Specifying Trace Directives for UML Attributes and State Machines

Hamoud Aljamaan, Timothy C. Lethbridge, Omar Badreddin, Geoffrey Guest, Andrew Forward
School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa, Ontario, Canada
hjamaan@uottawa.ca, tcl@eecs.uottawa.ca, obadr024@uottawa.ca, ggues044@uottawa.ca, aforward@gmail.com

Keywords: Tracing, UML, Trace directive, Attributes, State machines, Umple.

Abstract: Developers using model driven development (MDD) to develop systems lack the ability to specify traces that operate at the model level. This results in specification of traces at the generated code level. In this paper, we are proposing trace directives that operate at the model level to specify the tracing of UML attributes and state machines. Trace directives are implemented as part of the Umple textual modeling language, thus these directives can be expressed in a textual form. Trace code will be injected into system source code that corresponds to trace directives specified at the model level.

1 INTRODUCTION

Over the history of software, software development has been continuously becoming more complex. Many researchers and software practitioners have strived to simplify the process of software development by introducing levels of abstraction. In the early days of the software industry, software developers created software using assembly languages and machine code. As these languages became inadequate, successively higher-level programming languages were introduced as abstraction layers over machine code. Nowadays, as software complexity increases, researchers are directing their attention to model driven development (MDD).

Huge advantages are gained when software developers adopt the MDD approach (Selic 2003, Sendall, Kozaczynski 2003), with models representing the main artifact in the development cycle. One of the biggest advantages is the ability to view targeted systems at a high level of abstraction as compared to programming languages.

Tracing is used to understand the behaviour of systems in order to debug or monitor them. Data collected while tracing ranges from the output of simple programmed log commands to more sophisticated traces containing lower-level events triggered by tools that dynamically instrument user and kernel spaces. However, injecting trace code into a system is not a trivial task and controlling tracing so it is both effective and efficient is even more difficult.

Historically, developers have traced software using techniques such as adding simple print statements, breakpoints in a debugger, or more sophisticated tracing tools such as Dtrace (Cantrill 2006, Cantrill, Shapiro & Leventhal 2004) or Linux Linux Trace Toolkit - next generation (LTTNG)(Desnoyers, Dagenais 2009, Desnoyers et al. 2012) that allows tracing to be initiated either at compile time or run time. In situations where code is generated, such as when a pre-processor or model driven development is involved, these tracing techniques tend to be limited to working with generated code. They therefore require the understanding of the generated code's structure, and require extra work to map changes and understanding to the original source. Furthermore, tracing code needs replacing when the code is re-generated. Tracing thus occurs at a level of abstraction below the level at which the system is implemented.

Existing techniques focus on tracing at the code level: tracing function/method calls, lines executed, variables being set, etc. There has been little or no research into tracing at the model level. In this paper, we propose the notion of trace directives that operate at the model level by defining its syntax, implementing the parser, code generator and tests. Our tracing language, which we call Model Oriented Tracing Language (MOTL) is incorporated into the Umple technology for model-oriented programming (Forward, Lethbridge & Brestovansky 2009,

See http://www.modelsward.org/Abstracts/2014/MODELSWARD_2014_Abstracts.htm

Forward et al. 2012, Lethbridge, Forward & Badreddin 2010, Lethbridge et al. 2011) . Trace directives will allow developers to gain the ability to specify traces of different UML entities at the model level without the need to modify the generated code. This field, which we call model-oriented tracing, is currently immature, so there are tremendous opportunities to make contributions.

The remainder of this paper is as follows: Section 2 sheds light on the model oriented tracing research area and specifies the research problem we are addressing. Section 3 provides technical background necessary for our proposed tracing directives. Section 4 explores the details of our proposed trace directives. Section 5 illustrates the usage of trace directives by example. Section 6 presents related work in the literature. Finally, the paper is concluded in Section 7.

2 MODEL-ORIENTED TRACING

The concept of model-oriented tracing is the following: A developer modeling in UML, and generating much of their system directly from the UML model, should be able to indicate that they want any of the UML entities to be traced. The developer should not be forced to inject tracing into generated code, which he or she may never otherwise look at, and which is subject to re-generation every time the system changes.

Examples of UML entities to trace include:

- **Attributes**: Tracing attributes is conceptually similar to tracing variables, except that the level of abstraction is higher because the implementation of the attribute is deferred to the code generator, and the attribute may have automatically managed constraints, specialized initialization conditions, and triggers such that changes to the attribute cause system events.
- **State machines**: Tracing can be performed at state entry and exit, as well when particular transitions occur or when named events occur. Since state machines can be nested at several levels of depth, tracing can be scoped to certain substates. Tracing of attributes can be constrained to occur in specific states.

