Formalizing UML semantics and notations for Real-Time and Reactive systems

Gasso Wilson Mwaluseke
Professor Jonathan P. Bowen
London South Bank University

August 31, 2004

Key Words: Object-oriented, Formalism, Concurrency, Inheritance, Semantics, Verification, UML and Notations, Real-Time, Reactive systems, embedded systems.

Abstract

The Unified Modeling Language (UML) is a visual language that provides a way for people who analyze and design object-oriented systems to visualize, construct and document the artifacts of software systems and to model the business organization that use such systems. However, the syntax and semantics of a number of UML constructs are not formal which makes it hard to understand. The use of mathematical techniques to model mathematical entities by building mathematically rigorous models of complex systems are considered to be of utmost importance because it is possible to verify the systems’ properties in a more thorough fashion than empirical testing. When correctly and appropriately applied, these techniques have proven themselves to result in systems of highest quality, improved system reliability, improved design time and comprehensibility and provability. In this paper, we evaluate some of the formal methods on the basis of concurrency, inheritance support, semantics and the mapping process, traces, representation of abstract syntax components, notation, Real-Time features support, tool support and automatic verification of behaviour as applied to Dynamic models of UML semantics. This work leads to the establishment of the degree of formalism of a formal method that will be deemed generally acceptable for formalizing dynamic models of UML semantics and notations and the extent of UML support for real-time systems.

1 Introduction

UML is a graphical notation whose model (specification of a system) is composed of different kinds of diagrams each representing a different view or part of the system [8]. These diagrams can be classified into two categories namely
static and dynamic groups. UML is said to be a semi-formal language because its syntax and static semantics (the model elements, interconnections and well-formedness) are defined precisely, but its dynamic semantics are specified neither formally nor algorithmically [2]. UML suffers from a lack of formal semantics which is a common trend for many Object Oriented (OO) methods from which UML is derived. This does not only lead to confusions and different interpretations when analyzing a model but also decreases the ability to develop tools and guidelines to help the specification and design. To be able to check, validate, verify and refine software, designers require a more formal semantics of UML. Real-Time embedded and Reactive systems require more care and understanding in their design stages than other systems because of their complexity and also time handling. UML works hand in hand with Object Constraint Language (OCL). However, OCL and UML are both semi-formal notations. This means that the two are prone to misinterpretations and thus not very good modeling languages as far as Real-Time, embedded and Reactive systems which demand high level of correctness, concurrency, timeliness, responsiveness and fault tolerance. In order to define a formal semantics for UML, we list some features common to UML and indeed to object-oriented systems and use these features to select the suitable formal method(s) available in the market that best fits into the work of formalizing UML. In this paper, the matching of features of UML with the suitable formal method(s) is done in two stages namely listing down some features and then aligning these features with the features of some formal methods to see if such formal methods are capable of formalizing them for we strongly believe that mapping between UML and formal methods can only be achieved if their features match. The rest of the paper is arranged as follows: Modeling Real-Time embedded systems, Real-Time systems and Reactive systems using UML, Capturing and formalizing the dynamic semantics of the Real-time systems, related work and conclusion.

2 Some dynamic features of UML

UML has many features which are quite common for object-oriented systems. However, in this paper we have just picked a few of them which we believe are very common in complex systems. These features are explained below:

2.1 Concurrency and real-time

There are two reasons for constructing systems which show concurrent behavior:

1. Concurrency can be an important aspect of the world being modelled if the problem domain consists of co-existing elements that the model of the domain recognizes concurrently as active components.

2. Speeding up large computation may involve subdividing the computation into parts which can be computed concurrently thereby gaining time on
a multi-processor machine. This can lead to highly parallel systems, with hundreds, or even thousands of executions of some basic algorithm executing simultaneously.

