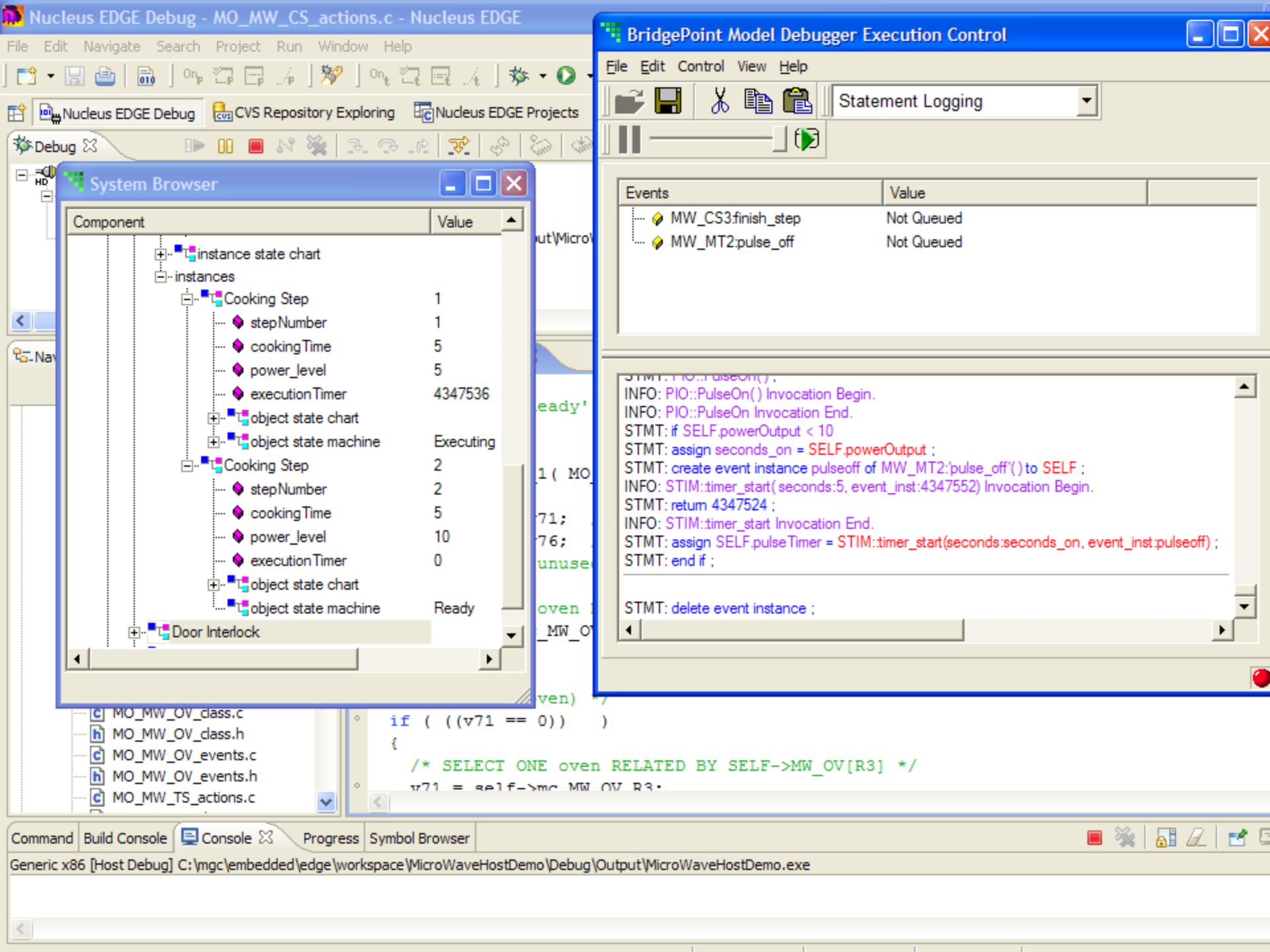


Execution Control



Execution


n_layer.c
demo.c
sys_thr...
Disasse...
MO_MW_C...
4
/* SELECT ONE magnetron RELATED BY oven->MW_MT[R4] */
v89_magnetron = v84_oven->mc_MW_MT_R4;
/* ASSIGN magnetron.powerOutput = SELF.power_level */
v89_magnetron->m_powerOutput = self->m_power_level;
/* GENERATE MW_MT1:'power_on'() TO magnetron */
{
OoaEvent_t * event90 = Escher_NewOoaEvent((void *) v8
Escher_SendEvent(event90);
}


Nucleus EDGE Debug - MO_MW_CS_actions.c - Nucleus EDGE
File Edit Navigate Search Project Run Window Help
Nucleus EDGE Debug
CVS Repository Exploring
Nucleus EDGE Projects
Debug
System Browser
Component
Value
instance state chart
instances
Cooking Step 1
stepNumber 1
cookingTime 5
power_level 5
executionTimer 4347536
object state chart
object state machine Executing
Cooking Step 2
stepNumber 2
cookingTime 5
power_level 10
executionTimer 0
object state chart
object state machine Ready
Door Interlock
BridgePoint Model Debugger Execution Control
File Edit Control View Help
Statement Logging
Events
Value
MW_CS3:finish_step Not Queued
MW_MT2:pulse_off Not Queued
INFO: PIO::PulseOn() Invocation Begin.
INFO: PIO::PulseOn Invocation End.
STMT: if SELF.powerOutput < 10
STMT: assign seconds_on = SELF.powerOutput ;
STMT: create event instance pulseoff of MW_MT2:'pulse_off'() to SELF ;
INFO: STIM::timer_start(seconds:5, event_inst:4347552) Invocation Begin.
STMT: return 4347524 ;
INFO: STIM::timer_start Invocation End.
STMT: assign SELF.pulseTimer = STIM::timer_start(seconds:seconds_on, event_inst:pulseoff) ;
STMT: end if ;
STMT: delete event instance ;
MO_MW_OV_class.c
MO_MW_OV_class.h
MO_MW_OV_events.c
MO_MW_OV_events.h
MO_MW_TS_actions.c
if (((v71 == 0)))
{
/* SELECT ONE oven RELATED BY SELF->MW_OV[R3] */
Command Build Console Console Progress Symbol Browser
Generic x86 [Host Debug] C:\mgc\embedded\edge\workspace\MicroWaveHostDemo\Debug\Output\MicroWaveHostDemo.exe