The above examples are the focus of this paper, but we are working on tracing UML associations, and it would still be possible to trace functions, methods and lines of code as has been traditionally possible. The key is that model-oriented tracing adds another level of abstraction to the elements that can be traced.

2.1 Problem Statement

The problem statement for this research is as follows (Aljamaan, Lethbridge 2012):

Developers frequently need to deploy tracing to debug programs, to test them in a white-box manner, to understand their internal behaviour and to detect anomalies such as hacker intrusion or performance degradation. Current technologies, however, only allow injecting traces into functions (procedures or methods) and data items. The ability to systematically trace at the model level is missing, since there are no tools available to meet this need.

3 UMPLE

Umple (Timothy C. Lethbridge et al. 2011, Badreddin, Forward & Lethbridge 2012, Forward, Badreddin & Lethbridge 2010, Forward et al. 2011) is a technology for modeling textually in UML and can be seen as both a modeling and a programming language. Software developers can represent UML concepts such as classes, attributes, associations and state machines in Umple, and can embed ordinary methods in an Umple class. As a result, an Umple program looks like a standard program (e.g. in Java) with some extra features added.

Part of the Umple philosophy is that software developers describing the system at a high level of abstraction will have less code to write and hence will have higher productivity. Umple users can specify the following high level constructs, most of which are based on UML. The Umple user manual provided full details (Cruise 2013) :

- **Classes and Interfaces**
- **Attributes**
- **State Machines:** Umple provides a complete and powerful textual specification of state machines at the model level. Umple supports nested or concurrent states, transitions with guards, entry or exit actions, and interruptible activities.
- **Associations:** These specify the sets of links among objects that can exist at run time.
- **Patterns:** Umple supports the singleton and immutable patterns plus the definition of database-style keys.
- **Aspect Orientation:** Umple users can insert code to be run before (i.e. as a precondition) or

after (i.e. as a postcondition) Umple-defined actions on attributes, associations and the components of state machines.

Developing in Umple can be performed using standard command-line tools (Lethbridge, Forward & Badreddin 2012) , using an Eclipse plugin, or, for small systems, with UmpleOnline (Lethbridge T.C., Forward & Badreddin 2012) . Programmers can use Umple in the manner they are accustomed, adding UML constructs as they gain confidence. On the other hand, modelers can take the opposite approach, starting with high-level models and then adding methods to specify detailed algorithms and in order to fully implement other details of the system not covered by the modeling constructs. Readers are referred to UmpleOnline (Lethbridge T.C., Forward & Badreddin 2012) for a list of systems modeled in Umple.

4 TRACE DIRECTIVES

Tracing is specified using structured trace directives that can be placed anywhere in Umple code describing a model. Generally, a trace directive is structured as follows:

trace <UMLconstruct> <Constraints> **;**

Trace directives start with a keyword 'trace' and end with a semicolon ';'. After the trace keyword, a UML construct to be traced is specified (attribute, association, state, etc.). The scope of tracing can be limited using appropriate optional constraints and conditions that can switch tracing on or off in certain situations. In addition, constraints can be used to specify data that will be outputted.

4.1 Attribute Tracing

Trace directives allow the tracing of attributes at the model level; trace output can be generated whenever an attribute value is changed (i.e. a setter is called) or/and when the value is accessed (i.e. the getter method is called). Moreover, attribute tracing can be limited based on constraints such as a condition being true or a maximum number of trace occurrences being reached.

Attribute tracing can occur in three modes: when the attribute value is accessed, changed, or both. Therefore, a selection of three keywords for each mode can be used as follows:

- **Attribute value changed**: When 'set' keyword is specified before the traced attribute, tracing occurs when the attribute setter method is executed. This is also the default case when no keyword is specified before the traced attribute in a trace directive.
- **Attribute value accessed:** When 'get' keyword is specified before the traced attribute, tracing occurs when the attribute getter method is executed.
- **Attribute value accessed or changed:** When both keywords 'set' and 'get' separated by ',' are used, tracing occurs either when getter or setter methods are executed.

