ZOOM - Automatic Code Generation: UML State Charts to Executable Java Code
Meta-model, translation engine and FSM framework design

Anoop Mathew, Michael Montenegro
DePaul University
School of Computer Science
Chicago, IL
ajmsuper@hotmail.com
mhm4@lucent.com

Abstract

Design of software that is of high quality, easily extensible and reusable are key requirements of any software project. Dynamic code generation has been touted as a promising emerging technique for achieving these software development goals. The overall goals of this project were to develop a finite state machine (fsm) framework and tools to specify and translate ZOOM behavioral models i.e. UML state charts and state implementations into executable Java code.

1. Introduction

ZOOM stands for Z-Object Oriented Modeling. It is based on the Z formal specification method. Z in its original form does not support object oriented constructs. It is more of a mathematical specification. ZOOM project headed by Dr. Xiaoping Jia aims to extend Z to add object-oriented constructs. The project aims to add a formal aspect to object oriented design using UML.

ZOOM supports specification of three (3) kinds of models. These are the structural model, the behavioral model and the user interface model. The structural model specifies the static parts of the system including class relationships and associations. The user interface model allows for the formal specification of the user interface. This allows for creation of user interfaces that are formal in nature and platform independent. The behavioral model describes the dynamic aspects of the system. It links the structural models and the user interface models. The behavioral model is specified using a finite state machine (FSM). The FSM follows the UML semantics for state machines [7] i.e. it is described using UML state charts.

Events are used to communicate between the different formal models. Events drive the system. The UI models generate events when a user interacts with the software system. This triggers finite state machine events that cause an operation to occur within the system [7].

Our goals as part of the ZOOM project were to develop a translation engine, meta-model and FSM framework to allow users to specify and execute the ZOOM behavioral models described above. In this paper, we start with a brief introduction to model driven architecture and UML state charts. We then cover the design of the project components, the advantages of our approach, comparison with other technologies, and the possibilities for future work related to this project.
2. Model Driven Architecture
MDA stands for model driven architecture. It is a standard approach to designing software systems developed by the Object Management Group (OMG). One of the main goals of MDA is to allow separation of application software design from implementation considerations such as programming languages, platforms, and operating systems. It promotes application software reuse. This would allow for a software engineer to benefit from the positives of various technologies without having to directly interact and understand their syntax, idiosyncrasies, or pitfalls. Implementation detail separation from business logic makes applications more robust [9].

Software development using MDA usually involves the following steps. First, a platform independent model (PIM) of the application software is created. This model is described using UML notations. Next, a platform specific model (PSM) is created to map the PIM to a specific platform. The PSM is also usually described using UML notations. The PSM can be created from the PIM using model generation tools. Once the PSM has been created, code generation tools can create code for that platform from the PSM [9].

Model driven architecture can provide more stable and secure applications because of the ability to alleviate the dependency of having the software engineer address language specific bugs, improper coding practices, and security issues that can be often overlooked. Applications can also be easily supported on future technologies that may not yet be available. This becomes possible by the platform independent model approach. The business logic is not associated with a specific technology so translation into a future technology would allow for applications developed in a model driven architecture to be translated into these future technologies. Since UML is the unifying factor in model driven architecture, all development groups familiar with it can communicate ideas and applications more effectively leading to an increase in productivity. Also, a reduction in time is achieved to demo and alter application functionality making applications easier to tailor to customer requests.

UML is used as the common denominator for all applications developed on a model driven architecture. This allows for a general non-technology specific syntax to describe an application’s functionality. The UML entities are typically pictorially based and altered using graphical interfaces. The UML syntax details are included in the UML specification that implementers of model driven architecture tools must follow to be compliant with the Unified Modeling Language.

3. UML State Charts
State charts diagrams are used widely in real-time and embedded software development to specify system behavior. They are used to specify object behavior that is dynamic, i.e. object behavior that changes depending on the state of the object. State charts show the states that an object may assume and the transitions (change in state) the object may make from state to state. There are three (3) major components to state charts. These are states, events and transitions.