Coding with UML Models

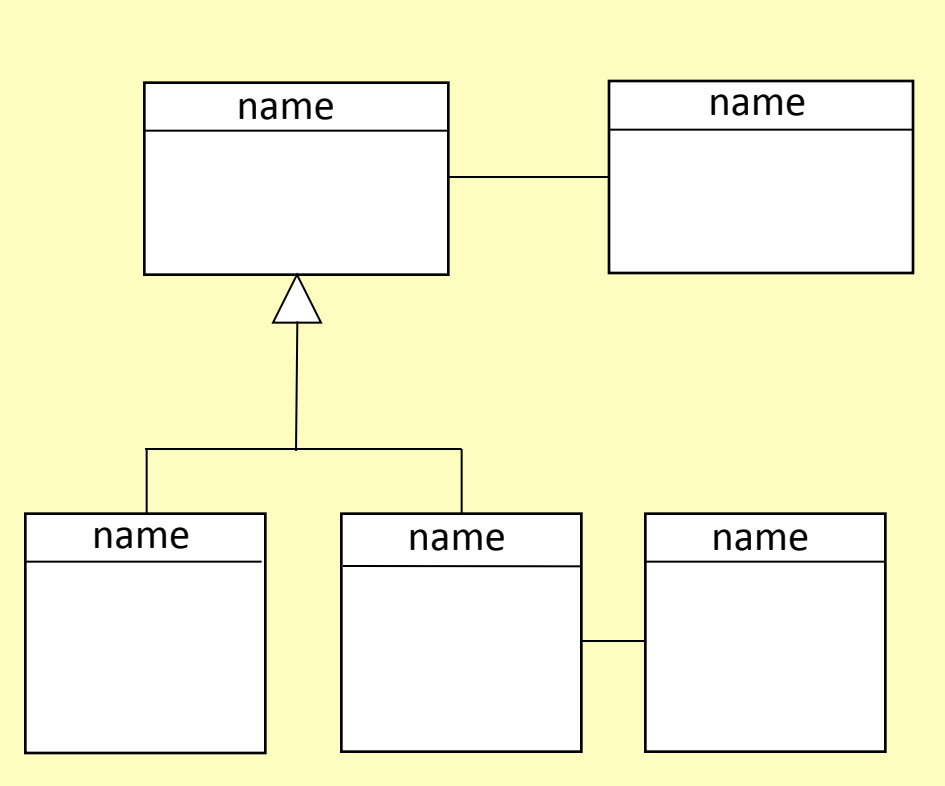
11



```
#include <c8051.h>// include files. This file is available online

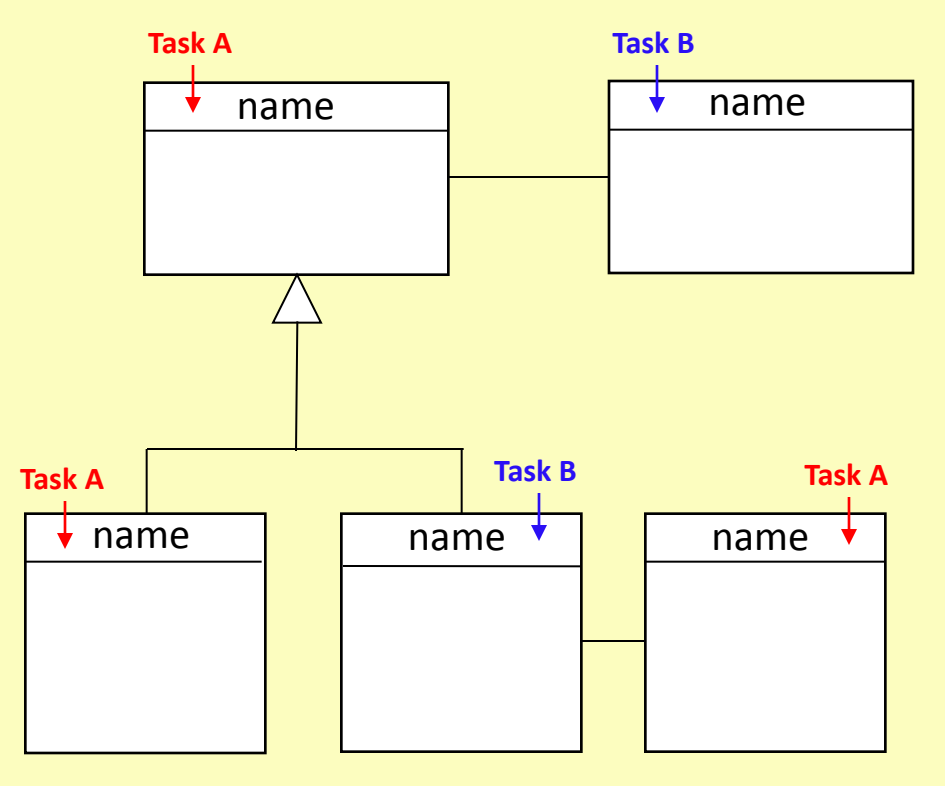

void main(void) // start main function
{
int imax, i; // declare variables before first executable code unsigned char count;
Sys_Init(); // Initialize UART, System clock and crossbar
while(1) // begin infinite loop
{
printf("\n Enter count number\n");
scanf("%d", &imax); // accept keyboard input
printf("\n\n Decimal \t hex \n");
for(i=0;i<=imax;i++)
{
```

Model it!



