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

Hamoud Aljamaan, Timothy Lethbridge, Miguel Garzón, Andrew Forward
School of Electrical Engineering and Computer Science
University of Ottawa
Ottawa, Canada
hjamaan@uottawa.ca, tcl@eecs.uottawa.ca, mgarzon@uottawa.ca, aforward@eecs.uottawa.ca

***Abstract*— In this paper, we present a tool named UmpleRun that allows modelers to run the textually specified state machines under analysis with an execution scenario to validate the model's dynamic behavior. In addition, trace specification will output execution traces that contain model construct links. This will permit analysis of behavior at the model level.**

Keywords— UmpleRun; Umple; UML; MOTL; state machine; execution trace; analysis

I. Introduction

Umple [1,2] is a model-oriented programming language that allows modelers to model UML constructs textually or graphically and generate high quality code in a number of targeted programming languages. In an extension to the Umple language, MOTL was introduced to allow trace specification at the model level for various modeling constructs using model level textual trace directives [3]. Trace specification of state machines, for instance, has the flexibility of tracing different state machine components such as the whole state machine, any state (at any level of nesting), and specific events.

In this paper, we are presenting the UmpleRun a tool that will allow modelers to execute textually modeled state machines written in Umple and validate their dynamic behavior using execution scenarios. If a modeler has written MOTL trace directives for the model under analysis, UmpleRun will generate informative execution traces in addition to the validation verdict. Execution traces can be used to analyse the system under the inspection.

Benefits of our approach include:

- *High-level validation of model dynamic behavior*: Accomplished by running models against execution scenarios to assert model behavior.
- *White box testing of models*: Thorough analysis, verification and debugging of the models themselves becomes possible. Models can be traced and then executed to produce execution traces, which can in addition be analyzed.

The remaining sections of this paper are organized as follows: Section 2 presents a Car transmission system example that will be used to help illustrate our approach. Section 3 describes the UmpleRun tool and execution scenarios. Section 4 demonstrates the tool usage. Subsequent sections walk through an example of instrumenting our example system and performing dynamic analysis.

II. Example Car Transmission Model to be Executed

In this section, we will present the car transmission model that will be our motivating example through this paper. It will also be used to explain Umple and MOTL syntax. The Car transmission model was inspired by a similar model in Lethbridge and Laganière's book [4]. The model consists of one class with car transmission behavior captured by the state machine shown in Fig. 1. The state machine consists of three states: two simple states ('neutral' & 'reverse') and one composite state ('drive'). The initial state for state machine is 'neutral' state. The composite 'drive' state has three substates for transmission levels (i.e. 'first', 'second', and 'third'). There are events to trigger transitions between states and some are guarded as in [driveSelected], where event reachSecondSpeed will not cause a transition unless Boolean guard is evaluated to true.

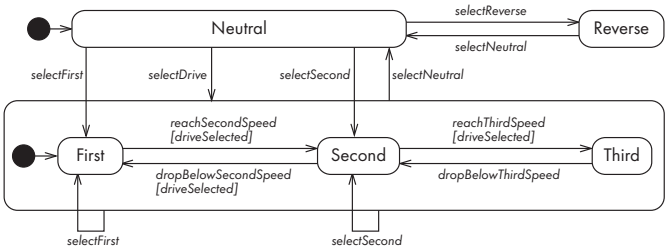

Fig. 1. Car Transmission State Machine [4]

The Car transmission system behavior was textually modeled using Umple as shown in Listing 1. Lines 3 to 29 represent the car transmission status state machine. Line 4 shows the state name, while line 14 declares that there is an exit action associated with this state. Line 19 is an example of a guarded event. In line 30 we are doing a code injection to event selectDrive to set the Boolean attribute 'driveSelected' to true to differentiate it from the manual triggering caused by event 'selectFirst' which indicates the manual diving mode.

Listing 1: Car transmission Umple code

Umple

```
1  class CarTransmission {
2    Boolean driveSelected = false;
3    status {
4      neutral {
5        selectReverse -> reverse;
6        selectDrive -> drive;
7        selectFirst -> first;
8        selectSecond -> second;
9      }
10     reverse {
11       selectNeutral -> neutral;
12     }
13     drive {
14       exit / { driveSelected = false;}
15       selectNeutral -> neutral;
16       selectFirst -> first;
17       selectSecond -> second;
18       first {
19     reachSecondSpeed [driveSelected] -> second;
20       }
21       second {
22     reachThirdSpeed [driveSelected] -> third;
23   dropBelowSecondSpeed [driveSelected] -> first;
24       }
25       third {
26         dropBelowThirdSpeed -> second;
27       }
28     }
29   }
30   before selectDrive {
31     driveSelected = true;
32   }
33 }
```

