Executable UML

Stephen J. Mellor
Project Technology, Inc.
http://www.projtech.com
Application-Independent Software Architecture
Properties

Separation of application from architecture

Executable UML models

Translation according to rules

Code
What’s in the Architecture?

The architecture comprises:

❖ an execution engine plus
❖ a set of archetypes.

Execution Engine

Archetypes
Archetypes define the rules for translating the application into a particular implementation.

```
.Function Class
Class ${Class.name} : public ActiveInstance {
private:
    .invoke PrivateDataMember( Class )
......
};

.Function PrivateDataMember( inst_ref class )
    .select many PDMs from instances of Attribute related to Class
    .for each PDM in PDMs
        ${PDM.Type} ${PDM.Name};
    .endfor
```
The software architecture is independent of the **semantics** of the application.

This offers:

- early error detection through verification
- reuse of the architecture
- faster performance tuning
- faster integration
- faster, cheaper retargeting
The Software Architecture
Challenges of Real-Time Development

How can we both:

❖ provide required functionality

and

❖ meet real-time performance constraints?

✧ (Re-)organize the software.
The abstract organization of software is called the *software architecture*.

It proclaims and enforces system-wide rules for the organization of:

- data
- control
- structures
- time
The architect prescribes the *storage scheme* for data elements:

- tables or arrays?
- special purpose structures such as trees, linked lists?
- independent?

and *access* to them:

- direct access by name or pointer?
- indirect access through functions that encapsulate the data structure?
Control

The architect prescribes control:

❖ what causes a task to execute?
❖ what causes a task to relinquish control?
❖ what is the next function to execute within a task?
❖ how to coordinate multiple tasks accessing common data to ensure data consistency?
The architect prescribes how *to package* code and data in:
- tasks?
- functions?
- shared data areas?
- classes?

and the *allocation criteria* for allocating parts of the application to these structures.
The software architect prescribes how to provide time-related services:

- absolute time
- relative time
Uniformity

A minimal, uniform set of organization rules:

- reduces cost of understanding, building, and maintaining the software
- decreases integration effort
- leads to smaller, more robust code
Executable Domain Models
Abstract classes based on both:

❖ data, and
❖ behavior

<table>
<thead>
<tr>
<th>Recipe</th>
<th>Batch</th>
<th>Temperature Ramp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipe Name {I}</td>
<td>Batch ID {I}</td>
<td>Ramp ID {I}</td>
</tr>
<tr>
<td>Cooking Time</td>
<td>Amount of Batch</td>
<td>Batch ID {R4}</td>
</tr>
<tr>
<td>Cooking Temp.</td>
<td>Recipe Name {R2}</td>
<td>Start Temperature</td>
</tr>
<tr>
<td>Heating Rate</td>
<td>Status</td>
<td>Start Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>End Temperature</td>
</tr>
<tr>
<td></td>
<td></td>
<td>End Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Status</td>
</tr>
</tbody>
</table>
Lifecycles

Build a lifecycle model for each class.

Lifecycle for Temperature Ramp

- Do Temp. Ramp (Batch ID, End Time, End Temp)
  - Creating
  - Start Controlling (Ramp ID)
  - Controlling
  - Temp. Ramp Complete (Ramp ID)
  - Complete
  - Ended (Ramp ID)
Specify the logic for each state’s action.

Do Temp. Ramp( Batch ID, End Time, End Temp )

Creating
Entry/
BatchID, EndTime, EndTemp
>> TempRamp;
CurrentTime > Self.StartTime;
Self -> [R4] CookingTank.ActualTemp
> Self.StartTemp;
Signal Start Controlling (Ramp ID );
Action Semantics

The action semantics should:

❖ not over-constrain sequencing
  ❖ i.e concurrency & data flow

❖ separate computations from data access
  ✚ to make decisions about data access without affecting algorithm specification

❖ manipulate only UML elements
  ✚ to restrict the generality and so make a specification language

Creating

Entry/
BatchID, EndTime, EndTemp
  >> TempRamp;
CurrentTime > Self.StartTime;
Signal Start Controlling (Ramp ID );
An Executable Model

**Lifecycle for Temperature Ramp**

- **Batch**
  - Batch ID {I}
  - Amount of Batch
  - Recipe Name {R2}
  - Status

- **Temperature Ramp**
  - Ramp ID {I}
  - Batch ID {R4}
  - Start Temperature
  - Start Time
  - End Temperature
  - End Time
  - Status

**Action for Creating**

- **Creating**
  - Entry/
    - BatchID, EndTime, EndTemp
    - CurrentTime > Self.StartTime;
    - Signal Start Controlling (Ramp ID);

- **Controlling**
  - Temp. Ramp Complete

- **Complete**
  - Ended( Ramp ID )

- **Do Temp. Ramp( Batch ID, End Time, End Temp )**

- **Temp. Ramp Complete( Ramp ID )**

- **Ended( Ramp ID )**
An executable model operates on instances.
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,
- other instances as they execute
- timers

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

Batch 2
Model Database

Each schema has a corresponding database for instances.
Model Capture
Model Repository

Capture the model in a model repository.