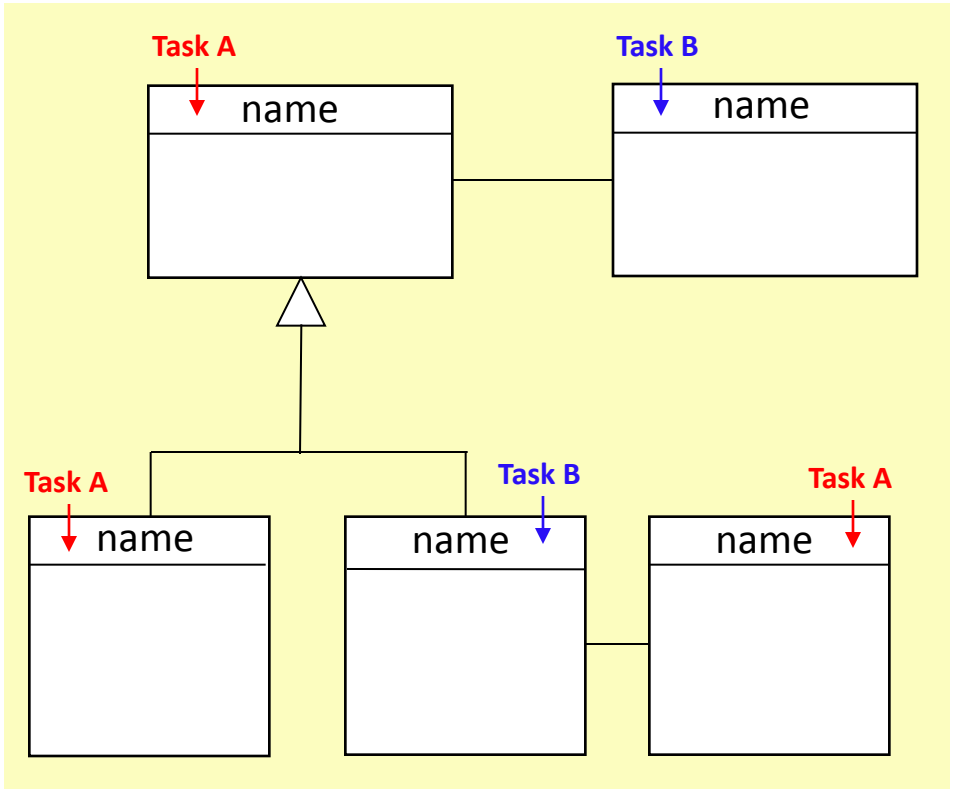
Model the Application

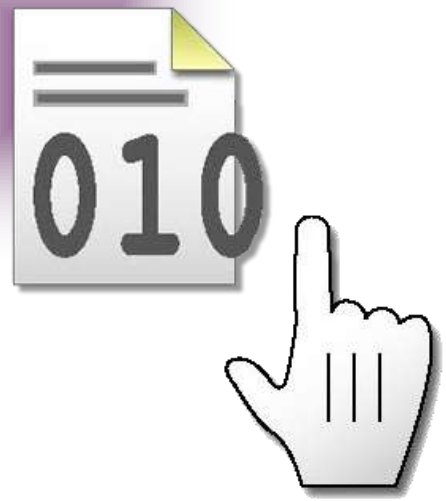
Mark it!



Assign Classes to Tasks

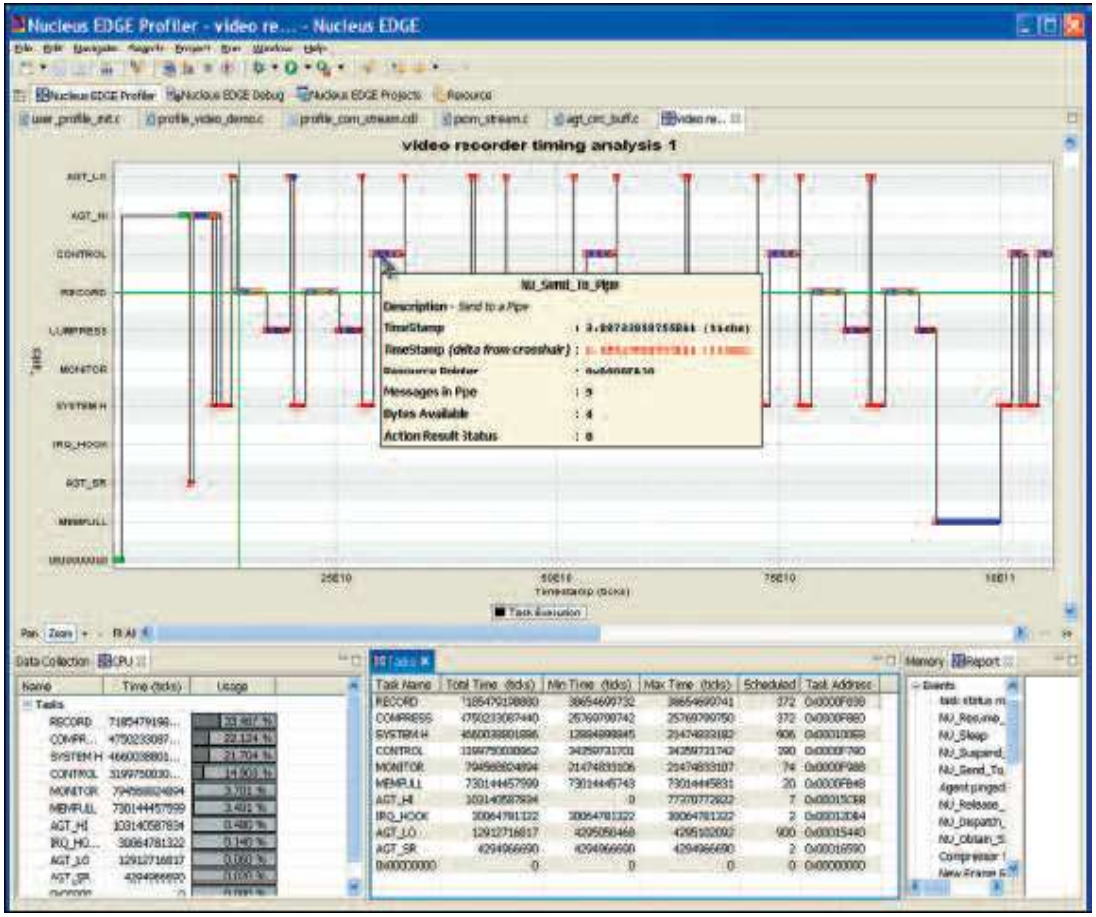
Make it!