Using MOTL, we can write trace directives for trace specification of attributes and state machine. Listing 2 presents an example of trace directives for the car transmission system. Using Umple's mixin capability, we can write MOTL trace directives independent of the model. Line 2 indicates that we are interested in tracing any changes to the value of Boolean attribute 'driveSelected'. Line 3 traces any incoming or outgoing transitions from or into state 'neutral'. The directive in line 4 will trace whenever event 'selectReverse' is triggered causing a transition. More details on MOTL syntax can be found here [3].

Listing 2: MOTL trace directive examples

Umple

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

III. UMPLERUN

UmpleRun is our tool for running a set of execution scenarios against a targeted model. This takes as input an execution scenario, a template for which is shown in Fig. 2. Line 1 of any scenario has the keyword 'command', then a set of query methods to execute at every step of execution. Lines 2 and onwards are of a set of commands to be executed to drive the scenario, along with assertions of the expected return values of the query methods. The commands can be object constructor invocations, or method calls such as state machine event calls.

Listing 3: Execution scenario template

Execution Scenario

```
command, method_calls_after_commands ...
command_1, values_from_method_calls ...
command_2, values_from_method_calls ...
...
command_n, values_from_method_calls ...
```

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

The dynamic validation process in this architecture is presented in Fig. 2 and described below:

1. **Compilation**: The input is an Umple model (named uModel). At this stage, uModel is parsed, analyzed and a Java system is created from the input model by the Umple compiler.
2. **Packaging**: The Java classes are then packaged into a container (JAR).
3. **Loading the model into memory**: A dynamic loader is created using the previously obtained JAR that will allow creating new instances of the classes previously generated by our input model.
4. **Validation**: 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.

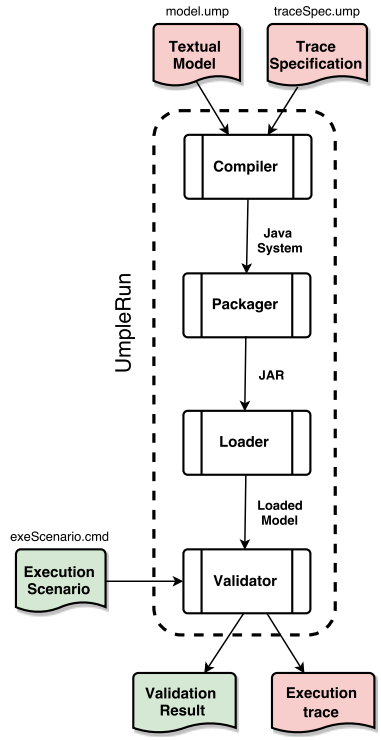

Fig. 2. Model execution and validation in UmpleRun

The command used to validate models dynamic behavior using UmpleRun is:

```
java –jar umplerun.jar model.ump exeScenario.cmd
```

IV. Car Transmission Dynamic Analysis

In this section, we illustrate the application of our tracing tool and UmpleRun to the Car transmission model we outlined earlier. First, we will create execution scenarios to verify the behavior of the Car transmission state machine and explore successful validation cases of model dynamic behavior. Then, we will introduce a bug in the Car transmission state machine and study the validation verdict and inject trace directives to produce execution traces from UmpleRun.

A. Successful validation verdict

We created two execution scenarios to validate the Car transmission model behavior as seen in Listing 4 and Listing 6, with each representing a sequence of commands executed against the model and then we assert the model constructs values after each command.

The execution scenario in Listing 4 follows the template described in Listing 3. The first line shows the three 'get' method calls that are invoked after every command; these query the current state of Car transmission state machine, the current state in the composite state 'drive' and the value of Boolean attribute 'driveSelected'.

Each subsequent line from 2 to 8 begins with a command to be executed and then the expected values after completing the executed command. The command on Line 2 creates a new Car transmission object. Upon creation of this we expect that we will be in initial state 'neutral', with the nested state value of Null since we haven't entered any nested state. The Boolean attribute should be the initialized value, which is false.

Line 3 executes a 'selectReverse' event with the resulting state expected to be 'reverse', and the Boolean attribute value remains false. Line 5 in the execution scenario specifies that event 'selectDrive' will be triggered and that we should enter the composite state 'drive' with initial state expected to be 'first', and Boolean attribute value changes to true, which means that transmission will be automatic.

