Ruben Campos
Candidate

Computer Science
Field of Concentration

TITLE:  Model Based Programming: Executable UML with Sequence Diagrams

Russell J. Abbott, Ph.D.
Faculty Member
Signature

Behzad Parviz, Ph.D
Faculty Member
Signature

Raj S. Pamula, Ph.D.
Department Chairperson
Signature

DATE: _____________________
MODEL BASED PROGRAMMING: EXECUTABLE UML WITH SEQUENCE DIAGRAMS

A Thesis
Presented to
The Faculty of the Computer Science Department
California State University, Los Angeles

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

By
Ruben Campos
June 2007
Acknowledgements

First and foremost I want to thank my Lord and Savior Jesus Christ for giving me the strength to complete this portion of my educational journey. I want to especially thank my wife Nelly Campos and my children, Jose, Alma, and Raul for their love, support, understanding, and patience.

I would like to give special thanks to my mother, who raised 10 children including myself. I thank my father for all his sacrifices and the legacy he left behind for my children and I to emulate. I thank my siblings for their support as well.

I would like to thank Dr Abbott for his blunt advice and supervision of my graduate work. Specifically, I appreciate Dr. Abbott’s honesty in recommending a new direction for my thesis work after taking time to listen to my academic goals.

I would like to thank all the other faculty members in the Computer Science Department for their instruction in their respective courses. In particular, I would like to thank Dr. Sun for his tough general advice during his courses, which helped motivate me to excel academically. Also, I want to thank Tricia Trejo for putting up with my quarterly phone calls regarding administrative matters.

Last but not least, I would like to thank my fellow graduate students especially Robert Ritchey with whom I collaborated on several projects and presentations.
Abstract

Model Based Programming: Executable UML with Sequence Diagrams

By

Ruben Campos

The history of programming can be described in terms of how the level of abstraction has been raised with each new generation of programming. Whether its going from Assembly Language to the C programming language or going from Procedural languages to Object-Oriented programming, the transition between programming generations always seems to be defined by the higher level of abstraction that was achieved. With the OMG pushing the Model Driven Architecture using OMG standards such as the UML, the next logical generational shift is a transition from Object-Oriented Programming to Model Based Programming.

The concept of Executable UML appears to provide the most potential to become the core for Model Based Programming. Although an official Executable UML Specification is still a work in progress, there have been attempts to define Executable UML. One such attempt is by Mellor and Balcer, who have designed a rendition of Executable UML albeit with design features that would not be compatible with true model based programming. This paper describes how an alternative Executable UML design can achieve true model based programming.
Table of Contents

Acknowledgements ........................................................................................................................................... iii

Abstract ......................................................................................................................................................... iv

List of Tables .................................................................................................................................................. viii

List of Figures ................................................................................................................................................ ix

Chapter 1 - Introduction ............................................................................................................................ 1

Chapter 2 - History of the UML .................................................................................................................. 5

  2.1 Booch Contribution ................................................................................................................................. 6

  2.2 Rumbaugh Contribution ......................................................................................................................... 7

  2.3 Jacobson Contribution ............................................................................................................................. 8

  2.4 Rational Contribution ............................................................................................................................ 8

  2.5 OMG Contribution ............................................................................................................................... 9

Chapter 3 - UML Semantics Problems ...................................................................................................... 10

  3.1 Class Diagram Semantics problems ....................................................................................................... 10

  3.2 Sequence Diagram Semantics problems ............................................................................................... 12

  3.3 Stereotype Semantics Problems ........................................................................................................... 13

Chapter 4 - MDA Initiative ......................................................................................................................... 15

Chapter 5 - OMG xUML Specifications ..................................................................................................... 18

  5.1 Objectives of Specifications .................................................................................................................. 19

  5.2 Executable UML Architecture ............................................................................................................ 19

Chapter 6 - Balcer and Mellor xUML Design ......................................................................................... 22
6.1 Class Diagram ................................................................................................... 24
6.2 State Modeling .................................................................................................. 26
6.3 Defining State Actions ...................................................................................... 27
6.4 Model Execution ............................................................................................... 27

Chapter 7 - Balcer and Mellor xUML Design Flaws .............................................. 29
7.1 Reduced role of Class Diagram................................................................. 29
7.2 Use of State Diagram .................................................................................... 30
7.3 Action Language .......................................................................................... 30
7.4 PIM to PSM Translation .............................................................................. 31

Chapter 8 - Alternative xUML Engine for Model-Based Programming .................. 32
8.1 Iterative Process for Building an xUML Model ............................................. 32
  8.1.1 Use Case Modeling .................................................................................. 34
  8.1.2 Create Sequence Diagram ....................................................................... 35
  8.1.3 Create Class Diagram ............................................................................ 37
  8.1.4 Define Class Methods Using Activity Diagrams ..................................... 37
  8.1.5 Compile/Execute Sequence Diagram using JVM ................................... 38
8.2 Assumed xUML Diagram Semantics .............................................................. 40
  8.2.1 Sequence Diagram Semantics ............................................................... 40
  8.2.2 Class Diagram Semantics ....................................................................... 42
  8.2.3 Activity Diagram Semantics ..................................................................... 45
8.3 xUML Engine Architecture ............................................................................. 46
8.3.1 XMI Format.............................................................................................................47
8.3.2 xUML Model Translator .......................................................................................48
Chapter 9 - Future Work ..............................................................................................49
Chapter 10 - Conclusion...............................................................................................50
References ....................................................................................................................51
List of Tables