4.2 State Machine Tracing

Developers can specify that they want a certain state, transition, or event to be traced, and have the ability to limit the scope of tracing to a certain level of substates, or trigger tracing when certain trace-based conditions become true. The following are some details:

- **State:** Tracing a state means trace output will be recorded for each incoming and outgoing transition. In addition, substates are traced, recursively. But, if tracing of substates is not desired and tracing scope should be limited to a certain level, then the 'level' keyword can be specified followed by an integer to represent the tracing level (i.e. recursion depth); level 0 would mean trace the current level and no substates.
- **Limiting to entry and exit:** Tracing a state normally involves the tracing of its entries and exits, but tracing can be limited to either case. In a trace directive, we can use the 'entry' keyword before a traced state to indicate tracing of its entries only, or the 'exit' keyword to indicate tracing of its exits.
- **Transitions:** A trace directive can trace a specific transition by using keyword 'transition' which involves the tracing of the original state, the destination state and the triggering event.
- **Events:** All occurrences of an event name may be traced too.

4.3 Trace Control with Constraints

We identified various ways to control the scope of tracing and what needs to be traced. Tracing can be controlled by specifying post- or pre-conditions that need to be satisfied before tracing triggers. In addition, just as with attributes, tracing can be

controlled by time manipulation and the specification of number occurrences.

4.3.1 Basic conditions

Basic conditions are used to control tracing by injecting code that outputs trace data only upon condition satisfaction. Conditions can be either pre-conditions or post-conditions. The 'where' keyword is used to inject tracing code with a pre-condition. The 'giving' keyword is used to inject tracing code with a post-condition. Note that these condition expressions work the same regardless of the entity being traced – i.e. they apply to state machines and methods. The 'giving' keyword is so-named because the tracing occurs when the operation 'gives' a certain result.

4.3.2 Occurrences

In addition to basic conditions, the functionality of tracing for a certain number of occurrences (i.e. appearances) of trace output is implemented. This is achieved by using the 'for' keyword followed by an integer to specify the number of trace output occurrences desired.

4.3.3 Timeline

Tracing can be limited for a period bracketed by a condition. Two keywords were designated for this purpose: The 'until' keyword triggers tracing to start and continues tracing until a given condition is satisfied, after which tracing stops permanently. The 'after' keyword provides the opposite behaviour; tracing will start once a given condition is satisfied and then continues indefinitely without any interruption.

4.3.4 Record

There are situations where tracing may necessitate the monitoring of other UML constructs for debugging and analysis purposes such as the tracing of additional attributes, state machines etc. We have provided the ability to specify these using the 'record' keyword. In addition, this record statement can be used to record an arbitrary string.

4.4 Trace Output

Trace directives written at the model level will inject traces in the source code generated from the model. Data collected from these injected traces depends on the tracing technology being used. We implemented two primitive tracers that collect a wide variety of information during run time. These tracers are:

- **Console:** Tracing using print statements has been used since the beginning of software. It forms the most basic and primitive type of debugging. However, being primitive doesn't necessarily mean that it's of no importance. It can be used for educational purposes or in situations where no deep tracing is needed. The console tracer directs trace output to standard error.
- **File:** Tracing output is stored in a specific file, which can act as a trace log file. Such files are stored for later analysis.

Information collected once a trace is triggered during run time consists of static, dynamic, and hard coded values as shown in Table 1. Dynamic values indicate that values differ for different traces (e.g. traces are triggered in different points of time). On the other hand, there some static values are recorded once a trace triggers (e.g. trace directive information that relates to triggered trace). Finally, we have identified a list of hard coded values to indicate what kind of operation triggered this trace (e.g. at_s means trace is triggered by an attribute set).

Table 1: Trace output components

Output component	Value
Timestamp	Dynamic
Thread	Dynamic
Name of Umple File	Static
Line number	Static
Class name	Static
Object hash code	Dynamic
Operation	Hard coded

We plan to add tracers for tools like Dtrace and LTTNG. Tracer selection is done using a tracer statement, which is structured as follows:

tracer <TracerType> ;

The tracer statement starts with 'tracer' keyword followed by the tracer chosen and ends with a semicolon ';'. If no tracer statement is included in the model, then trace output is directed to standard error.

5 EXAMPLE

In this section we present a student registration system to show capabilities and expressiveness of our proposed trace directives in specifying tracing at the model level. The student registration system consists of four classes with attributes and associations between them. In addition, a state machine is defined inside class CourseSection to handle and capture the desired dynamic behaviour of our system. Umple code for the student registration system is provided in the Appendix.

Figure 1 shows the class diagram for the student registration system. Each course has a unique course code with course description. Each course can have zero to many course sections, but a course section can be assigned to one course. Each course section has attributes to hold the information related to a course section such as minimum and maximum number of students allowed in the class if it is to be taught, the current class size, and the section id. A course section can have zero to many registration with a single student assigned to each registration.