3.1. States
A state represents a stage in the lifecycle of an object. States usually last some finite duration. There are various types of states defined in the UML model [4]. These are as follows:

- Simple states – simple states are states that do not contain other states.
- Composite states – these are states that contain other states. Composite state can be non-concurrent or concurrent. Non-concurrent state is defined as a state that contains exactly
one state region. Concurrent composite state is defined as a state where there are two or more state regions that execute independently of each other. States contained within composite states are usually referred to as sub-states.

- **Final State** – final state represents an object's completion of activity. In instance-based state charts, it usually signifies that the instance ceases to exist at this point. Final states cannot have outgoing transitions from them. Diagrammatically, it is represented as a bull's eye.

- **Initial State** – is a state in which an object spends no time. Diagrammatically, it is represented as a black dot. Initial state is a type of pseudostate. A pseudostate is an abstraction used to represent transient states. Pseudostates are usually used to connect complex transitions. There can be at most one initial state in a composite state.

- **Deep History State** – a pseudostate, is used to represent the last known active state configuration of the composite state that contains the history state. Deep history is represented diagrammatically as a circle with an “H” symbol within it. Deep history implies the active state configuration for all levels of the state configuration underneath the composite state are saved when the composite state is exited and restored when the composite state is reentered. If no history exits (i.e., the composite state is entered for the very first time), a default transition is taken. This default transition is diagrammatically represented as an outgoing transition from the history state to a simple or composite state. Deep history is only applicable within composite states and there can be at most one history state per composite state.

- **Shallow History State** - a pseudostate, is used to represent the last known active state configuration of the composite state that contains the history state. Shallow history is represented diagrammatically as a circle with an “H” symbol within it. Shallow history implies the active state configuration for only the first level of the state configuration underneath the composite state is saved when the composite state is exited and restored when the composite state is reentered. If no history exits (i.e., the composite state is entered for the very first time), a default transition is taken. This default transition is diagrammatically represented as an outgoing transition from the history state to a simple or composite state. Shallow history is only applicable within composite states and there can be at most one history state per composite state.

- **Synch State** – synch state is used to synchronize concurrent states. Synch states are used to ensure that transitions to a target state out of concurrent states or regions that are synchronized do not occur until all the outgoing transitions from the synchronized concurrent states are triggered. When this condition is satisfied, the transition to the target state is triggered automatically. Synch states are only applicable to composite concurrent states.

Simple and composite states can have actions or procedures associated with them. Actions are performed as a result of state entry or exit. These actions are optional in the specifications. There are 3 kinds of state actions defined within the UML model. These are as follows:

- **Entry action** – an action that is performed when a state entry occurs. It is performed prior to any other action within the state. The action is performed regardless of the transition that triggered the entry into the state.

- **DoActivity action** – an action that is performed while the state is active. The action is terminated when the state is exited.
3.2. Events
An event is defined as an abstraction of an incident or signal in the real world that causes a change in state [8]. Generation of events results in transitions. Events may have supplemental data (parameters) associated with it. There are four (4) types of events defined in the UML model [4]. These are:

- **ChangeEvent** – models an event that occurs when an explicit Boolean expression becomes true. Change events are also called predicate or Boolean events. It usually indicates a change in object attribute value or association. Change events are expressed syntactically using `when(booleanExpression)`. The state machine raises change events implicitly.
- **TimeEvent** – models the expiration of a time deadline. Time events are expressed syntactically using `after(timePeriod)`. The time period starts when the state machine enters a state that has an outgoing time event defined from it. The state machine raises time events implicitly once the time period expires. The time event evaluation is cancelled if the state machine changes its state.
- **SignalEvent** – models the occurrence of an incident in the real world. Signals are asynchronous in nature. Objects use signals to communicate with other objects [2]. A Signal is abstracted as a SignalEvent.
- **CallEvent** – models a synchronous event that triggers a specific operation on the recipient object’s state machine. Two cases of call events are creation and deletion events.