Table 8.1 - Sequence Diagram Semantics ................................................................. 41
Table 8.2 - Class Diagram Semantics ................................................................. 43
Table 8.3 - Activity Diagram Semantics ............................................................... 45
List of Figures

Figure 2.1 - UML Timeline........................................................................................................ 5
Figure 2.2 - Booch Method Class Diagram ............................................................................. 6
Figure 2.3 - OMT Object Diagram........................................................................................... 7
Figure 2.4 - OMT State Diagram ............................................................................................ 8
Figure 3.1 - Multiple Uses for Class Diagrams....................................................................... 11
Figure 3.2 - Loose semantics in Class Relationships.............................................................. 12
Figure 3.3 - Loose Semantics in Sequence Diagram .............................................................. 13
Figure 3.4 - Stereotype Example............................................................................................. 14
Figure 4.1 - Model Driven Architecture .................................................................................. 15
Figure 4.2 - PIM to PSM mappings ....................................................................................... 17
Figure 5.1 - xUML Architecture ............................................................................................ 20
Figure 6.1 - Balcer/Mellor xUML Process............................................................................. 23
Figure 6.2 - xUML Class, State Machine, and Action Language .......................................... 24
Figure 6.3 - xUML Class Diagram ....................................................................................... 25
Figure 6.4 - xUML State Machine Diagram ......................................................................... 26
Figure 6.5- xUML Model Execution in iUMLLite................................................................. 28
Figure 8.1 - Proposed xUML Modeling Process................................................................. 33
Figure 8.2 - Banking Use Case Example ............................................................................... 34
Figure 8.3 - Example Sequence Diagram ............................................................................. 36
Figure 8.4 - Example xUML Activity Diagram ..................................................................... 38
Figure 8.5 - Semantically correct Sequence Diagram 1 from IBM Website .................... 42
Figure 8.6 - Semantically correct Sequence Diagram 2 from IBM Website .................... 42
Figure 8.7 - Semantically Correct Class Diagram............................................................. 44
Figure 8.8 - Semantically Correct Class Definition.......................................................... 44
Figure 8.9 - Semantically Correct Activity Diagram ........................................................ 45
Figure 8.10 - xUML Architecture ..................................................................................... 46
Figure 8.11 - IBM site XMI Example ............................................................................... 47
Chapter 1 - Introduction

The replacement of the term programmer with the term software developer is indicative of the current level of abstraction being enjoyed by the modern programmer. Throughout the history of computer programming, the perpetual goal has been to make programming easier by making it more abstract. The motivation behind abstraction has always been to reduce the amount of low-level instructions that a programmer has to implement in order to arrive at a solution more expediently. So as programming has become more abstract, so have the academic areas of study surrounding computer programming. Programming has becoming less of a mathematical task and more of an Engineering task. As a result Software Engineering has emerged as a key area of study within Computer Science.

Currently, two major sub topics within Software Engineering are Object-Oriented Programming and Software modeling. The emergence of Object-Oriented programming has heavily contributed toward a standardized method of modeling known as the Unified Modeling Language (UML). In recent years, UML has become synonymous with software modeling and is commonly used to drive the design and implementation of software architectures and systems. Furthermore, UML tools have enabled the creation of source code from UML diagrams in order to initiate the programming phase of building software.

Considering all of the advancements made in software modeling, the pure abstractionist would ask, “Why not model-based programming”? In fact, some have
begun to tackle the problem of raising the level of programming abstraction once again in the form of model-based programming. Naturally, many have looked to standard UML as the most viable candidate for providing the basis for model-based programming. However, due to a lack of clearly defined semantics, it has been challenging to leverage standard UML for model-based programming. The Object Management Group (OMG) [1] has recognized that standard UML lacks the semantics necessary to achieve a higher level of abstraction. As a result, the OMG has in recent years designed a conceptual architecture known as the Model Driven Architecture (MDA) [2]. A big part of this initiative is Executable UML (xUML) [3], which adds execution facilities to UML. Executing a model representation of a system would allow a Software Engineer to verify the functionality of a software system before deciding on what platform to deploy the software to. In other words, xUML would allow for the implementation of model-based programs that inject functionality into UML models. Although a standard definition of xUML is still a work in progress, one of the main goals of the xUML specification [4] is to establish clear semantics so that UML models can become computationally executable.

Prior to the inception of the xUML draft specification, Mellor and Balcer [5] designed one possible manifestation of xUML. Their idea of xUML is consistent with the xUML specification in that it clearly defines the semantics of some UML diagrams, thus allowing a UML model to be compiled and executed. According to Mellor and Balcer, an xUML model only requires a small subset of semantically well-defined UML
diagrams in order to produce an executable model. In particular, the Class Diagram and the State Machine Diagram are the only required models in their xUML design. In addition, an action language based on OMG’s Action Semantics standard [6] is used to drive the execution of a model via state machine diagrams.

