UmpleRun: a Dynamic Analysis Tool for Textually Modeled State Machines using Umple

Hamoud Aljamaan, Miguel Garzon, Timothy Lethbridge

CRuiSE (Complexity Reduction in Software Engineering) Research Group
University of Ottawa

Sept. 2015
Outline

- Introduction
- Umple
  - Car transmission model
- UmpleRun
- Car Transmission Dynamic Analysis
  - Successful case
  - Failed case
- Future work
Introduction

- **Umple**
  - A model-oriented programming language that allows modelers to model UML constructs textually or graphically
    - focus of this paper: state machines
  - Generate high quality code in a number of targeted programming languages (Java/Ruby/Php ... etc)

- **MOTL**
  - trace specification at the model level for various modeling constructs using model level textual trace directives
Introduction (cont’d)

• What is missing?
  ▪ Validate the model's dynamic behavior.
  ▪ Generate execution traces.

• Why?
  ▪ High-level validation of model dynamic behavior.
  ▪ White box testing of models.
Introduction (cont’d)

• UmpleRun
  ▪ prototype tool – under continuous improvement.
A state diagram can be nested inside a state. The states of the inner diagram are called substates. Figure 8.18 shows a state diagram of an automatic transmission; at the top level this has three states: 'Neutral', 'Reverse' and a driving state, which is not explicitly named. The driving state is divided into substates corresponding to the three gears that the system automatically chooses. The advantage of the nesting is that it shows compactly that the driving substates are all very similar to each other – in particular, that they can all transition to 'Neutral' at any time, upon the user's command. The start symbol inside the driving state shows that it by default starts at the 'First' substate. However, the user can also manually select 'First' or 'Second' to force the transmission to move into, and stay in, these substates.

The notation \texttt{reachSecondSpeed[driveSelected]} illustrates the use of a guard condition. The system will only respond to the indicated event if the condition in square brackets is true. In Figure 8.18, this is used to prevent the transmission from changing gear if the driver had manually selected first or second gear. A guard condition differs from the type of condition we saw in Figure 8.14: a guard condition is only evaluated when its associated event occurs.

Figure 8.19 shows how we have converted Figure 8.14 to use nested substates. Now we need to show only one \texttt{cancel} transition and one \texttt{requestToRegister} transition. Note that the 'Planned' state has a transition that points directly to the 'NotEnoughStudents' substate, and both the transitions to the 'Closed' state come directly from the inner 'EnoughStudents' state. Finally, note that we have added an activity to the 'Canceled' state that deletes all registrations.

Exercises

E162 There is a missing transition in Figure 8.18. Study the diagram, and see if you can find it (do not add any new states or event types).

```java
class CarTransmission {
    Boolean driveSelected = false;
    status {
        neutral {
            selectReverse -> reverse;
            selectDrive -> drive;
            selectFirst -> first;
            selectSecond -> second;
        }
        reverse {
            selectNeutral -> neutral;
        }
        drive {
            exit / { driveSelected = false; }
            selectNeutral -> neutral;
            selectFirst -> first;
            selectSecond -> second;
            first {
                reachSecondSpeed [driveSelected] -> second;
            }
            second {
                reachThirdSpeed [driveSelected] -> third;
                dropBelowSecondSpeed [driveSelected] -> first;
            }
            third {
                dropBelowThirdSpeed -> second;
            }
        }
        before selectDrive {
            driveSelected = true;
        }
    }
}
```
Umple (cont’d)

- MOTL trace directives

![State diagram](image)

```
1 class CarTransmission {
2     trace driveSelected;
3     trace neutral;
4     trace selectReverse;
5 }
```
UmpleRun

- Our tool for running a set of execution scenarios against a targeted model.

- UmpleRun interprets and executes the commands in an execution scenario to produce a model validation verdict, including the failed assertions.

```text
command, method_calls_after_commands 
command_1, values_from_method_calls 
command_2, values_from_method_calls 
... 
command_n, values_from_method_calls 
```
1. **Compilation**: an Umlpe model is parsed, analyzed and a Java system is created.

2. **Packaging**: The Java classes are then packaged into a container (JAR).

3. **Loading** the model into memory: allow creating new instances of the classes.

4. **Running**: The commands in the execution scenario are run against the class instances and the assertions are validated. The validation verdict is produced at this final stage.
Car Transmission Dynamic Analysis

- Two execution scenarios to verify the behavior of the Car transmission state machine and explore successful validation cases of model dynamic behavior.

