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 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

Mapping

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
State Machines

Oven

- NotCooking
  - start
  - stop
  - start

- Checking
  - cook

- Cooking

CookingStep

- setStep(time, level)
- startStep
- interrupt
- finishStep
- finishStep

- Ready
- Executing
- Complete
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**

**Action Language**

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

- Stop_cooking
- Door
- Stop_cooking
- Start_step
- Interrupt_step
- Step_done
- Power_off
- Power_on
- Power_on
Instances

An executable model operates on data about instances.

<table>
<thead>
<tr>
<th>Recipe</th>
<th>Recipe Name {I}</th>
<th>Cooking Time</th>
<th>Cooking Temp</th>
<th>Heating Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nylon</td>
<td>23</td>
<td>200</td>
<td>2.23</td>
<td></td>
</tr>
<tr>
<td>Kevlar</td>
<td>45</td>
<td>250</td>
<td>4.69</td>
<td></td>
</tr>
<tr>
<td>Stuff</td>
<td>67</td>
<td>280</td>
<td>1.82</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Batch</th>
<th>Batch ID {I}</th>
<th>Amount of Batch</th>
<th>Recipe Name {R2}</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>100</td>
<td>Nylon</td>
<td>Filling</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>127</td>
<td>Kevlar</td>
<td>Emptying</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>93</td>
<td>Nylon</td>
<td>Filling</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>123</td>
<td>Stuff</td>
<td>Cooking</td>
</tr>
</tbody>
</table>
Instances

An executable model operates on instances.

Batch 1
- Create Batch
- Filling
- Filled
- Cooking
- Temperature Ramp Complete
- Emptying
- Emptied

Batch 2
- Create Batch
- Filling
- Filled
- Cooking
- Temperature Ramp Complete
- Emptying
- Emptied

Batch 3
- Create Batch
- Filling
- Filled
- Cooking
- Temperature Ramp Complete
- Emptying
- Emptied

Batch 4
- Create Batch
- Filling
- Filled
- Cooking
- Temperature Ramp Complete
- Emptying
- Emptied

Batch
- Batch ID
- Amount of Batch
- Recipe Name {R2}
- Status
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.

Batch 2

- Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
  - Temperature Ramp Complete (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 3

- Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
  - Temperature Ramp Complete (Batch ID)
- Emptying
  - Emptied (Batch ID)

T. Ramp 2

- Do Temp. Ramp (Batch ID, End Time, End Temp)
- Creating
  - Start Controlling (Ramp ID)
  - Controlling
  - Temp. Ramp Complete (Ramp ID)
  - Complete
  - Ended (Ramp ID)
Communication

Before and after.

Batch 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2

T. Ramp 2

Create Batch (Amount of Batch, Recipe Name)
- Filling
  - Filled (Batch ID)
- Cooking
- Temperature Ramp Ended (Batch ID)
- Emptying
  - Emptied (Batch ID)

Batch 2
Verification

The model can be interpreted.
How does time work....?

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

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.
Mappings

Mapping by hand:

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

The knowledge → Mapping → The software
Building the System

Application Models

Model Database

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Name</th>
<th>Descr'n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Oven</td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>Door</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>State ID</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>101</td>
</tr>
<tr>
<td>101</td>
</tr>
</tbody>
</table>

Libraries, Legacy or Hand-written code

Code for the System

Model Compiler

Rules

Reusable Library
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

<table>
<thead>
<tr>
<th>State</th>
<th>Signal</th>
<th>start</th>
<th>cook</th>
<th>step_done</th>
<th>stop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not Cooking</td>
<td>Checking</td>
<td>CANT HAPPEN</td>
<td>CANT HAPPEN</td>
<td>IGNORED</td>
<td></td>
</tr>
<tr>
<td>Checking</td>
<td>Checking</td>
<td>Cooking</td>
<td>CANT HAPPEN</td>
<td>IGNORED</td>
<td></td>
</tr>
<tr>
<td>Cooking</td>
<td>CANT HAPPEN</td>
<td>CANT HAPPEN</td>
<td>Cooking</td>
<td>Not Cooking</td>
<td></td>
</tr>
</tbody>
</table>

- Diagram showing the states and transitions of the Oven.
Activities

An activity may send a signal.

Door closed

What happens next?

- Door closed
- Oven: checking safety → cooking → executing
- Magnetron: pulse ON → pulse OFF

Cooking Step
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
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.
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.

- Oven
- Door
  - isOpen
- CookingStep
  - stepNumber
  - cookingTime
  - powerLevel
- Magnetron
  - powerOutput
  - pulseTimer
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
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

Event-driven Task

High-priority
Periodic Task

Event messages

State bits

Timer

Data copy

Event message

Fast Data
Application Mapping

Event Driven Task

Oven
Magnetron
  powerOutput
  pulseTimer
Cooking Step
  stepNumber
  cookingTime
  powerLevel

Periodic Task

Magnetron
  powerOutput
  pulseTimer

Fast Data
State Bits
Application Mapping

Event Driven Task

Ready → Executing → Complete

Periodic Task

pulseON
pulseOFF

State Bits
Extended Properties

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class ID {R14}</td>
</tr>
<tr>
<td>State Number</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>isFinal</td>
</tr>
<tr>
<td>isPeriodic</td>
</tr>
</tbody>
</table>

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

```plaintext
.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.

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

Practical guide for model developers.