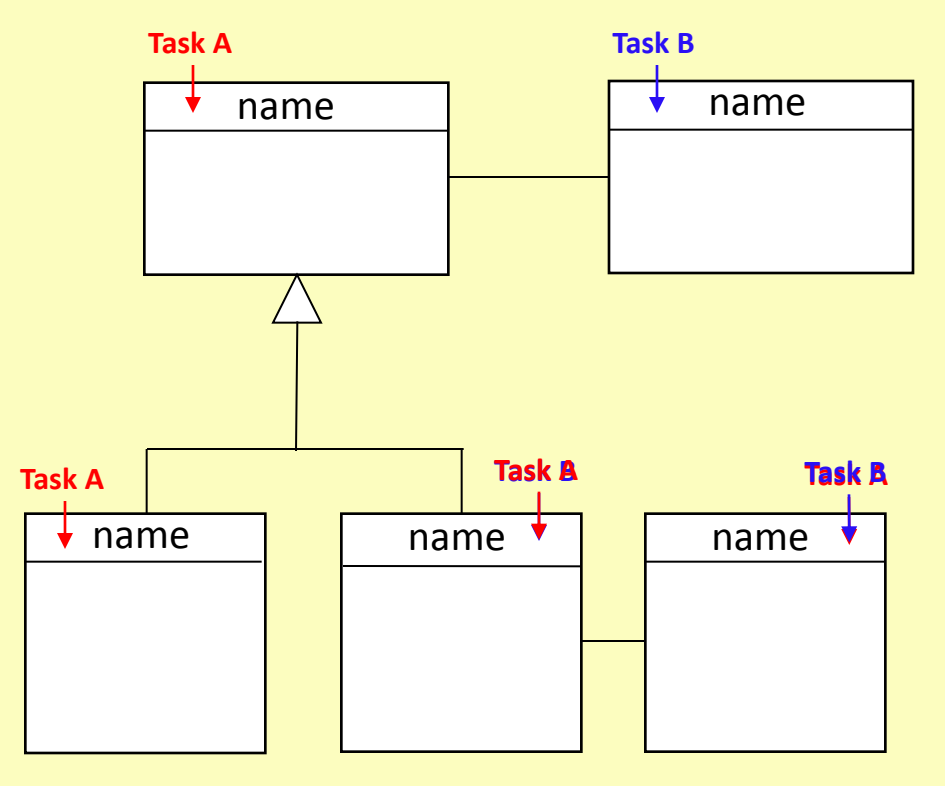


Profile it!



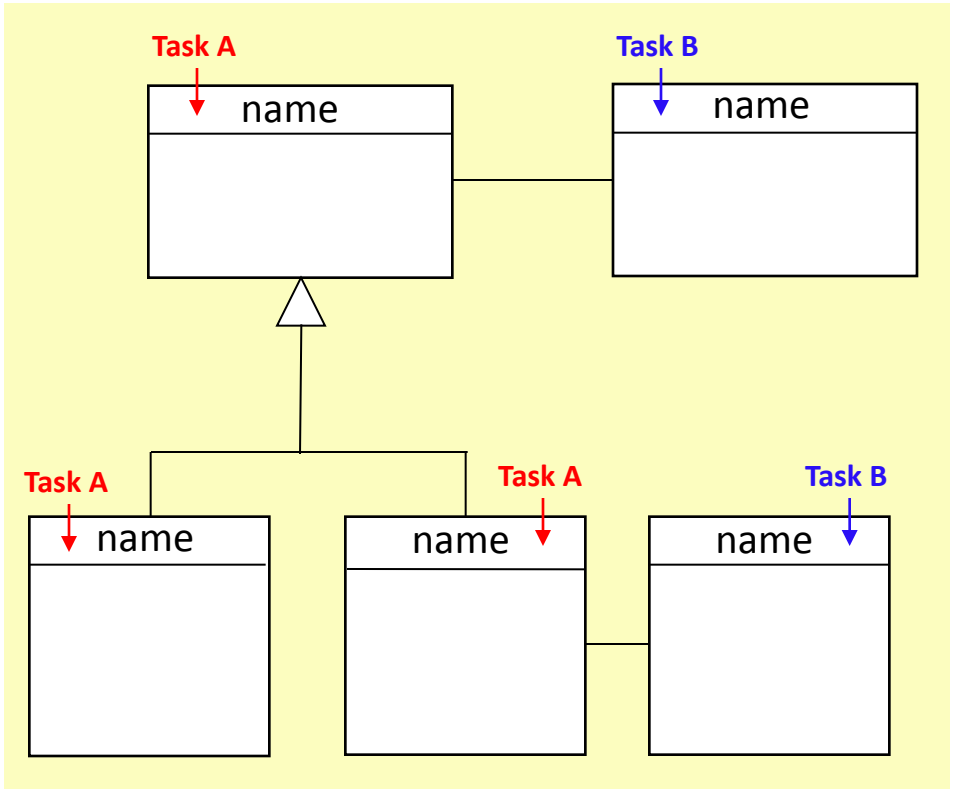
- Analyze the performance
- Not as good as you need?