What is the structure of the repository?
A *meta-model* defines the structure of the repository.
A *meta-model* defines the structure of the repository.
A meta-model defines the structure of the repository.
Meta-Model Instances

Just like an application model, the meta-model has instances.

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Name</th>
<th>Descr'n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Recipe</td>
<td>.....</td>
</tr>
<tr>
<td>101</td>
<td>Batch</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>Temp Ramp</td>
<td>.....</td>
</tr>
</tbody>
</table>

Create Batch( Amount of Batch, Recipe Name)

- Filling
- Cooking
- Emptying

Filled( Batch ID )

Temperature Ramp Complete( Batch ID )

Emptied( Batch ID )

<table>
<thead>
<tr>
<th>Class ID</th>
<th>State #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1</td>
<td>Filling</td>
</tr>
<tr>
<td>101</td>
<td>2</td>
<td>Cooking</td>
</tr>
<tr>
<td>101</td>
<td>3</td>
<td>Emptying</td>
</tr>
<tr>
<td>102</td>
<td>1</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>2</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>
Archetype
Language
Purpose

To generate code....

<table>
<thead>
<tr>
<th>State</th>
<th>Class ID</th>
<th>State #</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>101</td>
<td>1</td>
<td>Filling</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>2</td>
<td>Cooking</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>102</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Class ID</th>
<th>Name</th>
<th>Descr'n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100</td>
<td>Recipe</td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td>101</td>
<td>Batch</td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td>102</td>
<td>Temp</td>
<td>.....</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ramp</td>
<td></td>
</tr>
</tbody>
</table>
….traverse the repository and...

1

… output text.
The archetype language produces text.

```java
..select many stateS related to instances of class->State->StateChart
   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 ,
     Filling ,
     Cooking ,
     NUM_STATES = Emptying
   };
```
To generate text:

The quick brown fox jumped over the lazy dog.
To select any instance from the repository:

.select any class from instances of **Class**

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Name</th>
<th>Descr'n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Recipe</td>
<td>.....</td>
</tr>
<tr>
<td>101</td>
<td>Batch</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>Temp</td>
<td>Ramp</td>
</tr>
</tbody>
</table>
To access attributes of the selected instance....

$\{\text{class.Name}\}$

<table>
<thead>
<tr>
<th>Class ID</th>
<th>Name</th>
<th>Descr'n</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Recipe</td>
<td>.....</td>
</tr>
<tr>
<td>101</td>
<td>Batch</td>
<td>.....</td>
</tr>
<tr>
<td>102</td>
<td>Temp</td>
<td>Ramp</td>
</tr>
</tbody>
</table>
Association Traversal

To traverse an association....

Not just any one--
the one that’s associated

.select one StateChart related to instances of class->StateChart

State Chart
Class ID {I, R13}
Name

Class
Class ID {I}
Name
Description

Create Batch( Amount of Batch, Recipe Name)
  Filling
  Filled( Batch ID )
  Cooking
  Temp. Ramp Complete( Batch ID )
  Emptying
  Emptied( Batch ID )
To select an arbitrary one….

.select any state related to instances of StateChart->State

Or...

.select any state related to instances of Class->StateChart->State
To qualify the selection...

.select any *state* related to instances of *StateChart*->*State*
where *(isFinal == False)*
To select many instances:

```
$select many states related to instances of Class->StateChart ->State where (isFinal==False)
```

States =

- Filling
- Cooking
- Emptying
Iteration

To iterate over instances...

.select many states related to instances of Class->StateChart -> State where (isFinal == False)
.for each state in states
    ${state.Name} ,
.endfor

Filling,
Cooking,
Emptying,
We may combine these techniques....

.select many *stateS* related to instances of
   *class*->*StateChart*->*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
   }
An archetype language gives access to

- the semantics of the application
- as stored in the repository.

We may use the archetype language to generate code.
A Direct Translation
Application Classes

Each application class becomes an implementation class.

.select many classES from instances of class
.for each class in classES
class ${class.Name} : public ActiveInstance {
    .invoke addPDMDecl( inst_ref class)
    ...
};
.endfor
Application Attributes

Each attribute becomes a private data member:

```pseudo
.function addPDMDecl( inst_ref class )
private:
    .select many attr$ related to class->Attribute
    .for each attr in attr$
        ${attr.Type} {attr.Name} ;
    .endfor
.end function
```
To declare a state chart: (i.e. all the actions in the state chart)

.function addProtectedActions( inst_ref class )
.select one statechart related by class->StateChart
protected:
// state action member functions
.select many states related by statechart->State
.for each state in states
  .invoke addActionFunctionDecl( inst_ref state )
.endfor
.end function
To generate the state action declaration:

```assembly
.function addActionFunctionDecl( inst_ref state )
// State action: state.Name
static void sAsyncAction${state.Name}(
    stda_eventMsg_c *eventPtr, int nextState);
    void ${state.Name}(stda_eventMsg_c *p_evt );
void asyncAction${state.Name }();
.endfor
```
To define the state action function....

...traverse the repository in the same manner.
An Indirect Architecture
Where Have All The Cycles Gone?