Although a step in the right direction toward fulfilling the MDA initiative, the Mellor/Balcer xUML engine falls short of providing a foundation for true model-based programming due to some design flaws. Among the design flaws is the use of State Diagrams for model execution. State diagrams seem to be an awkward choice to provide model execution as they were intended to give an introspective view of an object’s lifecycle. Furthermore, state diagrams fail to provide a graphical picture of how objects interact with each other in order to achieve model execution. Another design flaw is the use of an action language that seems to be more procedural rather than Object-Oriented. Ironically, rather than using UML artifacts to achieve model-based programming, an action language reminiscent of an older generation programming language is used. A third design flaw is the reduced role of the Class diagram. The Mellor/Balcer design seems to move away from the Object-Oriented programming roots of UML. Specifically, the definition of Class methods are disregarded and replaced by the definition of State Actions using the aforementioned action language. The last of the design flaws identified is the unnecessary complexity of converting a platform independent model into a platform specific model.
This paper will expound on the flaws in the Balcer/Mellor design and propose an alternative xUML Engine design. The proposed xUML Engine will be more consistent with raising the level of programming abstraction to true model-based programming. The basic idea behind the alternative design is that no text-based programming should be necessary for true model-based programming. In other words, the xUML engine will internalize the details behind translating UML models into a text-based program. The proposed xUML engine is comprised of the UML class, sequence, and activity diagrams in conjunction with the Java language. The combination of these components will allow for platform independent model-based-programming.

Before delving into the design details of the proposed xUML engine, a relevant historical and technical background is provided. In particular, this paper will highlight the relevant semantic problems that have prevented standard UML from being executable. In addition, a description of the official xUML draft specification is given to provide insight into how the semantic problems are being addressed by the OMG. Furthermore, an overview of the MDA initiative will reveal the important role that model-based programming will play in the future of Software Engineering.
Chapter 2 - History of the UML

The UML is the product of different modeling projects that were eventually consolidated to form a “unified” modeling language called the “Unified Modeling Language”. The following figure illustrates the work along with its contributors that has led to the current official UML version.

Figure 2.1 - UML Timeline
2.1 Booch Contribution

While at Rational Software, which is now part of IBM, Gary Booch developed an Object Modeling Language and called it the Booch Method [7]. Some of the modeling symbols present in the Booch Method can still be found in the UML. The following illustration shows a sample class diagram as represented by the Booch Method:

![Figure 2.2 - Booch Method Class Diagram](image-url)
2.2 Rumbaugh Contribution

The OMT (Object Modeling Technique) is a modeling language developed in large part by James Rumbaugh while at General Electronic Research and Development Center [8]. A close observation of the symbols, object diagram, and state diagram in the OMT examples below bears many similarities to what can be found in the UML today.

![Figure 2.3 - OMT Object Diagram](image-url)
2.3 Jacobson Contribution

OOSE (Object-Oriented Software Engineering) was developed by Ivar Jacobson in 1992, while at Objectory AB, a Swedish-based software company [9]. OOSE featured the concept of Use Cases that was to drive software design. The Use Case diagram remains one of the mainstays of the UML.

2.4 Rational Contribution

Rational Software Company was able to merge the minds of the three major contributors to Object-Oriented Software Modeling in Booch, Rumbaugh, and Jacobsen. Rational formed the hub to what drove the creation of the Unified Modeling Language or the UML. The first major version was 0.8 and reflected the work from Booch and Rumbaugh as shown in Figure 2.1 [10]. The next major release was version 0.9 which included the contributions by Ivar Jacobson [11,12].
2.5 OMG Contribution

The Object Management Group (OMG) is a Consortium that focuses on Object-Oriented processes, modeling, and standards [1]. OMG members include Hewlett Packard, IBM, and Sun Microsystems. In 1996 the OMG put out an RFP (Request for Proposal) seeking an object modeling language. Rational was in the best position to fulfill the RFP having merged the talents of Booch, Rumbaugh, and Jacobsen. It is after this point in time that UML got submitted as the proposed solution to the RFP. Since then, the UML did not only fulfill the original RFP but has also evolved into new versions of UML. Currently UML 2.0 is the latest official version that is OMG approved although UML 2.1 is already being used in the industry [13].
Chapter 3 - UML Semantics Problems

The inability to breathe life into standard UML models is directly due to a lack of concrete, organized semantics in the UML modeling language. The history behind UML described in the previous Chapter gives some insight as to why there is a lack of clearly defined semantics in the UML. With many contributors adding an aspect of their modeling techniques to the “Unified” Modeling language it is no surprise that some UML diagrams can have different uses and interpretations. The result of this has been that behavioral and/or run-time semantics are not well defined for standard UML modeling elements. In order to remedy the semantics problems many research projects have sprung up to retrofit semantics into standard UML modeling elements [14]. This chapter focuses on the semantics problems that have made it difficult to achieve model-based programming using standard UML.

3.1 Class Diagram Semantics problems

Figure 3.1 illustrates how Class Diagrams can be used for analysis and design modeling. Although this gives the modeler flexibility, it makes it hard to define clear semantics. Figure 3.1 shows how the analysis phase modeling uses a more informal English format, while the design phase modeling uses a more formal naming semantics. In other words, a modeler can use the Class diagram in ways that make it hard to derive computational patterns.
Similarly, loose semantics in Class relationships and multiplicities allow the modeler flexibility, but make it impossible to bring a computational consistency to UML Models. Figure 3.2 shows a high level Class Diagram with relationships and multiplicities. Note that the relationships are not uniquely identified, thus making it difficult to apply unique constraints on the relationships. In particular, notice that there are 2 associations that contain the verbiage “lives at”.