In the real world concurrency is the norm. The world consists of collections of entities, people, machines, etc. which co-exist each of which may be pursuing its life and objectives, modifying its own state and the state of other objects in its environment, concurrently with others. The state of a world entity is represented in the model by its data variables; its laws of behavior are captured in the algorithms of the code. If the objects are active then they must be concurrently active. If an object in the real world changes its state spontaneously, by its own actions, then in the computer model its algorithms should execute concurrently with those of other world objects, they will be concurrent processes. The Unified Modeling Language (UML) is capable of modeling a system that exhibits concurrency and with some extensions, it is also capable of modeling real-time systems. UML notations fall under two categories namely those dealing with static features and those dealing with dynamic features with the extension to deal with real-time features.

2.2 Inheritance support

This is one of the key features of any object-oriented methodology or language. It is divided into two main categories namely:

1. Representational inheritance.
   The composed structure, built from the enumeration of the instance variables of the object, forms a representation or state of the object. This representation is completely determined by the class definition in which case a type can be associated to each class definition representing the state. The definition of a subclass of an existing class introduces an extended type for the subclass. The type definitions may form a long chain of types, which are 'backwards' assignment compatible in the direction of the root type. However, it is important to note that the extended type is not a subtype because it contains elements of different types than those of the basic type. In general, this representation inheritance scheme for a subclass implies inheritance of all invariant expressions from superclasses. Representational inheritance is restricted to strictly single inheritance.

2. Functional inheritance.

Inheritance of methods in a strongly typed environment might lead to conflicts with the notion of typing. A good example of such a conflict is trying to add a vertex to a square without deforming the square. The addition of a vertex to a square does result in a polygon but different from the square. This can be avoided by introducing the notion of controlled inheritance which lets the user to specify the behavior without automatic coupling to the representational inheritance. This leads to multiple inheritance.
UML is capable of capturing and presenting both single and multiple inheritance at design level.

2.3 Consistency checking
In UML, consistency checking is very important. For instance interaction diagrams and class diagrams should be mutually consistent. Likewise statecharts diagrams must be consistent with other models. Tools developed should be able to trace and verify the consistence between models.

2.4 Possibility to animate a formal specification
This is very important in order to provide a prototype of the model under consideration. The syntactical notations of a specification language should be flexible enough to allow the possibility of animation.

2.5 Timing
UML allows times and timing constraints to be represented on sequence diagrams as follows:

1. Distances between two messages
   Distances on the lifelines represent intervals of real time either exactly or loosely.

2. Time for messages passing between Objects
   This involves drawing arrows to indicates messages sent and received at specified time intervals.

3 A cross section of formal methods
With increasing complexity of software, and a greater requirement for software reliability, it has become clear that the old ad hoc informal methods of programming are no longer stringent enough [7]. The following is a list of some formal methods that have been picked up for matching with the above UML features:

3.1 The Z specification language
Z is a state-based specification language that has not been used for the specification of concurrent systems partly because it lacks a means of conveniently specifying systems in terms of components (which is an implementation issue) [13]. The Unified Modeling Language (UML) is a visual language that provides a way for people who analyze and design object-oriented systems to visualize, construct and document the artifacts of software systems and to model the business organizations that use such systems. Using Z specification language to
formalize some of the UML constructs will prove to be insufficient. However, Z specification language is ideal for specifying top-level functionality of a system. For example, a simple buffering system that can store up to ten items can be modeled in Z as follows:

$$\text{System}[T]$$

$$s : \text{seq}T$$

$$\#s \leq 10$$

$$\text{Init}[T]$$

$$\text{System}[T]$$

$$s = \langle \rangle$$

$$\Delta \text{System}[T]$$

$$\text{in?} : T$$

$$\#s < 10$$

$$s' = s \smallsetminus \langle \text{in?} \rangle$$

$$\text{Out}[T]$$

$$\Delta \text{System}[T]$$

$$\text{out!} : T$$

$$s \leftrightarrow \langle \rangle$$

$$s = \langle \text{out!} \rangle \smallsetminus s'$$

### 3.2 Object-Z language

Object-Z is an extension of Z which includes a special class construct to encapsulate a state schema with the operations which may effect that state and is designed to support an object-oriented specification style. In Object-Z, a system is thought of as a collection of distinct, interacting components called objects. The above example could be modeled in Object-Z using one class inside of which there are one nameless state and three operations namely initializing operation, in and out operations.
In Object-Z, an operation can refer only to the state of the object to which it belongs whereas a Z specification typically defines a number of schemas defining state and operations.