Listing 4: Execution scenario (1)

	Execution Scenario
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

Overall, this execution scenario provides the means for modelers to assert the dynamic behavior of their state machines and alert them by detecting if there is any unexpected behavior. We executed the previously explained execution scenario on the Car transmission state machine using UmpleRun and we received a detailed validation result as shown in Listing 5.

If the validation using UmpleRun is not successful, then the modeler can do a detailed step-by-step examination of the more detailed trace output to obtain clues as to what might have gone wrong, as we will see in the next section.

Listing 5: Successful dynamic validation result (1)

```
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.
```

We created another execution scenario as seen in Listing 6, where an event 'selectFirst' will force the Car transmission into manual.

Listing 6: Execution scenario (2)

	Execution Scenario
1	command,getStatus,getStatusDrive,getDriveSelected
2	new CarTransmission, neutral, Null, false
3	selectReverse, reverse, Null, false
4	selectNeutral, neutral, Null, false
5	selectFirst, drive, first, false
6	selectNeutral, neutral, Null, false

We ran the previous execution scenario using UmpleRun, and received the following successful validation verdict.

Listing 7: Successful dynamic validation result (2)

```
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 #selectFirst
  getStatus = drive
  getStatusDrive = first
  getDriveSelected = false
  Executed #selectNeutral
  getStatus = neutral
  getStatusDrive = Null
  getDriveSelected = false
Done.
```

B. *Failed validation verdict*

In this section, we present the dynamic validation of a failing behavior by introducing a defect in the design of the Car transmission state machine and by running the previous execution scenario in Listing 4 against the faulty model. A model defect has been created by removing the code injection for the setting of Boolean attribute 'driveSelected'. Thus, making guarded events non triggerable. After execution of the scenario on the Car transmission state machine, UmpleRun produces a validation verdict indicating failed assertions.

Listing 8 displays the UmpleRun verdict output signifying five failed assertions from the expected state machine behavior. The assertions indicate that after the triggering of event selectDrive the value of Boolean attribute is not as expected. Then, after entering composite state 'drive' and the triggering of event reachSecondSpeed, the resulting state should be 'second', but the failed assertion indicated that the current state 'first'. A similar situation occurred in the fourth failed assertion. Indicating there have been non-triggerable events.

Listing 8: Failed dynamic validation result

```
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
  !!! ASSERTION FAILED on getDriveSelected,
EXPECTED true, ACTUAL false
  Executed #reachSecondSpeed
  getStatus = drive
  !!! ASSERTION FAILED on getStatusDrive,
EXPECTED second, ACTUAL first
  !!! ASSERTION FAILED on getDriveSelected,
EXPECTED true, ACTUAL false
  Executed #reachThirdSpeed
  getStatus = drive
  !!! ASSERTION FAILED on getStatusDrive,
EXPECTED third, ACTUAL first
  !!! ASSERTION FAILED on getDriveSelected,
EXPECTED true, ACTUAL false
  Executed #selectNeutral
  getStatus = neutral
  getStatusDrive = Null
  getDriveSelected = false
Done.
```

To study the failed validation verdict further, we wrote a trace directive to examine the failed modeling element as presented in Listing 9. The trace directive will trace composite state 'drive' and record the value of Boolean attribute 'driveSelected' at the same time.

Listing 9: Trace directive for defect investigation

Umple

```
1 class CarTransmission {
2   trace drive record driveSelected;
3 }
```

Adding the above trace specification to the model, and then rerunning the validation using UmpleRun, we obtain the execution trace in comma-separated-value (CSV) form as shown in Listing 10 (we have replaced the system time and the object hash code values with * to save space in the paper).

The operation code 'sm_t' in line 2 shows that this trace was recorded when a state event was triggered named 'selectDrive' that made a transition from state 'neutral' to state 'drive', and reported the value of the Boolean attribute was false. The next event triggered in composite state 'drive' was an exit transition by event 'selectNeutral', confirming that none of the events inside composite state 'drive' was triggered.

Listing 10: Execution trace

Execution trace

```
Time,Thread,UmpleFile,LineNumber,Class,Object,Operation,Name,Value
*,1,CarTrans.ump,6,CarTransmission,*,sm_t,neutral,selectDrive,drive,false
*,1,CarTrans.ump,6,CarTransmission,*,sm_t,drive,selectNeutral,neutral,false
```

V. Related Work

Derezińska and Pilitowski [5] presented an execution framework (FXU) for UML state machines to verify their correctness. FXU consists of two components: a code generator and a run time library. Execution is realized by transforming UML classes and state machines into a C# implementation as follows: First, a modeler creates a UML model using any modeling tool, then the model is exported as an XMI file. Next, code for the targeted programming language (i.e. C#) is

generated from the model. Third, the generated code is modified, compiled, and linked to a run time library. Finally, code is executed to reflect model behavior.