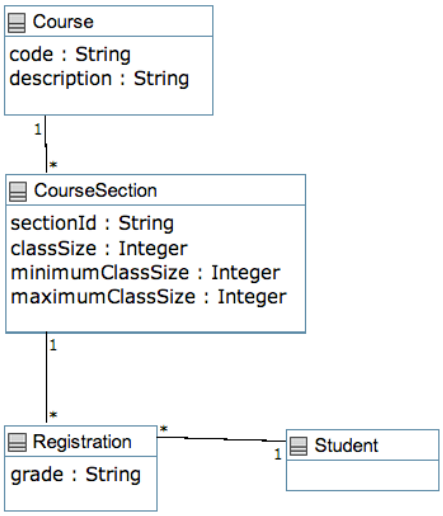

Figure 1: Student registration system class diagram

Figure 2 shows the state machine diagram for the student registration system. There are five states with different events to transit between states. For example, the initial state is the Planned state and the event openRegistration triggers a transition from state Planned to state OpenNotEnoughStudents. Some events are guarded, such as the register event from state OpenNotEnoughStudents to state OpenEnoughStudents.

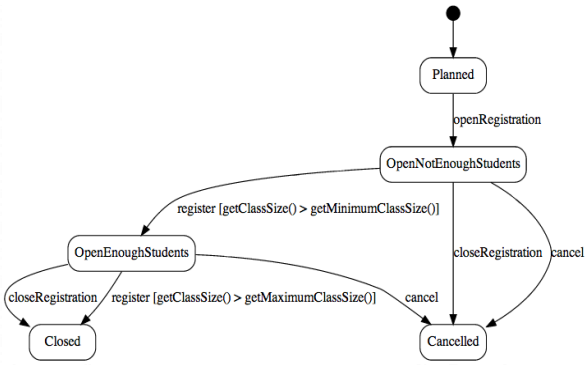

Figure 2: Student registration system state machine

Now, developers can utilize the expressiveness of trace directives to specify traces at the model level for any debugging, monitoring, or analysis purposes. Using code mixins in Umple, trace directives can be written as a tracing script and seen as independent from the model. The Next code snippet shows a wide range of possible trace directives, followed by their description:

```
class CourseSection
{
  // Trace directive 1
  trace classSize record sectionId;
  // Trace directive 2
  trace sectionId where [classSize
== maximumClassSize];
  // Trace directive 3
  trace CourseSectionStm;
  // Trace directive 4
  trace entry Cancelled record
classSize;
  // Trace directive 5
  trace Closed record classSize;
}
```

Listing 1: Student registration system trace directives

- **Trace directive 1.** Developer wants to keep track of any changes to class size in a course section. Trace of class size and course section id will trigger when the value of class size is changed (i.e. the class size set method is called).
- **Trace directive 2.** Trace ids of course sections with their class size reaching the maximum capacity.
- **Trace directive 3.** Trace the whole state machine that includes all states with any events and transitions.
- **Trace directive 4.** The developer might want to check the class size of cancelled sections for debugging and analysis purposes.
- **Trace directive 5.** Developer would like to track the class size of closed sections.

6 RELATED WORK

Limited research has been found that tackles the problem of specifying traces at the model level. Different approaches have been proposed to allow developers to trace models either through executable models, or by executing generated code from models. Each approach has its advantages and disadvantages. However, none of them provided a complete specification of traces at the model level.

G. Eakman (Eakman 2000) introduced the idea of instrumenting UML models for debugging purposes. Systems are more visible at the model level than at the code where it's difficult to visualize systems due to implementation details. The main objective of this idea is that model level instrumentation will provide full access to the system under test at the level of UML modeling, allowing a glass box approach to testing with greater observability, controllability, and testability. The proposed instrumentation will occur at the translational process where UML models are mapped to the implementation (i.e. targeted programming language) by the model compiler. Hence, the model compiler will be responsible for the insertion of instrumentation into generated code and ensuring that instrumentation will not add any additional functionality to the software, other than enhanced testability.

In Eakman's proposed instrumentation approach, important data values, attributes, inputs, and control points must be identified and appropriate instrumentation added. Instrumentation can be triggered based on data access and/or dynamic behaviour. In data access situations, attributes values can be monitored and state machines response to an event can be recorded, while in dynamic behaviour, creation and deletion of instances and/or associations are monitored.