3.3 Vienna Development Method (VDM)

Like Z, Vienna Development Method is a state-based specification language that allows not only specification, but also the ability to move from the high-level abstract data types of the original specification to the data types of the target programming language. It also embodies a principle known as operational decomposition which enables decomposition of specified functions and operations into more implementable versions of the target language. However, VDM does not include the ability to specify concurrent processes [7] and this is a disadvantage in the specification of communications protocol and other areas such as UML statecharts and sequence diagrams. A lot of work is done to make VDM have the ability to handle concurrency but none has been documented and approved by ISO.

3.4 VDM++

VDM++ is an object-oriented specification based on VDM-SL. It consists of a set of class definitions which provide templates for objects that possess attributes and operations (methods) whose types and properties are defined in the classes. There are also features provided that are specifically related to concurrency.
and real-time. Concurrent systems are often confused with real-time systems because real-time systems are often concurrent and vice versa [18].

3.5 Language Of Temporal Ordering Specification (LOTOS)

LOTOS is a formal description technique designed to describe distributed concurrent information processing systems, in particular for service definition and protocol specification within the OSI (Open Systems) architecture and related standards [7]. It is a well-defined mathematical notation providing a good basis for analysis and the development of support tools, including simulators, compilers and test sequence generators. The basic constructs of LOTOS allow modelling of sequencing, choice, concurrency and non-determinism in an entirely unambiguous way. It also permits modelling of synchronous and asynchronous communication.

3.6 Extended State Transition Language (Estelle)

Estelle is a modification and extension to Pascal where program-level constructs have been replaced by constructs to define finite-state modules exchanging queued messages. Estelle specification defines a system of hierarchically structured state machines. A distributed system specified in Estelle is viewed as a collection of communicating components called module instances. Modules are declared either as processes or activities which can be used to represent loose parallelism, where concurrency is expressed by interleaving of parallel actions.

3.7 Larch

Larch is a property-oriented specification language combining both axiomatic and algebraic specifications in two-tier specification namely the auxiliary specification and the interface specification. Larch is particularly good when verifying an implementation.

3.8 The B Specification Language

4 Comparison table

It is apparent from the above explanations that most of the object-oriented formal methods can be considered to be suitable for formalization of UML semantics because they combine both the static and dynamic aspects which are the properties of UML. The following table shows a summary of the formal methods and their qualities.
From the above table Object-Z emerges favourable to formalizing UML semantics and notations. The work of formalization of UML semantics and notations is very important for model validation and verification to be possible. The Model Driven Architecture, for instance, will be incomplete if UML will be in its current state. However not all formal methods are capable of formalizing all the semantics and notations that UML can exhibit as depicted in different domains of systems. In all fairness, almost all of the above formal methods can formalize the UML semantics to some extent. However, some can do so the hard way whereas others one need not work very hard to accomplish the goals.