Re-Mark it!



Reassign Classes to Different Tasks

Make it Again!







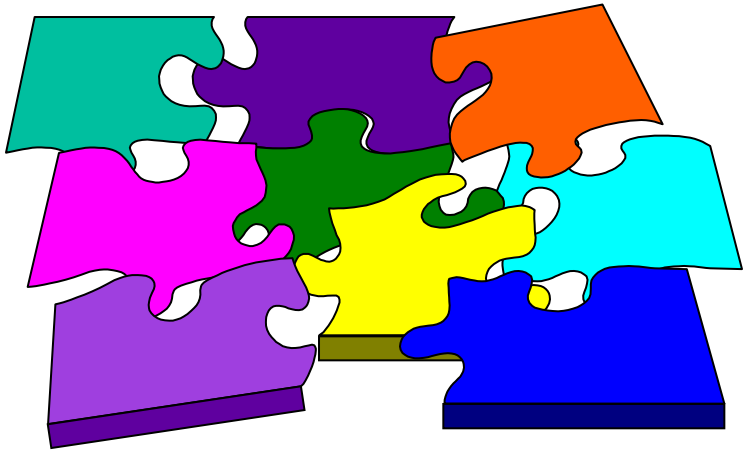
Translative System Generation

- Translative System Generation is a system-construction method based on:
  - separating systems into subject matters (domains)
  - specifying each domain with an executable model
  - translating the models *according to a set of rules*

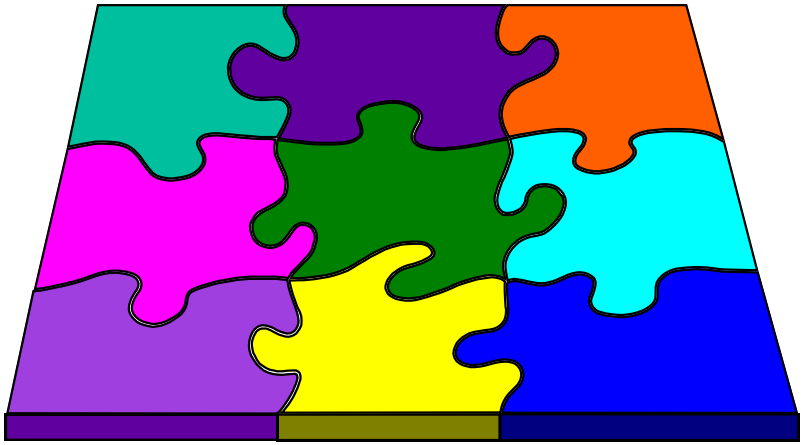


Integration

- (Properly tested) rules *guarantee* the pieces will integrate.