A. Derezinska and M. Szczykulski (Derezinska, Szczykulski 2013) presented a framework for executable UML, called FXU, that performs transformation from a UML class and state machine model into a C# implementation. Their framework consists of two main components: a code generator that assumes direct model transformation to the target code, and a runtime library that contains realization of different UML meta-model elements. As an extension to this framework, the FXU tracer (Derezinska, Szczykulski 2010) was designed to allow the tracing of state machine execution generated in C# code. Tracing will be specified and commence after the state machine execution using log files created in the FXU environment. The FXU tracer is intended to help increase state machine comprehension and verify state machines behavioural correctness. Many disadvantages and drawbacks can be seen from this tool design:

- Specification of traces can only be done after the execution of the application and not simultaneously as the application is being executed.
- The FXU tracer can only use trace logs produced by the FXU environment.
- There is no control of information collected during state machine execution, which results in collecting irrelevant information and producing massive files.
- The authors indicate that not all events can be traced since the FXU environment doesn't log all events that occur during state machines execution.
- Tracing of state machines can't be specified in terms of other UML constructs (e.g. tracing a state when an attribute has a certain value).
- This tool is limited to C#

K. Mehner (Mehner 2002) developed the JaVis environment for visualizing and debugging of concurrent and sequential Java programs. Their motivation is that debugging of concurrent Java programs is complex due to usage of threads. The JaVis environment consists of three stages: collecting traces while executing, visualizing these traces, and performing thread deadlock detection and analysis.

The tracing component of JaVis uses the Java Debug Interface (JDI) of the Java Platform Debugger Architecture to allow the collection of debugging and tracing information from a running Java programs. Traces are represented in a textual format with each trace entry consisting of a single line. Each trace line contains a method entry, a method exit, object IDs, calling thread, and other information that can be used for deadlock detection. Usage of JDI provides advantages when used for tracing such as it allows tracing of remote and already running Java programs and does not require the source code to be modified. After the generation of traces, they are visualized using UML 1.6 sequence and collaboration diagrams. These diagrams will show dependencies between different program threads to help detect deadlocks.

To conclude, a limited number of research papers have been found that directly relate to our work, which is an indication that this area needs further investigation. One paper (Eakman 2000) shared the same research objectives as ours but didn't provide any deep details or any sort of implementation

strategies. Our trace directives should significantly contribute to the area of model-driven development.

7 CONCLUSIONS

This paper explored and presented the notion of trace directives that allows modelers to specify traces of UML attributes and state machines at the model level. Syntax of proposed directives is explained and implementation is incorporated as part of the Umple technology. Trace directives are expressed in a textual form with a simplified syntax and can be written as a trace script independent of the model. Usage of these directives is described by an example. Currently, the proposed trace directives are limited by the capabilities of the modeling language (Umple).

We foresee many directions for our research. Next in our research roadmap, we plan to implement tracing of associations by allowing modelers to specify tracing of associations and base tracing constraints on the cardinalities of associations. Support of additional tracers (e.g. LTTNG) is planned. Experiments will be conducted to evaluate the usability and usefulness of our proposed trace directives.

ACKNOWLEDGEMENTS

Hamoud Aljamaan would like to thank King Fahd University of Petroleum and Minerals (KFUPM) for their financial support during his PhD studies. We also thank Ericsson, Defence Research and Development Canada (DRDC), and NSERC for sponsoring this research.

REFERENCES

Aljamaan, H. & Lethbridge, T.C. 2012, "Towards Tracing at the Model Level", *19th Working Conference on Reverse Engineering (WCRE), 2012*, 15-18 Oct. 2012, pp. 495.

Badreddin, O., Forward, A. & Lethbridge, T.C. 2012, "Model oriented programming: an empirical study of comprehension", *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research,* , pp. 73-86.

Cantrill, B. 2006, "Hidden in Plain Sight", *Queue,* vol. 4, no. 1, pp. 26-36.

Cantrill, B.M., Shapiro, M.W. & Leventhal, A.H. 2004, "Dynamic instrumentation of production systems", *Proceedings of the annual conference on USENIX Annual Technical Conference*USENIX Association, , pp. 2.

Cruise 2013*, Umple User manual*. Available: http://cruise.site.uottawa.ca/umple/UsingUmpleOnline.html.

Derezinska, A. & Szczykulski, M. 2013, "Towards C# Application Development Using UML State Machines: A Case Study" in *Emerging Trends in Computing, Informatics, Systems Sciences, and Engineering*, eds. T. Sobh & K. Elleithy, Springer New York, , pp. 793-803; 68.