**Figure 3.1 - Multiple Uses for Class Diagrams**
3.2 Sequence Diagram Semantics problems

The sequence diagram is best suited to represent the object interactions involved in a Use Case Scenario. This diagram also happens to be a good candidate for executing a UML model. Unfortunately, semantic variations of how certain components in the Sequence Diagram can be used prevent it from being executable. Figure 3.3 shows an example of a sequence diagram.
In this example alone there are a few semantic problems present. When the Student invokes the “wish to enroll” method from the EnrollSeminar Object, it is in an English like fashion. However the SecurityLogin Object calls the isValid function from the Student Object in a manner that is more consistent with Object Oriented Programming. Another semantic variation point is the optional use of a return value from a method call. For instance, the isValid method returns “yes” whereas the getAvailable method does not return anything.

3.3 Stereotype Semantics Problems

Stereotypes present a quick and flexible way of introducing custom modeling elements to the UML. The versatility of the stereotype is evident in how it is used in different modeling elements and diagrams. In fact, some UML core modeling elements
such as the Component were created using stereotypes. However, the UML does not have strict constraints as to how a stereotype can be used and as a result it introduces even more semantics problems to standard UML. An extreme example of how the stereotype can be used is shown in Figure 3.4.

![Figure 3.4 - Stereotype Example](image)

Although practically speaking a system modeler would not typically create such a model, there is nothing in standard UML that prevents this from occurring. A system modeler can create a limitless amount of stereotypes without violating any modeling rules. Figure 3.4 shows how stereotypes did not provide clarity to the diagram, yet did not violate UML syntax. In short, stereotypes only exasperate the problem of loose semantics in standard UML.
Chapter 4 - MDA Initiative

According to the Object Management Group (OMG), the MDA separates business and application logic from underlying platform technology [2]. The OMG aims to achieve this by using current OMG standards, such as UML, to build the static and behavioral aspects of a system through platform independent models. As Figure 4.1 illustrates, models based on OMG standards are at the core of the MDA. Using model transformations, the models can become platform specific implementations, such as .Net or Java.

Figure 4.1 - Model Driven Architecture
According to IBM [15], four principles underlie the OMG's view of MDA:

- Models expressed in a well-defined notation are cornerstones to understanding systems for enterprise-scale solutions.
- The building of systems can be organized around a set of models by imposing a series of transformations between models, organized into an architectural framework of layers and transformations.
- A formal underpinning for describing models in a set of metamodels facilitates meaningful integration and transformation among models, and is the basis for automation through tools.
- Acceptance and broad adoption of this model-based approach requires industry standards to provide openness to consumers, and foster competition among vendors.

In other words, bringing UML models to the core of building software architectures in a semantically coherent manner is the goal of the MDA initiative. As we will see in later chapters, Executable UML attempts to provide both a well-defined notation as well as a way to organize a system around platform independent models. The notion of Platform Independent Models (PIM) is important because it does not commit to a specific technology. Once the decision is made to commit to a specific technology or platform a PIM would be translated into a Platform Specific Model (PSM). Figure 4.2 is an example of how the OMG envisions a PIM translated to a PSM within the context of the
MDA. Note, that for every platform independent concept within the PIM, there is a technology mapping that is required for the PSM to be realized.

Figure 4.2 - PIM to PSM mappings
Chapter 5 - OMG xUML Specifications

As mentioned in the prior chapter, the MDA initiative aims to use models to drive not only the design of systems, but also the implementation of systems (i.e. – Model-Based Programming). The MDA initiative has spawned the idea of clearly defining semantics for a subset of UML modeling elements that would allow UML models to be executed. So in essence, Executable UML would be the engine behind the conceptual Model Driven Architecture.

Currently the concept of Executable UML is not an OMG standard. However, OMG has put out an RFP (Request for Proposal) titled "Foundational Subset for Executable UML Models". This RFP is basically a call to define the foundations and semantics of Executable UML. Dr. Robert Soley (CIO of the OMG) has graciously provided access to a collaborative response to the RFP [4]. Although still in draft mode, the Executable UML Foundation Specification provides a good idea of how the Executable UML foundation is being defined. The submitters of this draft include IBM and Lockheed Martin.
5.1 Objectives of Specifications

- The objective of the specification is to enable a chain of tools that support the construction, verification, translation, and execution of computationally complete executable models.

- In order to meet the objective, the specification takes a computationally complete subset of the currently defined UML 2.0 artifacts and defines execution semantics for this subset.

- "Computationally complete" means that the subset shall be sufficiently expressive to allow definition of models that can be executed on a computer.

5.2 Executable UML Architecture

Figure 5.1 offers an architectural description of xUML using UML notations.
The following describes key areas of this architecture:

- The Foundational UML Subset (fUML) is an executable subset of standard UML that can be used to define, in an operational style, the dynamic and static semantics of modeling languages such as standard UML or its subsets and extensions. For example, the semantics of UML statecharts can be specified as a program written in fUML.
• The Basic UML Subset (bUML), which is a subset of fUML that (a) has a corresponding mathematically formal semantics definition and (b) is used to specify the semantics of fUML. The intent is to provide a basis for formal treatment of fUML programs.