An event can be deferred in a state. A deferred event is one that does not cause a transition in the current state. Deferred events stay on the state machine’s event queue until the event is not deferred anymore (in which case it’s thrown away) or until the event causes a transition.

3.3. Transitions
A transition is change in the current state configuration from a source state to a destination state. The source and/or destination states could be pseudostates. A transition is triggered as a result of an event. There can be at most one event that triggers a particular transition. Transitions may optionally have guards associated with them. A guard is a Boolean expression. If a guard is specified, the transition is only taken if the Boolean expression evaluates to true at the time the event is triggered. Transitions can also have optional actions/procedures associated with them. These are called transition actions. If an action is specified, the action is executed during the transition.

As part of the ZOOM FSM framework design, an extension was added to the transition behavior in UML. The framework added support for the concept of transition priority. Transition priority is used to select between multiple events on an object’s event queue that could potentially cause a transition. Transition priority is specified as an integer value.

4. Project Architecture
The diagram below (figure 1) describes the components designed to support the behavioral model aspects of the ZOOM project and the relationship between these components.

As the project architecture diagram describes above, the purpose of the project was to provide tools and a framework to allow users to specify the behavioral models in ZOOM as finite state machines and provide support to translate these models into java source code and to be able to fully execute these models. The project was also MDA compliant. The various components of the project are described below.