Derezinska, A. & Szczykulski, M. 2010, "Tracing of state machine execution in the model-driven development framework", *2nd International Conference on Information Technology (ICIT), 2010*, 28-30 June 2010, pp. 109.

Desnoyers, M. & Dagenais, M. 2009, "LTTng, Filling the Gap Between Kernel Instrumentation and a Widely Usable Kernel Tracer", *Linux Foundation Collaboration Summit*.

Desnoyers, M., McKenney, P.E., Stern, A.S., Dagenais, M.R. & Walpole, J. 2012, "User-Level Implementations of Read-Copy Update", *IEEE Transactions on Parallel and Distributed Systems,* vol. 23, no. 2, pp. 375-382.

Eakman, G. 2000, "Strategies for Debugging Embedded Systems", *Embedded Systems Programming,* , pp. 139-147.

Forward, A., Badreddin, O., Lethbridge, T.C. & Solano, J. 2011, "Model-driven rapid prototyping with Umple", *Software: Practice and Experience,* .

Forward, A., Badreddin, O. & Lethbridge, T.C. 2010, "Umple: Towards Combining Model Driven with Prototype Driven System Development", *IEEE International Symposium on Rapid System Prototyping (RSP)*.

Forward, A., Lethbridge, T.C. & Brestovansky, D. 2009, "Improving program comprehension by enhancing program constructs: An analysis of the Umple language", *IEEE 17th International Conference on Program Comprehension, 2009 (ICPC '09)*, pp. 311.

Forward, A., Badreddin, O., Lethbridge, T.C. & Solano, J. 2012, "Model-driven rapid prototyping with Umple", *Software Practice and Experience,* vol. 42, no. 7, pp. 781-797.

Lethbridge T.C., Forward, A. & Badreddin, O. 2012, *, Umple language online.* Available: http://try.umple.org [2013, October/20].

Lethbridge, T.C., Forward, A. & Badreddin, O. 2012, *Umple Google Code project*.

Lethbridge, T.C., Forward, A. & Badreddin, O. 2010, "Umplification: Refactoring to Incrementally Add Abstraction to a Program", *17th Working Conference on Reverse Engineering (WCRE), 2010*, pp. 220.

Lethbridge, T.C., Mussbacher, G., Forward, A. & Badreddin, O. 2011, "Teaching UML using umple: Applying model-oriented programming in the classroom", *Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training, IEEE Computer Society*, pp. 421.

Mehner, K. 2002, "JaVis: A UML-Based Visualization and Debugging Environment for Concurrent Java Programs", *Revised Lectures on Software Visualization, International Seminar, Springer-Verlag*, pp. 163.

Selic, B. 2003, "The pragmatics of model-driven development", *IEEE Software,* vol. 20, no. 5, pp. 19-25.

Sendall, S. & Kozaczynski, W. 2003, "Model transformation: the heart and soul of model-driven software development", *IEEE Software,* vol. 20, no. 5, pp. 42-45.

Timothy C. Lethbridge, Gunter Mussbacher, Andrew Forward & Omar Badreddin 2011, "Teaching UML Using Umple: Applying Model-Oriented Programming in the Classroom", *CSEE&T,* , pp. 421-428.

APPENDIX

```
// **** Description
// In this Appendix, we present the
// Umple code written to model the
// Student registration system
// **** Information
// 4 classes
// 1 State Machine
class Course {
  code;
  description;
  1 -- * CourseSection;
}

class CourseSection
{
  sectionId;
  Integer classSize = 0;
  Integer minimumClassSize = 10;
  Integer maximumClassSize = 100;

  // State machine
  CourseSectionStm
  {
   Planned
   {
     openRegistration ->
     OpenNotEnoughStudents;
   }
   OpenNotEnoughStudents
   {
     closeRegistration -> Cancelled;
     cancel -> Cancelled;
     register [getClassSize() >
          getMinimumClassSize()]
         -> OpenEnoughStudents;
     }
     OpenEnoughStudents {
         closeRegistration -> Closed;
         cancel -> Cancelled;
         register [getClassSize() >
         getMaximumClassSize()] ->
         Closed;
     }
     Cancelled {}
     Closed {}
   }

  // Code mixins
  boolean requestToRegister(Student
  aStudent)
  {
    register();
    setClassSize(getClassSize() + 1);
  }
}

class Student {}

class Registration {
  grade;
  * -- 1 CourseSection;
  * -- 1 Student;
}
```