• A Runtime Model Library which contains definitions of auxiliary classes used by the runtime services.

The various components in the xUML specification point to the clear definition of UML semantics as well as the concept of a model Virtual Machine to execute the model.
Chapter 6 - Balcer and Mellor xUML Design

The concept of xUML can be best described as using UML models for both design and implementation. Currently, a typical UML 2.0 modeling tool will allow an architect to construct a model of a system and use a portion of that model, typically the class diagram, to implement source code in a specific programming language. Code compilation then takes place either within or outside of the UML tool. Additionally, deployment steps may be required in order to get the software into a state where it can be executed. With the xUML concept an architect is able to theoretically execute a UML model without implementing any language or platform specific code.

xUML should not be confused with the current industry practice of creating a UML model and using tools to "manufacture" code from the model. An xUML model would not need to be translated before being executed and verified. In xUML, the translation would only take place as a final step of converting the model into a Platform specific implementation. The translation would be the job of a Model Compiler. The Model Compiler would basically translate action language into a specific language. In other words, the level of abstraction for compilation would be raised to the model level via the model compiler. Mellor and Balcer [5] describe one possible manifestation of xUML modeling. Figure 6.1 describes the process they suggest for building a xUML model.
In this method, Use Case and Domain modeling are used in the analysis stage to separate a system into different domains/aspects. Doing so helps to build reusable models that can be utilized in different projects. The last three steps are described in more detail in the following sections and illustrated in Figure 6.2.
6.1 Class Diagram

Many of the techniques used in creating UML 2.0 class diagrams are the same. The main difference is that relationships, associations, and multiplicities need to be explicitly defined in xUML. Explicitly defining these components of the class diagram is necessary in order to define semantics and constraints that will allow a model to be computationally executed. Without solid semantics, a model would be open to many interpretations and would make it impossible to compile and execute. Note that in contrast to standard UML diagrams, the xUML diagrams do not contain methods inside
the Class element. In other words, the Balcer/Mellor design of xUML does not require the definition of Class methods.

Figure 6.3 shows an example of a Class Diagram sketched in an xUML tool implemented by Kennedy Carter [19]. Note that all relationships are clearly and uniquely labeled so that a Model Compiler can distinguish between relationships.

Figure 6.3 - xUML Class Diagram
6.2 State Modeling

State diagrams are used to define the various states that an instance of a class can be in. Associated with each state are events and actions. Events are what trigger the transitioning of states in a state diagram. The signal that activates an event can come from another state within the state diagram or from an independent state diagram. Actions are associated with individual states and are basically a set of steps that take place as a result of entering a given state. Figure 6.4 is an example of an xUML State Diagram also sketched using the Kennedy Carter tool.

Figure 6.4 - xUML State Machine Diagram
A system at any point in time can be thought of as having various state diagrams running in parallel. In other words, each class instance can be thought of as an autonomous, active object running in parallel with other objects. These active objects and the semantics associated with their lifecycles are defined as state diagrams.

6.3 Defining State Actions

For each state that is defined in a state diagram, there are associated actions that are implemented using an action language. Actions are a set of procedures that carry out a set of instructions. Currently, there are different implementations of action languages [16,17] based on an official Action Semantics [18] defined by the OMG. Actions can be thought of as the lowest common denominator in an xUML model. Actions are responsible for relating objects, sending signals to other objects, and in general defining instructions that allow a model to be simulated/executed.

6.4 Model Execution

In the Kennedy Carter Tool, model execution is based on Test Methods that are defined by the modeler. State Charts are used to keep track of Class Instances along with their current state and attribute values. The test methods are once again written in an action language compliant with OMG’s action semantics. Also, model initialization logic is implemented in the action language in order to create some initial class instances necessary for the model to execute. Figure 6.5 shows a screenshot of the Kennedy Carter tool executing the example Gas Station System that comes with the tool.
Figure 6.5- xUML Model Execution in iUMLLite
Chapter 7 - Balcer and Mellor xUML Design Flaws

Balcer and Mellor have created an innovative approach for creating an xUML model. However, there are several design flaws in their proposed method for creating xUML models. Among them is the reduced role of the Class diagram, centering model execution around the state diagram, the dependence on an action language, and the added complexity of a model compiler. The following sections will expound on each of these design flaws.

7.1 Reduced role of Class Diagram

For good reason the class diagram is considered the focal point of the standard UML diagram when it comes to design and implementation. UML’s inception was largely created to facilitate Object-Oriented Methodology. In particular, the Class diagram was designed to allow a software architect to sketch out classes, attributes, operations, and class relationships. Most UML tools use the Class diagram to help the software developer implement the source code for a software system. Although Balcer and Mellor make progress in more clearly defining semantics in the Class diagram, they remove the need to define operations at the class level. Removing the need to do define operations at the Class Diagram level is in contrast to what is taught in Object-Oriented Methodology.
7.2 Use of State Diagram

The state diagram is most often associated with designing the lifecycle of an object. In the Balcer/Mellor xUML design, the role of the State diagram takes on a larger role in becoming the focal point of adding execution facilities to a UML model. In the prior section it is pointed out that the Class Diagram takes on a reduced role. Methods are traditionally defined in the Class Diagram, however in this xUML design all of the operations are defined in the State Diagram using a procedural action language. In the History chapter we saw that a strong motivator for creating UML was to model for the Object-Oriented programming paradigm. By focusing on defining actions within State nodes of a state diagram, a modeler seems to be encouraged to use techniques consistent with procedural based programming rather than the Object Oriented Programming that UML was originally intended to support.