4.1. State Chart Description File (fsm)
State chart description files or fsm files are the textual representation of a graphical UML state chart model. It represents the PIM. It contains the dynamic aspects of UML state charts. It supports the specification of states, transitions, signal events, deferred events, synchronization, state history, transition actions, guards, transition priority, composite states, concurrent states, time events, change events, initial states, and final states. It is described using the state chart specification syntax. The description is very similar to java syntax. The fsm file for a given state chart is the first component that a user supplies to the translation process. Figure 2 below is an example of a state chart description

```java
fsm SellableItem()
{
    state received;
    state inInspection;
}```
4.2. ZOOM AST Parser
The ZOOM Translation Engine, like many other ZOOM sub-projects depend on the ZOOM AST parser. This parser takes the fsm input file to generate an abstract syntax tree, or AST, object. This object includes all the data about the fsm file in a hierarchical tree data structure. In order to generate this file the user needs to be familiar with UML state charts and the BNF syntax to accurately formulate the state machine’s functionality. The Backus-Naur Form describes how to specify the syntax of the fsm file to describe the state chart. There are two general structures that nest a majority of the data in the AST tree. The state element describes a state name, its sub states, and regions. The transition elements includes sub elements that are related such as, transition from and to states, events, guards, triggers, and synchronization. The ZOOM AST parser produces the AST object in which the translation engine uses to generate the code.

4.3. ZOOM FSM MetaModel
A metamodel is defined as the model of a language expressed as a modeling language [2]. The metamodel designed as part of this project described the structure of state charts, states, transitions and events and the relationship between these entities. The metamodel was described using a UML class and association diagram. The metamodel for the project was designed using the OMG-UML Semantics v.1.5 state machine and event abstract syntax as the base model and adding enhancements and customizations unique to the project. The customizations added included support for the concept of priority for transitions as well as ports associated with transitions. The metamodel is an input to the translation engine. It is used as the template to create the model for the state chart that is described in the fsm file supplied by a user.

4.4. ZOOM Translation Engine
Before the code can be generated, the AST object needs to be parsed to gather all the relevant info needed for code generation. The translation engine recursively walks the tree and stores info described in the metamodel into a library. Since there are two general types of data in a state chart, there are two recursive methods that build the library. The steps of the library building are sequential given that the state chart is grouped properly. The processing would follow as such: for each state that is encountered, a loop would follow that would get all the elements of that object. On certain elements that are deemed vital to code generation, such as a state name, it would be inserted into the library. Once a transition was hit another recursive method would be called to traverse the transition elements populating all the vital elements into the library. There is a close relationship between the tree parser and the BNF. Once the library, or the PSM was built, code generation can begin. The code generation follows sets of rules that generate custom code based on what types of elements of objects are contained within the library. The rules specify how the ZOOM FSM framework is expecting to instantiate the classes and call its methods. Once the code is generated, it can be compiled and invoked through the
ZOOM FSM framework. The framework controls program execution based on the fsm state chart logic.

4.5. ZOOM Finite State Machine Framework
A framework can be described as a semi-complete application that users extend to achieve their goals. Frameworks by design are flexible and easily extendable. Frameworks also retain top-level control of the application (inversion of control). The finite state machine software framework designed as part of this project provides two specific functions. These are to allow the user to specify the state implementations associated with a state chart and to allow the execution of the state machines once the source code has been generated.

The ZOOM FSM framework has two clients. One is the user who uses the framework APIs to specify the state implementations. The second is the translation engine that generates the java source code based on the framework API.

As described above, the finite state machine framework allows a user to specify UML state chart semantics as well as providing support for state machine life cycle management and execution. The framework supports the following UML state chart concepts:

- **States** – the framework provides support for specifying a state (simple, composite, or concurrent) and the action semantics associated with states. Specification of Entry, DoActivity, and Exit actions is supported. The framework also provides support for specifying initial and final states.
- **Events** – the framework provides support for specifying events, including time, signal, and change events and the supplemental data associated with events.
- **Transitions** – the framework provides support for specifying transitions and the data associated with transitions. These include transition guards, transition actions and transition priority. The framework also provides support for specifying initial transitions.
- **State Entry** – the framework provides support for all state entry types including default, explicit, shallow history and deep history.
- **Synch States** – the framework provides support for synchronizing multiple regions of a concurrent state.
- **Deferred Events** – the framework provides support for deferring events in a given state and management of deferred events during state machine execution.
- **State History** – the framework provides support for saving state history. Both shallow and deep histories are supported. The framework manages saving and restoration of the state configuration when state history is specified.
- **State Machine Lifecycle Management** – the framework provides support for creation and deletion of state machine instances.
- **State Machine Event Management** – the framework provides support for event creation, posting (queuing), de-queuing and dispatch to state machine instances. The framework ensures that event management is performed in a thread-safe manner.
- **State Machine Execution** - the framework also provides support for threading to allow parallel execution of state machines. It also support run-to-completion for state actions. This means that one state action completes execution before other state actions are allowed to run within the same state machine instance.
• State Machine Selection – the framework provides support for lookup of any active instance of a state machine either given its name or its primary key (if defined). The framework also provides support for lookup of many instances of a given state machine.

4.6. State Action Semantics

State action semantics are the second user supplied component to the translation process. A java source file, it contains the FSM framework based implementations of the actions associated with the states described in the fsm file. The user uses the framework API to specify the Entry, Exit, and DoActivity actions for each state.

