System Design: Architectures and Archetypes

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

- Executable UML models
- Separation of application from architecture
- Translation according to rules
- Code for the System
What’s in the Architecture?

The architecture comprises:

- an execution engine plus
- a set of 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:
- reuse of the architecture
- faster performance tuning
- faster integration
- faster, cheaper retargeting
- centralized diagnostics and instrumentation
- centralized error-detection and recovery
- specialization of skills
Table of Contents

Executable Domain Models

Model Execution

Model Capture

Archetype Language

A Direct Translation

An Indirect Translation

Model-Driven Architecture
Executable Domain Models
Abstract classes based on both:

- data, and
- behavior
Build a lifecycle model for each class.

Lifecycles for Temperature Ramp:

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

- **Do Temp. Ramp**: 
  - Batch ID, End Time, End Temp

- **Creating**
  - Entry/Creation
  - Create Temperature Ramp with Batch ID, End Time, End Temp
  - Assign Current Time to Self.StartTime;
  - Generate StartControlling(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/
Create TempertaureRamp with
  BatchID, EndTime, EndTemp
Assign CurrentTime to Self.StartTime;
An Executable Model

<table>
<thead>
<tr>
<th>Batch ID {I}</th>
<th>Amount of Batch</th>
<th>Recipe Name {R2}</th>
<th>Status</th>
</tr>
</thead>
</table>

Lifecycle for Temperature Ramp

Action for Creating

Creating

Entry/
Create TemperatureRamp with BatchID, EndTime, EndTemp
Assign CurrentTime to Self.StartTime;
ActualTemp to Self.StartTemp;
Generate StartControlling (Ramp ID);
Model Execution
An executable model operates on data about instances.
An executable model operates on instances.
Execution

The lifecycle model prescribes execution.

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

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

Some instances exist before the model begins to execute...

Recipe
- Recipe Name {I}
- Cooking Time
- Cooking Temperature
- Heating Rate

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

Pre-existing

Created during execution
Executing the Model

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

<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
To generate code....

<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>
</tbody>
</table>

<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>
....traverse the repository and...

... output text.
The archetype language produces text.

```
.selectAll 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 ,
    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>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
To access attributes of the selected instance....

\[$\{\text{class.Name}\}\]
To traverse an association.....

Not just any one—
the one that’s associated

.select one StateChart related to instances of class->[R13]StateChart

State Chart
Class ID {I, R13}
Name

Class
Class ID {I}
Name
Description

R13
0..1 1

Create Batch( Amount of Batch, Recipe Name)
Filling
Filled( Batch ID )
Cooking
Temp. Ramp Complete( Batch ID )
Emptying
Emptied( Batch ID )
Arbitrary Instance

To select an arbitrary one….

.select any state related to instances of StateChart->[R14]State

Or...

.select any state related to instances of Class->[R13]StateChart->[R14]State
To qualify the selection...

.select any state related to instances of StateChart->[R14]State
where (isFinal == False)
Instance Sets

To select many instances:

\[
\text{select many } \text{stateS related to instances of Class-> } \text{[R13]StateChart ->[R14]State} \text{ where (isFinal==False)}
\]

StateS =

- Filling
- Cooking
- Emptying
To iterate over instances...

.select many states related to instances of Class->[R13]StateChart -> [R14]State where (isFinal == False)
.for each state in states
   ${state.Name},
.endfor

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

```cpp
.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
    };
```

```cpp
public:
    enum states_e
        { NO_STATE = 0 ,
            Filling ,
            Cooking ,
            NUM_STATES = Emptying
        };
```
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
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:

```
.function addPDMDecl( inst_ref class )
private:
  .select many attrs related to class->[R12]Attribute
  .for each attr in attrs
  ${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->[R13]StateChart
protected:
  // state action member functions
  .select many states related by statechart->[R14]State
  .for each state in states
    .invoke addActionFunctionDecl( inst_ref state )
  .endfor
.end function
```
To generate the state action declaration:

```c
.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
```
State Action Definition

To define the state action function . . .

---

Action Group

Class ID \{I\}
State Number \{I\}

Action

Class ID \{I, R34\}
State Number \{I,R34\}
Action ID \{I\}

Signal Generator

Event Name \{I\}
Supplemental Data

Computation

Computation ID \{I\}
Expression

Data Accessor

Accessor ID \{I, R13\}
Class Name Selector Expression

---

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

Consider a system that controls 500 instances “continuously” at 100Hz.
Where Have All The Cycles Gone?

Model the controller so that each:

- 500 instances * 100 cycles transitions per second, or
- 20 microseconds per transition including the actions

```
10ms Expired ( Ramp ID )

Controlling

Temp. Ramp Complete( Ramp ID )
```
Where Have All The Cycles Gone?

Or, we could buy several cheaper processors, each controlling different instances.
Where Have All The Cycles Gone?

Or, we could use a single task that executes periodically.

Diagram:
- Periodic Task
- Controlling
- 10ms Expired
This last architecture 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
  - Event messages
  - Event message
  - Data Copy

- Periodic Task
  - Timer
  - Instance bits
Application Mapping

Recipe
- Recipe Name {I}
- Cooking Time
- Cooking Temperature
- Heating Rate

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

Event Driven Task

Periodic Task
- Temperature Ramp
  - Ramp ID {I}
  - Start Temperature
  - Start Time
  - End Temperature
  - End Time

Start Temperature
- Start Time
- End Temperature
- End Time

Ramp Id Bits
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
Extended Properties

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

```plaintext
.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>
Model-Driven Architecture
Invoke the archetypes and iterate over instances of the corresponding architecture objects to generate the complete source code for the system.
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.

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.
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, and
- time.

Architectures, like programming language compilers, can be bought.
Translative System Generation is a system-construction method based on:

- separating systems into subject matters (domains)
- specifying each domain with an executable UML model
- translating the models
Subject Matter Separation

The application and architecture are separate subject matters.

Separation of application from architecture

Code for the System
Executable UML Models

Executable UML models

Code for the System
Executable UML Models

Executable models can be simulated before coding begins.

**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

**Lifecycle for Temperature Ramp**
- Entry/
  - Create Temperature Ramp with BatchID, EndTime, EndTemp
  - Assign CurrentTime to Self.StartTime;
  - ActualTemp to Self.StartTemp;
  - Generate StartControlling (Ramp ID);
- Creating
  - Do Temp. Ramp (Batch ID, End Time, End Temp)
  - Start Controlling (Ramp ID)
- Controlling
  - Temp. Ramp Complete
- Complete
  - Ended (Ramp ID)
- Creating
Translation is the act of combining the subject matters.
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
Translating Inspiration Into Products.

Makers of BridgePoint ® and DesignPoint ®

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