As an extension to FXU, FXU tracer [6] a graphical interface extension to FXU has been implemented showing a tree representation of the UML model, and a textual information about the tracing process. FXU tracer requires the generation of trace logs from the FXU framework during state machine execution. After generation of trace logs, these logs are fed to the FXU tracer and tracing is conducted either step-by-step or stopping at inserted breakpoints. The FXU tracer suffers from design flaws and certain other limitations. The mechanism for trace logs creation in the FXU environment wasn't specified and information collected during state machine execution is not explained. Obviously, as state machines get more complex, the size of trace logs becomes a concern and the authors didn't address it in the FXU environment. Furthermore, as indicated by the authors, not all events are supported and code generation is limited to C#.

StateForge [7] is a tool that transforms state machine models expressed in XML into C, C++, and Java source code. StateForge includes some implemented observer classes that observe and record state machine behavior. Further, more observers can be created by implementing an observer interface. However, modelers using StateForge can't limit the scope of observations to substates, transition, etc.

In the area of model execution via virtual machines, Mayerhofer et al. [8,9] proposed extensions to the standardized fUML virtual machine to enable the debugging of models at run time. These extensions aim to overcome the limitations of fUML in monitoring the models' runtime behavior. Three models were proposed: (1) Trace model, a dedicated trace metamodel capable of recording the model execution carried out by the fUML virtual machine. (2) Event model, monitors run time state and triggers events based on changes to run time state. (3) Command API: a set of commands that enables the control of models execution.

VI. Conclusion

This paper presented our approach for model dynamic analysis for modelers and other developers performing model driven development. UmpleRun is a tool to automatically drive execution of scenarios to validate dynamic behavior. The software engineer drives execution using UmpleRun; if execution is not as expected, then he or she can examine the detailed execution trace. A key benefit of this work is that it allows analysis of behavior of a system generated from a UML specification, without the need to instrument generated code and allow the generation of execution traces referencing modeling constructs.

As future work, we are investigating automatically generating a comprehensive set of execution scenarios and using this to validate model dynamic behavior. We anticipate enhancing UmpleRun so that the expected output to be matched can examine details of the tracer output, including using pattern matching. Finally, this work can be further extended by building tools that can formally verify conformance of traces to the specified UML models.

Acknowledgment

Hamoud Aljamaan would like to thank King Fahd University of Petroleum and Minerals (KFUPM) for their financial support during his PhD studies.

References

[1] M. A. Garzón, H. Aljamaan, and T. Lethbridge, "Umple : A Framework for Model Driven Development of Object-Oriented Systems," in 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER), 2015.

[2] A. Forward, O. Badreddin, T. C. Lethbridge, and J. Solano, "Model-driven rapid prototyping with Umple," Softw. Pract. Exp., vol. 42, no. 7, pp. 781–797, Jul. 2012.

[3] H. Aljamaan, T. C. Lethbridge, O. Badreddin, G. Guest, and A. Forward, "Specifying Trace Directives for UML Attributes and State Machines," in 2nd International Conference on Model-Driven Engineering and Software Development, 2014, pp. 79–86.

[4] T. Lethbridge and R. Laganiere, Object-Oriented Software Engineering: Practical Software Development using UML and Java, 2nd ed. McGraw-Hill, Inc., 2004.

[5] A. Derezinska and R. Pilitowski, "Correctness issues of UML class and state machine models in the C# code generation and execution framework," in 2008 International Multiconference on Computer Science and Information Technology, 2008, pp. 517–524.

[6] A. Derezinska and M. Szczykulski, "Tracing of state machine execution in the model-driven development framework," in 2nd International Conference on Information Technology (ICIT), 2010, pp. 109–112.

[7] "StateForge - State machine generator & state diagram editor." [Online]. Available: http://www.stateforge.com/. [Accessed: 09-Jun-2014].

[8] T. Mayerhofer, P. Langer, and G. Kappel, "A runtime model for fUML," in Proceedings of the 7th Workshop on Models@run.time - MRT '12, 2012, pp. 53–58.

[9] T. Mayerhofer, "Testing and debugging UML models based on fUML," in Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 1579–1582.