7.3 Action Language

To understand the flaw in using an action language to executable models, we need to first understand the motivation for its use. As mentioned in the chapter about the MDA initiative, basing software architecture on Platform Independent Models is an important component of the MDA. OMG chose to create and use an action semantics language that would allow models to execute in a platform independent manner.

One of the fundamental problems with using an action language is that a procedural language is being used in a modeling paradigm that was supposed to model an object-
oriented system. The second problem is that there already exists an object-oriented languages such as Java that can be utilized to provide the platform independence that the MDA hopes UML models can achieve.

7.4 PIM to PSM Translation

One of the ideas for xUML is to allow a Model to be translated into a Platform Specific implementation. Having complicated model compilers to produce platform specific implementations is unreasonable. If a model is built in a platform independent manner, then why would a modeler be forced to produce a platform specific implementation.
Chapter 8 - Alternative xUML Engine for Model-Based Programming

This chapter will define an alternative conceptual xUML design by which true Model-Based Programming can be achieved. The proposed design draws on some of the strengths of the Mellor/Balcer design, while disregarding or redefining some of its weaknesses. The design will be described using a top down approach. First, the process that a UML modeler would use to create an xUML model will be described as illustrated in Figure 8.1. Next, the semantics of the relevant UML diagrams and elements necessary for creating xUML models are described. Finally, the architectural details of the xUML Engine design will provide an understanding of how an xUML model would be executed.

8.1 Iterative Process for Building an xUML Model

Figure 8.1 illustrates an iterative approach for creating xUML models. The process utilizes diagrams that are commonly used by UML modelers including the Use Case, Class, Sequence, and Activity Diagrams. As with standard UML modeling, Use Case diagrams would be used to illustrate the high level functionality of the system. The Use Case diagram would then drive the static and dynamic design of the system in the form of Class and Sequence Diagrams. As illustrated in 8.1 the creation of the Sequence and Class diagrams are complementary in that the Sequence diagram requires that Classes be defined. Likewise the design of Sequence Diagrams can contribute to the elaboration of a Class diagram, such as adding methods to the Class Diagram. To define the details of the Class methods, activity diagrams would be used rather than a programming language.
Utilizing Activity Diagrams in such a manner is what would make this design a truly model-based programming paradigm.

The last step of the process will utilize the xUML Engine to compile and execute the xUML model. In order for these diagrams to be utilized for model execution, it is assumed that they are semantically well defined as described later in this chapter. The architectural and design details of the xUML Engine will also be described later in this chapter. The following sections provide more details of each process in Figure 8.1.

![Figure 8.1 - Proposed xUML Modeling Process](image-url)
8.1.1 Use Case Modeling

During the Use Case modeling phase, the functional aspects of the system are modeled in a manner that is consistent with Use Case modeling in standard UML. Figure 8.2 shows an example use case. In this figure, there is an Actor called Customer as well as 4 Use Cases. Each use case is a function that can be performed by the Customer Actor.

![Customer Use Case Diagram](image)

**Figure 8.2 - Banking Use Case Example**

In the proposed xUML design the Use Case diagram provides the functional basis for creating the remainder of the diagrams needed to arrive at an executable diagram. The Use Case drives the design of the Class Diagram and Sequence Diagram. It is
assumed that the Use Case diagram will be tightly linked with the Sequence diagrams in order to keep the xUML diagrams consistent.

8.1.2 Create Sequence Diagram

The creation of the Sequence Diagram elaborates on the functionality defined in the Use Case diagram and relies on the existence of Classes defined in the Class diagram. The sequence diagram can also prompt the creation of methods in the Class diagram. Figure 8.1 shows the interdependent relationship between the Class Diagram and the Sequence Diagram. The relationship is one where the Class and Sequence Diagrams help to define each other. Recall that in the Balcer/Mellor design, the State Machine Diagram is the focus of execution. In the proposed alternative design, the sequence diagram was selected as the focal point of execution because of its attributes that naturally lend to model execution. In particular, the sequence diagram sketches out the series of operation calls involved in a Use Case Diagram. Furthermore, it clearly illustrates the communication and dependencies among the Objects involved in a particular Use Case.
Figure 8.3 shows an example of a Sequence Diagram as taken from the IBM website. Note that the example sequence diagram uses explicit notation. In particular, the return values are always explicitly specified, the guards are explicitly defined, and the parameters are clearly identified. In other words the loose semantics allowed by standard UML would not be compatible with how Sequence Diagrams are created for xUML models. Thus, any tool that is implemented for xUML would not allow Sequence diagrams to be created with loose semantics. More details on the assumed semantics are provided later in this chapter.
8.1.3 Create Class Diagram

In the Balcer/Mellor design, the Class Diagram is semantically sound because it requires a strict definition of class relationships and multiplicities. These design aspects will be reused in the proposed xUML design to establish constraints and boundaries in an xUML model. These relationships and multiplicities must be well defined if the xUML Engine is to properly apply constraints when executing xUML models. In contrast to the Balcer/Mellor design, the concept of defining methods in Classes is reintroduced. Defining operations within Classes is consistent with the original intent of the standard UML Class Diagram. In summary, processes of Creating Class Diagrams in the proposed xUML design is similar to that described in the Balcer/Mellor design with the added step of defining methods inside the Classes.