- Introduce a bug in the Car transmission state machine and study the validation verdict and inject trace directives to produce execution traces from UmpleRun.
Successful validation case

- Execution scenario

```java
1. command, getStatus, getStatusDrive, getDriveSelected
2. new CarTransmission, neutral, Null, false
3. selectReverse, reverse, Null, false
4. selectNeutral, neutral, Null, false
5. selectDrive, drive, first, true
6. reachSecondSpeed, drive, second, true
7. reachThirdSpeed, drive, third, true
8. selectNeutral, neutral, Null, false
```

Compiling CarTrans.ump... success.
Building model... success.
Loading model into memory... success.
Running commands:
- Created CarTransmission
- getStatus = neutral
- getStatusDrive = Null
- getDriveSelected = false
- Executed #selectReverse
- getStatus = reverse
- getStatusDrive = Null
- getDriveSelected = false
- Executed #selectNeutral
- getStatus = neutral
- getStatusDrive = Null
- getDriveSelected = false
- Executed #selectDrive
- getStatus = drive
- getStatusDrive = first
- getDriveSelected = true
- Executed #reachSecondSpeed
- getStatus = drive
- getStatusDrive = second
- getDriveSelected = true
- Executed #reachThirdSpeed
- getStatus = drive
- getStatusDrive = third
- getDriveSelected = true
- Executed #selectNeutral
- getStatus = neutral
- getStatusDrive = Null
- getDriveSelected = false

Done.
Failed validation case

- Model defect
  - Removing code injection for the setting of Boolean attribute ‘driveSelected’.
  - Thus, making guarded events non triggerable.

![State diagram for a car's automatic transmission showing substates](image-url)
Failed validation case (cont’d)

- Trace Directive
  - obtain execution trace.

A state diagram can be nested inside a state. The states of the inner diagram are called substates.

Figure 8.18 shows a state diagram of an automatic transmission; at the top level this has three states: 'Neutral', 'Reverse' and a driving state, which is not explicitly named. The driving state is divided into substates corresponding to the three gears that the system automatically chooses. The advantage of the nesting is that it shows compactly that the driving substates are all very similar to each other – in particular, that they can all transition to 'Neutral' at any time, upon the user's command. The start symbol inside the driving state shows that it by default starts at the 'First' substate. However, the user can also manually select 'First' or 'Second' to force the transmission to move into, and stay in, these substates.

The notation `reachSecondSpeed[driveSelected]` illustrates the use of a guard condition. The system will only respond to the indicated event (`reachSecondSpeed`) if the condition in square brackets is true. In Figure 8.18, this is used to prevent the transmission from changing gear if the driver had manually selected first or second gear. A guard condition differs from the type of condition we saw in Figure 8.14: a guard condition is only evaluated when its associated event occurs.

Figure 8.19 shows how we have converted Figure 8.14 to use nested substates. Now we need to show only one `cancel` transition and one `requestToRegister` transition. Note that the 'Planned' state has a transition that points directly to the 'NotEnoughStudents' substate, and both the transitions to the 'Closed' state comes directly from the inner 'EnoughStudents' state. Finally, note that we have added an activity to the 'Canceled' state that deletes all registrations.

Exercises

- E162 There is a missing transition in Figure 8.18. Study the diagram, and see if you can find it (do not add any new states or event types).

```uml
class CarTransmission {
  trace drive record driveSelected;
}
```

<table>
<thead>
<tr>
<th>Time,Thread,UmpleFile,LineNumber,Class,ObjectName,Operation,Name,Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>*,1,CarTrans.ump,6,CarTransmission,874088044,sm_t,neutral,selectDrive,drive,false</td>
</tr>
<tr>
<td>*,1,CarTrans.ump,6,CarTransmission,874088044,sm_t,drive,selectNeutral,neutral,false</td>
</tr>
</tbody>
</table>
Future

• Automatically generating a comprehensive set of execution scenarios

• Full model execution including associations.
Questions ?

try.ample.org
Command line execution

```
java -jar umplerun.jar model.ump exeScenrio.cmd
```
Information sources

Umple open source website
• http://code.umple.org

Continuous integration and quality assurance pages
• http://cc.umple.org http://qa.umple.org

Umple user manual – MOTL pages
• http://motl.umple.org

Downloading
• http://dl.umple.org