Consider 500 Temperature Ramp instances all in the Controlling state, controlling temperature 100 times a second.

Using the direct architecture this means:
- $500 \times 100 = 50,000$ state transitions per second, or
- 20 microseconds per transition including the actions

The direct architecture won’t work on a single (cheap) processor.
Where Have All The Cycles Gone?

We could buy several cheaper processors, each controlling different temperature ramps…….

Or we could:

❖ build a separate task that handles all periodic activity at a single rate
❖ execute the periodic task 100 times/second using a timer
❖ leave the remaining “event-driven” logic in another task
Description of Architecture

This introduces new issues:

❖ how to indicate transition into the periodic state (use one bit per instance), and
❖ how to indicate transition out of the periodic state

❖ the periodic task has to be able to execute when “it’s time,” so it needs higher priority----
❖ which raises issues of inconsistent data, so we should duplicate data needed for the control loop----
❖ and copy it over by the periodic task when required
Description of Architecture

Event-driven Task

Periodic Task

Event messages

Instance bits

Data Copy

Event message

Timer
## Application Mapping

### Event Driven Task

<table>
<thead>
<tr>
<th><strong>Recipe</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Recipe Name ({I})</td>
</tr>
<tr>
<td>Cooking Time</td>
</tr>
<tr>
<td>Cooking Temperature</td>
</tr>
<tr>
<td>Heating Rate</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Batch</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Batch ID ({I})</td>
</tr>
<tr>
<td>Amount of Batch</td>
</tr>
<tr>
<td>Recipe Name ({R2})</td>
</tr>
<tr>
<td>Status</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Temperature Ramp</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ramp ID ({I})</td>
</tr>
<tr>
<td>Batch ID ({R4})</td>
</tr>
<tr>
<td>Start Temperature</td>
</tr>
<tr>
<td>Start Time</td>
</tr>
<tr>
<td>End Temperature</td>
</tr>
<tr>
<td>End Time</td>
</tr>
<tr>
<td>Status</td>
</tr>
</tbody>
</table>

### Periodic Task

<table>
<thead>
<tr>
<th><strong>Temperature Ramp</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ramp ID ({I})</td>
</tr>
<tr>
<td>Start Temperature</td>
</tr>
<tr>
<td>Start Time</td>
</tr>
<tr>
<td>End Temperature</td>
</tr>
<tr>
<td>End Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Ramp Id Bits</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
Application Mapping

Event Driven Task

- Do Temp. Ramp( ... )
- Creating
- Start Controlling ( Ramp ID )
- Controlling
- Temp. Ramp Complete( Ramp ID )
- Complete
- Ended( Ramp ID )

Periodic Task

- Controlling
- Ramp Id Bits
To make certain distinctions, we need to tag elements of the meta-model.

```
.function addPeriodicStateAction
...
RampIDbits[insNumber].activateActions();
```

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class ID {I, R14}</td>
</tr>
<tr>
<td>State Number {I}</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>isFinal</td>
</tr>
<tr>
<td>isPeriodic</td>
</tr>
</tbody>
</table>
System Construction
Compile the source code and include initialization data files (if any) to generate the deliverable production code.
Model-Based Maintenance

To address performance-based issues:
❖ modify the architecture models, and
❖ and regenerate the system.

明ら Do not modify the generated code directly.
Model-Based Maintenance

To address application behavior issues,
❖ modify the relevant application model, and
❖ regenerate the system.

Do not modify the generated code directly.
Model-Based Maintenance

For subsequent product enhancements,
❖ modify or replace the domain in question, and
❖ regenerate the system.

Do not modify the generated code directly.
An architecture is an *Executable UML model compiler*. It translates a system specified in X-UML into the target programming language incorporating decisions made by the architect about:

❖ data,
❖ control,
❖ structures, and
❖ time.

Architectures, like programming language compilers, can be bought.
System Design
Executable UML

Executable UML relies on:

❖ separating systems into subject matters (domains)
❖ specifying each domain with an executable model
❖ translating the models

The method may employ UML or any other notation that has a defined semantics.
Subject Matter Separation

The application and architecture are separate subject matters.
Executable UML Models
Executable UML Models

Executable models can be simulated before coding begins.

Lifecycle for Temperature Ramp

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

Temperature Ramp
- Ramp ID {I}
- Batch ID {R4}
- Status
- Start Temperature
- Start Time
- End Temperature
- End Time
- Status

Action for Creating

- Entry/
  - BatchID, EndTime, EndTemp
  - TempRamp;
  - CurrentTime > Self.StartTime;
  - Signal Start Controlling (Ramp ID );
Translation is the act of combining the subject matters.

Translation according to rules

Code
Translation

Translating the application domain models generates:

- highly systematic
- uniform
- reproducible
- *understandable application code*

and minimizes:

- *coding and code inspection effort*
- *coding errors*
- *component integration issues*
Executable UML meets the challenges of 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
Translating Ideas Into Products.

Makers of BridgePoint ® and DesignPoint ®
Modeling Tools

Stephen J. Mellor
Project Technology, Inc.
http://www.projtech.com