8.1.4 Define Class Methods Using Activity Diagrams

In standard UML tools the Class Diagrams allow for skeleton code to be created in a platform specific language. For instance, Rational Rose supports implementing Class code in languages such as Java or C#. The proposed xUML still aims to use the Class diagram as the entry point in implementing Class methods. In keeping with the concept of Model-Based programming, the xUML Design will use Activity Diagrams to implement the details of a Class method rather than a platform specific language. Figure 8.4 shows an illustration of an Activity Diagram that calculates the Fibonacci Series as taken from “Teach Yourself UML” [20]. A similar method could be employed to define
the details of UML Class methods. Semantics would have to be clearly defined regarding input parameters, output parameters, return values, and invocation of Class methods whether the methods are inside or outside of the calling object.

Figure 8.4 - Example xUML Activity Diagram

8.1.5 Compile/Execute Sequence Diagram using JVM

As pointed out in the section about the MDA initiative, one of the main goals for the OMG is to facilitate the creation of platform independent models. The xUML engine
therefore must be implemented and run on top of a platform independent environment. Thus, implementing the xUML Engine in Java and executing models on top of the Java Virtual Machine would provide the necessary backbone for platform independent xUML models. More details on the architecture and design details of the xUML Engine will be provided later in this chapter.

The idea behind the process of executing an xUML model is to use the Sequence Diagram as the entry point for execution. This is in contrast to the Balcer/Mellor design, where the State Diagram serves as the focal point of execution. It is intuitive to use the Sequence Diagram because it is an elaboration of the Use Case diagram and it lends to a graphical manner for debugging code. This would allow for the functionality represented by each Use Case to be tested independently.
8.2 Assumed xUML Diagram Semantics

Section 8.1 identifies the participating UML diagrams used in the proposed xUML design. In order for the process to work, semantics have to be defined for all UML diagrams involved. This section describes the key semantics that are required for each relevant UML diagram.

8.2.1 Sequence Diagram Semantics

In order for the xUML Engine to execute an xUML model, the sequence diagrams must be well defined. In particular, the sequence diagrams must address all semantic problems pointed out in Chapter 3. The following is a table of semantics that need to be defined for a sequence diagram to be executed.
<table>
<thead>
<tr>
<th>Defined Semantics</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit return values</td>
<td>Explicit return values are needed by the invoking Class method in order to complete execution.</td>
<td>Figure 8.5 shows the explicit return value pastDueBalance</td>
</tr>
<tr>
<td>Explicit parameter values</td>
<td>Specifying parameters along with their types makes it feasible for a xUML Engine to process sequence diagram logic.</td>
<td>Figure 8.5 shows the explicit parameter studentID passed to addStudent</td>
</tr>
<tr>
<td>Guards are needed for decision points and/or loop control</td>
<td>In order for decision points to be executed by the xUML Engine, the sequence diagrams would have to contain well-defined guards</td>
<td>Figure 8.6 shows the use of guards.</td>
</tr>
<tr>
<td>Class definitions must exist in the called Class.</td>
<td>Class definitions must exist in the called Class for the xUML Engine to fairly execute an xUML model</td>
<td>Figures 8.5 and 8.6 shows examples of Class method Calls.</td>
</tr>
</tbody>
</table>

Table 8.1 - Sequence Diagram Semantics
8.2.2 Class Diagram Semantics

Well-defined semantics on Class diagrams are necessary for applying the appropriate constraints to data structures when executing an xUML model. The following table outlines the key semantics:
<table>
<thead>
<tr>
<th>Defined Semantics</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Properly labeled relationships</td>
<td>Uniquely labeled relationships makes it easy for the xUML engine to identify relationships</td>
<td>Figure 6.3 shows a class diagram in the iUMLite tool with unique relationship labels.</td>
</tr>
<tr>
<td>Explicit Multiplicities</td>
<td>The xUML Engine will rely on multiplicities to make sure that the object population is controlled.</td>
<td>Figure 8.7 shows an example of a well-defined multiplicity between the Item and OrderDetail Classes</td>
</tr>
<tr>
<td>Strict naming of Classes, Attributes, and Methods.</td>
<td>The xUML Engine requires strict naming for classes/attributes/methods in order to correctly convert them into Java Source code.</td>
<td>Figure 8.8 shows an example of a Class definition with strict naming.</td>
</tr>
<tr>
<td>Explicit Parameters/Return Values</td>
<td>Explicit Parameters along with their types are required for constructing Java classes.</td>
<td>Figure 8.8 shows examples of well-defined parameters and return types.</td>
</tr>
</tbody>
</table>

Table 8.2 - Class Diagram Semantics
Figure 8.7 - Semantically Correct Class Diagram

Figure 8.8 - Semantically Correct Class Definition
Activity Diagrams will be used to implement the “low level” programming within the Class methods. The following table defines some of the key semantics required:

<table>
<thead>
<tr>
<th>Defined Semantics</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Parameters</td>
<td>Capability to receive Input parameters.</td>
<td>See boxed labeled A and B in Figure 8.9</td>
</tr>
<tr>
<td>Output Parameters</td>
<td>Able to return values.</td>
<td>See box labeled Answer in Figure 8.9</td>
</tr>
<tr>
<td>Method Calls from Objects</td>
<td>Able to call methods from other Objects.</td>
<td>Call to square method from Math Class in Figure 8.9</td>
</tr>
</tbody>
</table>

Table 8.3 - Activity Diagram Semantics

Figure 8.9 - Semantically Correct Activity Diagram
8.3 xUML Engine Architecture

The xUML Engine is responsible for compiling and executing an xUML model comprised of a Use Case Diagram, Sequence Diagrams, a Class Diagram and each Activity Diagram associated with Class methods. Assuming that the XMI representation for the UML diagrams is provided to the xUML engine, the engine would take the collection of diagrams and translate them into Java code. The xUML Engine would then use the Java Virtual Machine (JVM) to execute the java code necessary to execute an xUML model. Figure 8.10 summarizes the xUML architecture, which is described in subsequent sections.

Figure 8.10 - xUML Architecture
8.3.1 XMI Format

OMG has established a standard XML representation for UML diagrams known as XML Metadata Interchange (XMI). Figure 8.11 illustrates a simple class diagram along with its XMI representation. All diagrams within a xUML model would be included in the XMI file. The xUML Model translator as illustrated in 8.10 would then parse the XMI file in order to execute an xUML model.

Figure 8.11 - IBM site XMI Example

```xml
<?xml version="1.0"?>
<OML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:uml="http://www.omg.org/UML/1.4"
     xsi:schemaLocation="http://www.omg.org/UML/1.4 http://www.omg.org/spec/UML/1.4.xsd"
     xmlns="http://www.omg.org/T2/1.4"
     xmlns:meta="http://www.omg.org/uml"
     xmlns:xmi="http://www.omg.org/xmi"
     xsi:noNamespaceSchemaLocation="http://www.omg.org/xmi/1.4.xsd">
  <OML:package>
    <OML:createClass name="Address" visibility="public"
                     isSpecification="false" isAbstract="false">
      <OML:attributes name="street" visibility="private"
                      isSpecification="false" isAbstract="false"
                      isActive="false"/>
      <OML:attributes name="zip" visibility="private"
                      isSpecification="false" isAbstract="false"
                      isActive="false"/>
      <OML:attributes name="region" visibility="private"
                      isSpecification="false" isAbstract="false"
                      isActive="false"/>
      <OML:attributes name="country" visibility="private"
                      isSpecification="false" isAbstract="false"
                      isActive="false"/>
    </OML:createClass>
  </OML:package>
</OML>
```
In defining the semantics for the xUML Engine in Section 8.2, variations of standard UML diagrams were defined as xUML diagrams. It is assumed that these xUML diagrams have the appropriate associated XMI representations defined.

8.3.2 xUML Model Translator

The xUML Model Translator would basically be a Java program that does the following:

- Parse the XMI representation of the xUML Diagrams using some type of XMI Parser such as the Java-based Java Metadata Interface (JMI)
- Receive initial input values needed to begin executing an xUML, model including the Sequence Diagram to execute and the initial input values.
- Learns the order of operations in the sequence diagram and generates the appropriate Java code.
- Learns the lowest level of instructions from the Activity Diagram and generates the appropriate Java code.
- Examines the Class Diagrams to ensure that valid methods are being called and that multiplicity and relationship constraints are properly enforced.
- Executes the models and renders the execution in a hypothetical xUML modeling tool.
Chapter 9 - Future Work

Although the proposed xUML Engine design is based on existing proven technology, implementing and testing the design along with the tools necessary to execute xUML models would provide a stronger argument for model-based programming. In particular, implementing the xUML Engine with XMI and Java would provide the platform independence sought by the MDA initiative. In addition, providing a concrete and practical Case Study of a Software system built on the xUML Engine would be effective in illustrating the features of the proposed xUML design.
Chapter 10 - Conclusion

xUML seems to be an ideal candidate for true model-based programming. However, work remains to be done in solidifying semantics for UML model elements so that truly executable models can be implemented. Although the Mellor/Balcer xUML design provides some benefits such as tighter semantics, it fails to qualify as a true model-based programming paradigm. The proposed architecture of the xUML Engine in this paper provides a means of model-based programming that will truly not require a platform specific implementation.

However, whether model-based or text-based, programming at some level of abstraction will always be required. In the proposed xUML design, the lowest level of programming detail happened to be an Activity Diagram, but nonetheless it is still a type of programming. So as far as reaching the panacea of programming, which would be programming-free software implementation, it appears that for the foreseeable some level of programming will be required whether model-based or text-based. As for now raising the level of abstraction to model-based programming via a reasonable xUML modeling design seems attainable.
References


    Architecture, Addison Wesley, Boston, 2002


    Lawrence, KS, USA.

    NY, USA.

    object-oriented architecture. Proceedings of the Conference on Object-Oriented
    Programming Systems, Languages, and Applications - OOPSLA '91, Oct 1991,


http://www.cs.colostate.edu/~trungdt/uml_testing/tool/eptud_download/JAL.doc
(accessed 4/1/2007)

3/15/2007)

3/20/2007)

2004.