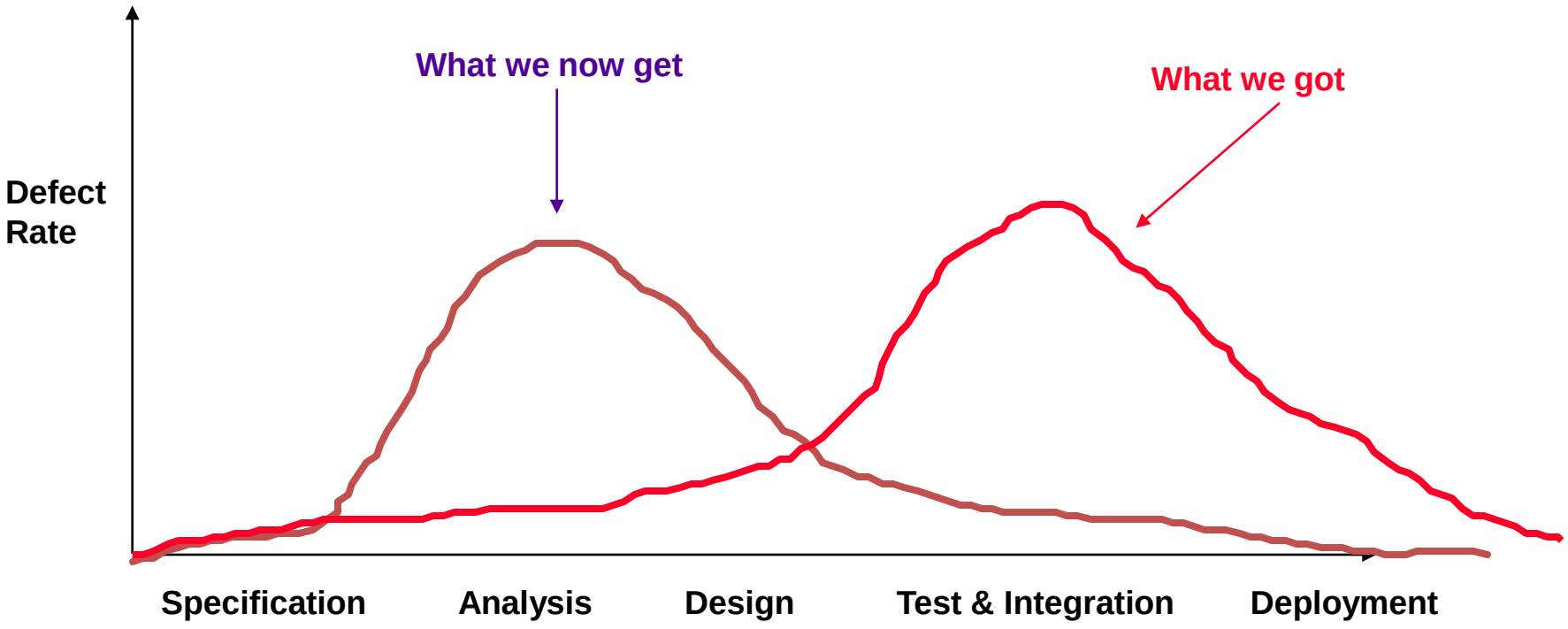


vs.