5 Modeling Real-Time, Reactive and Embedded Systems

A Real-Time system is a system (device) that has performance deadlines on its computations and actions. In other words, a Real-time system is a system in which actual times are important [17]. Real-Time systems are often embedded in the sense that the computational system exists inside a larger system with the purpose of helping that system to achieve its overall responsibilities. Real-time systems are broadly characterized by the need to respond to external events within tight time constraints and thus they frequently exhibit concurrent behaviour in the form of simultaneous execution pathways or threads of control. As a result, an application of that nature will include objects that coordinate and initiate threads of control (Active objects) and instances of active classes. Real-time application also may include objects that work only within a thread of control (passive objects) and they belong to passive classes. UML is capable of capturing all the above features using Sequence diagrams, collaboration diagrams and other notations. A sequence diagram is an interaction diagram that captures the behaviour of one scenario. A sequence diagram consists of objects represented in a rectangle with the name underlined, messages represented as a solid-line arrow, and time represented as a vertical progression [10]. The message can be simple involving transfer of control from one object to another), synchronous (indicating that the object waits for an answer before it proceeds with its business), or asynchronous (indicating that the object sends messages without waiting for an answer before proceeding with other business). Time
taken for messages to pass between objects is also considered and is represented by vertical distance between two messages. Let us consider embedded system where a thread or a task is a program which performs some meaningful job within an application. We consider the CPU (Central Processing Unit) in particular with multitasking. Each thread has a number that denotes its priority within the application program and is usually in one of the six states namely:

- Dormant - in memory not made available to operating system
- Read - it can run, but the thread that’s running has a higher priority
- Delayed - suspended itself for a specified amount of time
- waiting for an event - some event has to happen for it to run
- Running - it has the attention of the CPU
- Interrupted - the CPU is taking care of an interrupt

In Real-time Operating system, a kernel, manages the time the CPU spends on individual threads. The kernel has a scheduler that determines which thread will execute next. Kernels, depending on how they deal with interrupts, can be pre-emptive or non-pre-emptive. The following is a model of a non-pre-emptive kernel as a UML sequence diagram as indicated in Figure 1.

6 Formalizing the semantics of Real-time systems

UML can capture the semantics of Real-time systems and indeed all systems but because it does so in English or natural language, it has no means of validating or verifying the same. Formal methods are therefore are placed strategically to help UML semantics to be provable. However, the formalism of different formal methods differ with different systems. Object-Z and CSP seem to have a stable progressive semantics that can accommodate many complexities of systems. Of these two Object-Z fits in quite well with UML semantics. Semantically, a message is a communication between two objects that convey information with the expectation that an action will be performed. In the above example, there are three objects namely the Interrupt, the CPU and the Thread. The interrupts sends a request to the CPU. When the CPU recognizes an interrupt (which is an asynchronous event), it saves what it was doing and invokes an Interrupt Service Routine (ISR) that processes the event. Using Object-Z, we can name the first message as IntMessage and the sender’s name as Interrupt and the receiver is the CPU. This alone can formalized as follows in an Object-Z schema:

\[
\text{TypeOfMessage} ::= \text{Simple} \quad \text{– simple message} \\
| \quad \text{Asynchronous} \quad \text{– Asynchronous message} \\
| \quad \text{Synchronous} \quad \text{– Synchronous message}
\]
Figure 1: Interrupt Sequence Diagram.
The IntMessage class or schema will be displayed as follows:

```
IntMessage
  sender : Interrupt
  receiver : CPU
  ready? : B
  sequence : N
  MType : TypeOfMessage

  MType = Asynchronous
  ready ⇒ receiver.send()
  sequence = 1
```

According to the above explanation, the CPU at this point suspends whatever it was doing by invoking the ISR. We can call this process as CPUInvokes and proceed as follows with Object-Z operation:

```
CPUInvokes
  sender : CPU
  receiver : ISR
  ready? : B
  sequence : N
  MType : TypeOfMessage

  MType = Synchronous
  ready ⇒ receiver.save()
  sequence = 2
```

The above examples do not include time but in reality time is necessary and it is possible to represent in Object-Z schemas.

7 Related work and conclusion

We are currently working on formalization of UML semantics using Object-Z. We are also developing a formal tool based on ArgoUML using Object-Z for the same purpose of proving the Hypotheses presented on the reasons for formalizing UML semantics. We believe that our tool will be able to meet the requirements of real-time systems as well as the ever changing versions of UML semantics.

References