5. Benefits

Some of the benefits of the translation engine and the automatic code generation process include the following. Since state interaction is handled outside of manual coding, it makes it easier to change program logic through the fsm file and automatically generate new code. The generated code is also more resilient to bugs since it is being automatically generated and idiosyncrasies of the Java language that can produce bugs are addressed at the code generation level. Program logic is structured at a higher level, with UML, making the specific programming language independent. It can be possible to support additional languages, both present and future, that the code generator can produce.

The ZOOM FSM framework offers many benefits to software developers. Some of these are as follows.

• Ease of use – the framework is relatively easy to learn and use. A software developer is freed from the complex mechanisms needed to setup and run state machines. This allows the developer to focus on solving the business problem at hand.

• Support for complex state machine concepts – the framework supports advanced state machine concepts like nested and concurrent states, state history, deferred events, state/region synchronization etc. This allows for development of complex state charts.

• Supports thread safe state machine execution – since threading behavior for state machines is built into the framework, it supports thread-safe and parallel execution of state machines.

• Exception handling support – the framework supports exception handling for state machine run-time exceptions allowing users to recover from unexpected system behavior.

• Logging support – the framework provides support for logging capabilities that makes debugging of state machine behavior easier.

• Platform Independence – the framework is java-based and hence benefits from the advantages of the java language including platform independence and portability.

• Execution Capability – the execution capability of the framework allows UML state charts to be turned into executable Java code. This speeds up software development time, since code development time is significantly reduced.

6. Comparison to Executable UML

Executable UML (xUML) was one of the technologies researched in the initial phase of the project. xUML is an MDA compliant methodology that aims to make UML models executable. It is an automated object oriented methodology that uses a limited set of the UML constructs. xUML emphasizes the separation of design from implementation. Design using xUML
methodology involves creating models that describe the application. Minimally, these models include the UML class diagram for the domain(s) under study, as well as UML state chart diagrams for active objects within the class diagrams [2,5]. The actions that the objects perform as they change state is specified as well. These models are the PIM. A model compiler turns these models into a specific implementation based on rules specific to an implementation. These rules are called archetypes.

Comparing xUML to ZOOM, we can see that they both follow similar approaches to supporting MDA. Both methodologies use UML models to describe a system. ZOOM uses structural and behavioral models while xUML uses class diagrams and state chart diagrams. They both also use the concept of UML state charts to specify the behavior of the system. They both also support automatic generation of executable applications from these models.

The differences between these approaches are as follows. While xUML only supports a limited set of UML state chart constructs, the ZOOM behavioral models support a rich set including composite/concurrent states, state history, and state synchronization. ZOOM uses the Java language to specify the state actions while xUML uses Action Specification Language (ASL) that varies from one vendor’s implementation to another. Finally, xUML has vendor support for graphical tools to specify the models while ZOOM behavioral models are text-based.

7. Future Work
This team was the first to work on the behavioral model aspects of the ZOOM project. The functionality that we provided created opportunities for additional enhancements. Some of these include run-time performance enhancements to the FSM framework, and enhancing the logging and exception handling capabilities of the framework. Other enhancements could include adding support for distributed computing environments and support for the Java 2 Micro Edition (J2ME) platform to the FSM framework.

The ZOOM behavioral model automatic code generation project could also benefit from adding support for Ant and Junit. Since the project is responsible for generating a fully functional application, it could be beneficial to add a dynamic generation of an Ant script to build other included files, compile it and run Junit tests. Junit tests would be automatically generated as well to test problem areas of an application such as, external interfaces and its input and output. They can also support regression testing of existing functionality to verify that no changes have broken any other areas. This would remove more of the manual steps that are associated with the translation process.

Another large enhancement would be the support of multiple languages in the translation engine. We are currently supporting only Java and have kept the opportunity to add other languages in mind while developing the project. One of the benefits of the model driven architecture is that it depends on a non-programming language specific syntax to describe program functionality. The model generator and code generator are java classes. The model generator would not need to change because it parses UML. The code generator would need to be altered to support reading and analyzing of a .Net or other programming language skeleton class and convert the framework rules for building a java class into another language class. It may then be possible to provide a conversion from one skeleton class language to another. This would mean that a user could input an fsm file and a Java skeleton class and generate a .Net class. It makes future programming languages irrelevant because software engineers do not have to be concerned with programming languages and re-factoring code to address new technologies or be affected by the often steep learning curve new languages bring. All the individual
language support would be contained within the code generator and software engineers would continue to use the tool in the same fashion.

8. Conclusion
Model Driven Architecture and automatic code generation has the potential to revolutionize the software development world. Many companies are beginning to see the benefits of UML model based application software development and the ability to automatically generate source code from these models. Companies across the industry are using it for various applications on embedded device platforms, networking components, and, various operating systems. These applications can now benefit from translation into other programming languages, greater reliability, and quicker prototyping to name a few advantages over traditional software development.

ZOOM adds a formal foundation to the MDA approach to software development. The behavioral model components we developed, being part of the ZOOM approach, hence benefit from this formal foundation and facilitate the design of software that is of high quality, easily extensible and more reusable.

References
   By Hongming Liu, Lizhang Qin, Chris Jones, Xiaoping Jia