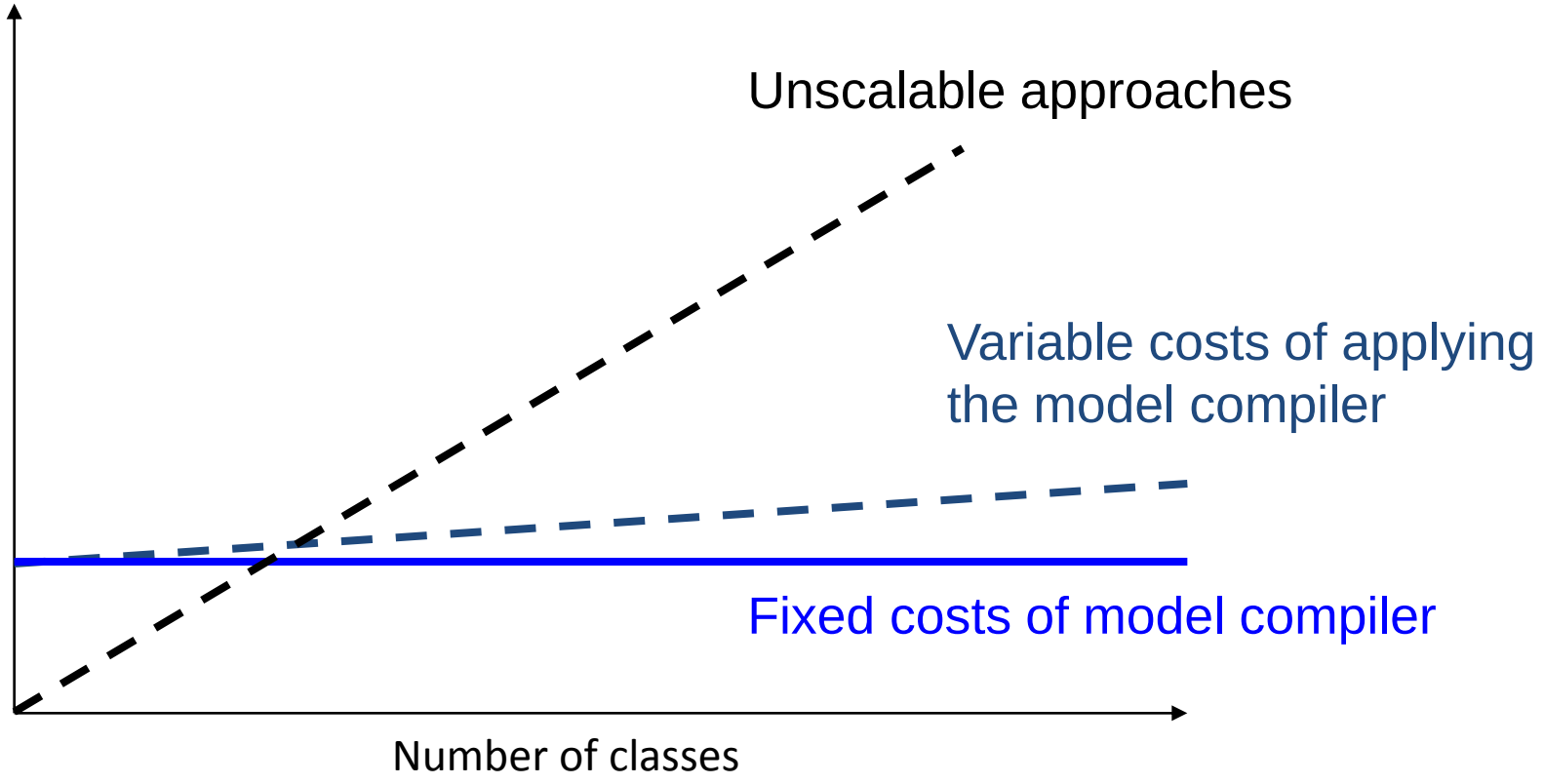
Testing

- Executable models can be tested early



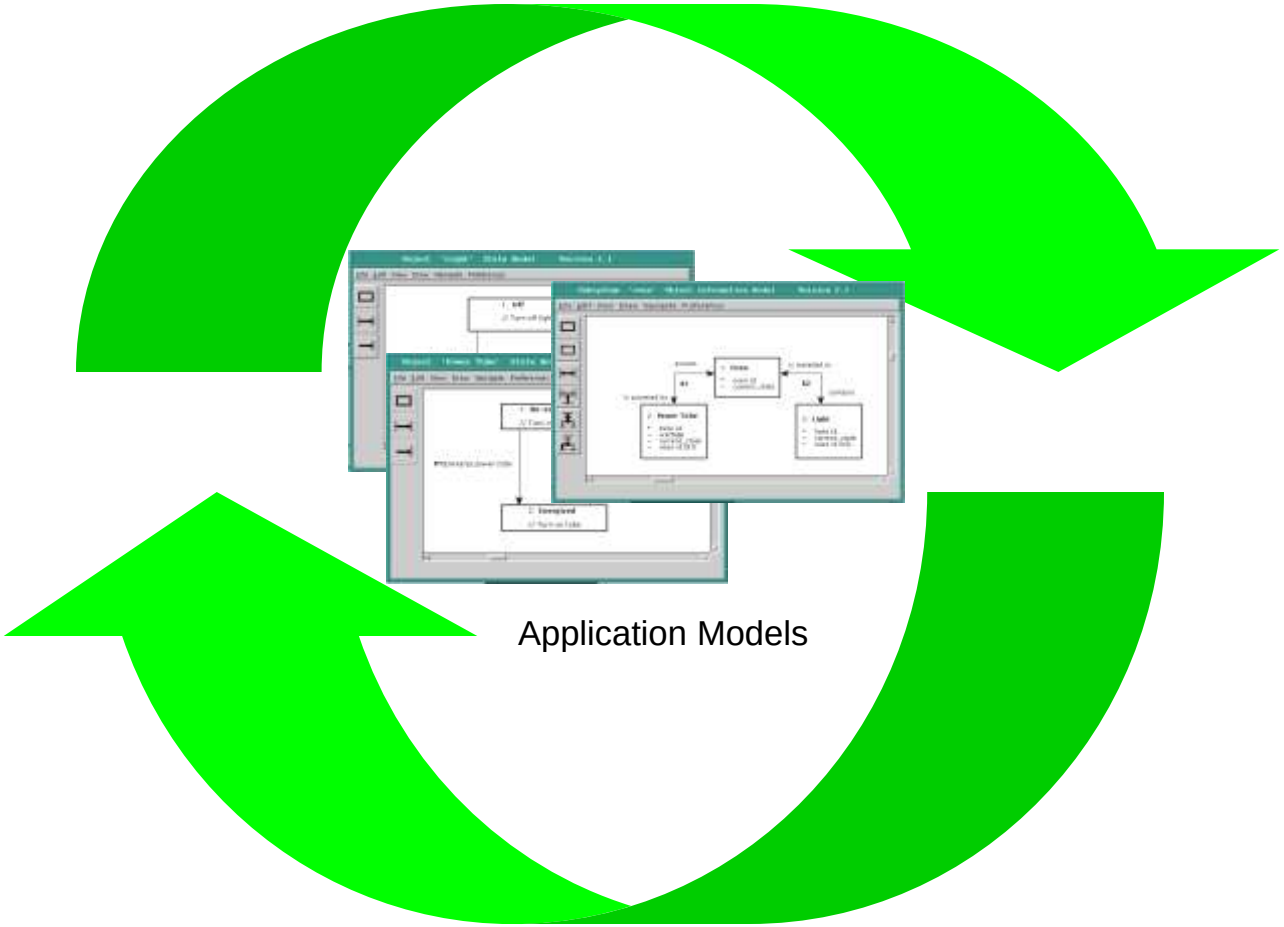
Scale

The cost of a design remains the same for a given model compiler.



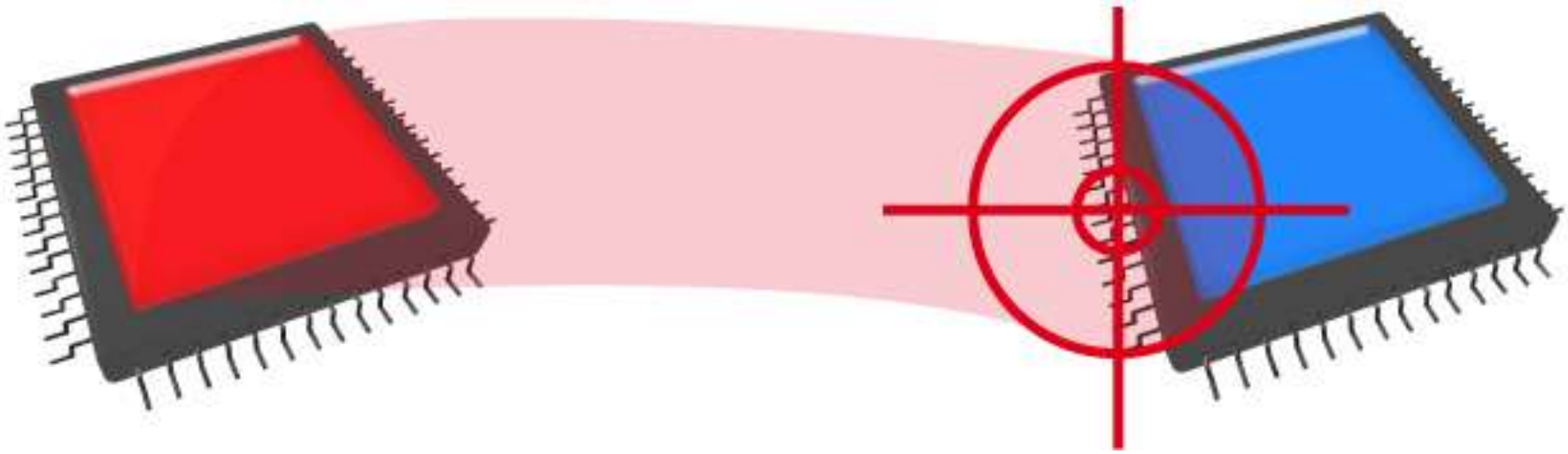
Reuse

Designs can be reused.



Retargeting

Designs can be re-targetted by selecting a different model compiler.



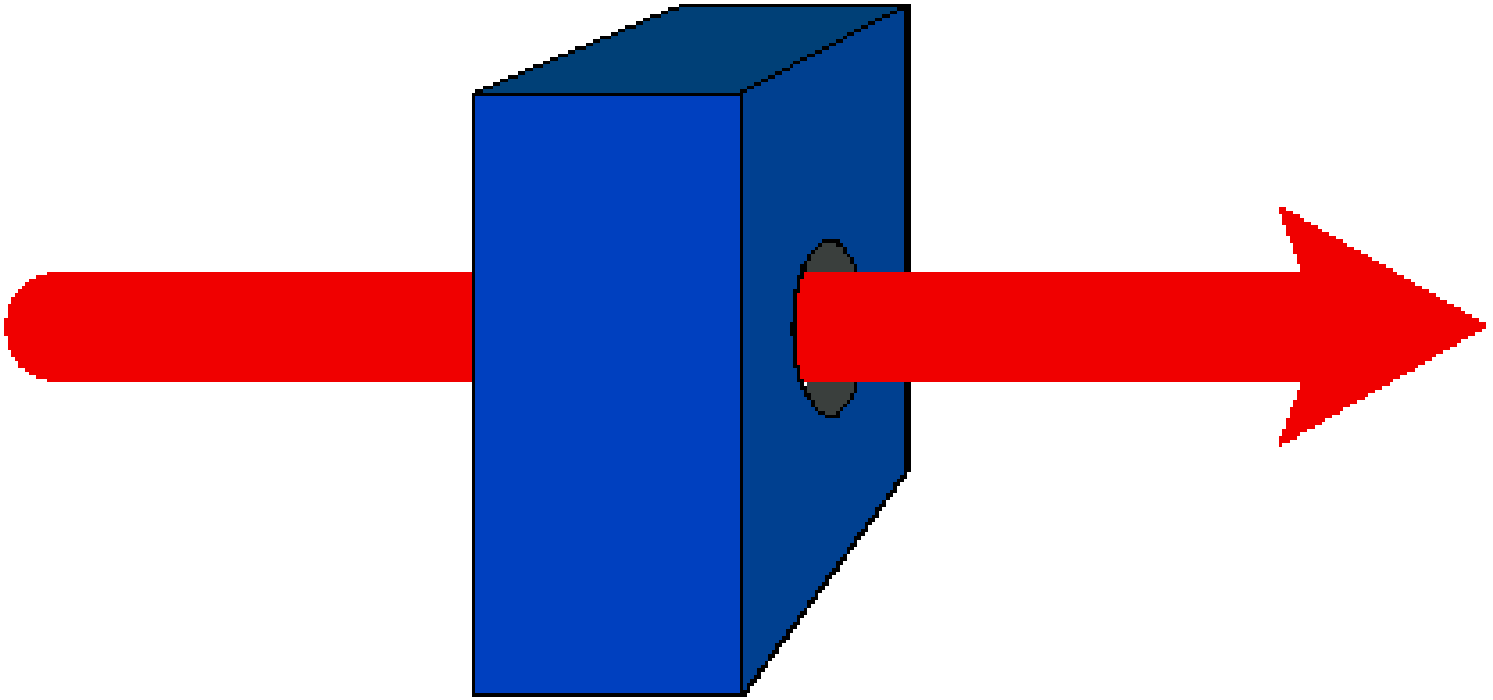
Instrumentation and Diagnosis

Designs are instrumented and diagnosed in one place: the model compiler.



Optimization

Optimization is propagated throughout the system reliably using automated rules.



Skills

- Application experts worry about the application…








- ...while architects worry about the architecture.

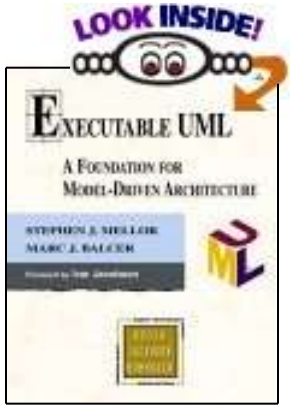
Translative System Generation

Translative system generation meets the challenges of real-time software development by:

- localizing critical software design issues to the software architecture domain
- ensuring that the design decisions are incorporated uniformly and systematically
- providing a framework to modify system performance without affecting system behavior

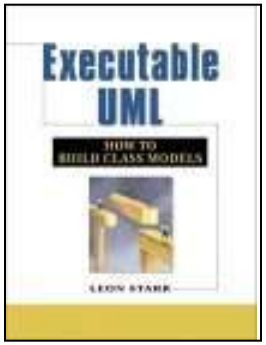
In other words, it s**cales**.

Want to learn more?



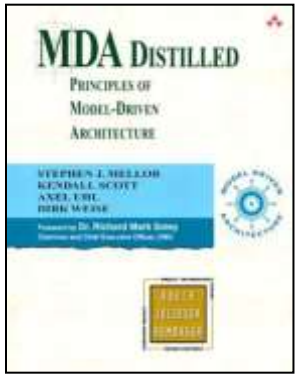
Executable UML: A Foundation for Model-Driven Architecture, Stephen J. Mellor, Marc Balcer

Comprehensive language introduction and reference



Executable UML: How to Build Class Models, Leon Starr

Practical guide for model developers.



MDA Distilled: Principles of Model-Driven Architecture, Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise

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.