Formal methods for object and component-based systems (Embedded systems)

Gasso Wilson Mwaluseke
Institute for Computing Research
London South Bank University
August 7, 2006

Key Words: Object-oriented, Formalism, Component-Based Software Development (CBSD), MDA, Semantics, Formal Verification, UML and Notations, Real-Time, Reactive systems, embedded systems.

Abstract
Object-oriented style of programming has become the prime technology of software development in the past few decades. The main reasons for the overwhelming popularity of this programming paradigm are better structuring and modularization as well as a high degree of polymorphic reuse that it supports. Component-based programming takes the benefits of object-orientation to an even higher level by allowing development of light-weight, highly-customized, independently extensible applications. Component-based systems consisting of loosely-bound highly configurable components and capable of accommodating continuous evolution can provide software developers and users with the same levels of plug-and-play interoperability that are available to manufacturers and consumers of electronic parts or custom integrated circuits. Establishing a software component market similar to the market of hardware components is the ultimate goal of the software industry.

The issue of correctness of object-oriented and component-based systems deserves close consideration in view of the present and ever-growing popularity of the corresponding programming styles and the necessity to enhance reliability of programs. Traditionally, correctness has been considered as a crucial requirement mainly for safety-critical systems, but nowadays, the need for ensuring correctness of object-oriented and component-based systems is becoming more widely recognized. In open systems, which are composed and extended by end users and characterized by a late integration phase, it is impossible to conduct a global integrity check. Therefore, it should be possible to guarantee error-free operation of resulting applications by establishing correctness of constituting components. In order to design and develop such systems, a formal modelling language will be required. The Unified Modelling 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 look at the possibility and suitability of using UML for Component-Based software developments (CBSD) and also evaluate some of the formal methods for their suitability to formalize UML semantics and notations. This work leads to the establishment of the degree of usage of UML for Component-Based Software Development or indeed for any software development.

1 Introduction

Component-Based Software Development is a software development approach in which all aspects and phases of development lifecycle, including requirements analysis, design, construction, testing, deployment and project management are based on components [8]. 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 [7]. 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 analysing 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 modelling 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: Some Features of Component-Based Software Development, How UML facilitates Component-Based Software development, 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 Modelling Language (UML) is capable of modelling a system that exhibits concurrency and with some extensions, it is also capable of modelling 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 and simulation. For the UML to be effective in terms of modelling systems, it must have its semantics formalized. The current formal specification languages have different formalisms differing in
their capabilities of formalisation. However, we believe that a suitable formal method is one that maps most of the features of object-oriented modelling languages so that very little is lost in the translation between the specification and the model and formalising agent.

3 A cross section description 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 [6]. 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) [11]. 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:

\[
\begin{align*}
\text{System}[T] & \\
& s : \text{seq}T \\
& \#s \leq 10
\end{align*}
\]

\[
\begin{align*}
\text{Init}[T] & \\
& \text{System}[T] \\
& s = ()
\end{align*}
\]

\[
\begin{align*}
\text{In}[T] & \\
& \Delta \text{System}[T] \\
& \text{in}? : T \\
& \#s < 10 \\
& s' = s \smallfrown (\text{in}?)
\end{align*}
\]
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 as indicated below:

\[
\begin{align*}
\text{Buffer}[T] & \\
\Delta \text{System}[T] & \\
\text{out! : T} & \\
\text{s <> \{\}} & \\
\text{s = (out!) ^ s'} & \\
\end{align*}
\]

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 [6] 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 [16]. There is still ongoing work on making VDM++ capable of performing the object-oriented formalising duties.

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 [6]. 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 modeling of sequencing, choice, concurrency and non-determinism in an entirely unambiguous way. It also permits modeling 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 speci-
fication and the interface specification. Larch is particularly good when verifying an implementation.

### 3.8 The B-Method Specification Language

The B-Method is not intended for creating software in the application layer, since applications usually have informal requirements, but it is intended for producing software in the API layer and the component layer because in these layers, the requirement is often formally expressed [17]. B-Method is an implementation formal specification language. The B-Method is a collection of mathematically based techniques for the specification, design and implementation of software components. Systems are modeled as a collection of interdependent Abstract Machines, for which an object-based approach is employed at all stages of development. Abstract Machine constructs are described by the following BNF-like syntax; the order in which optional clauses appear is not significant.

Here, the syntax `exp1 — exp2` indicates `exp1` or `exp2` (choice), `¡ exp ¿` indicates zero or one occurrence of `exp` (optionality) and `¡¡ exp ¿¿` indicates zero or more occurrences of `exp` (repetition).

Identifier, UpperCaseIdentifier, Rule, Bnumber, Formula and ProgramLike-Formula are as defined in the section on the B-Platform and in the B-Tool Reference Manual.

An Abstract Machine is described using the Abstract Machine Notation (AMN). A uniform notation is used at all levels of description, from specification, through design, to implementation.

AMN is a state-based formal specification language in the same school as VDM and Z. An Abstract Machine comprises a state together with operations on that state. In a specification and a design of an Abstract Machine the state is modeled using notions like sets, relations, functions, sequences etc.. The operations are modeled using Pre- and Post-conditions using AMN.

In an implementation of an abstract machine the state is again modeled using a set-theoretical model, but this time we already have an implementation for the model. The operations are described using a pseudo-programming notation that is a subset of AMN.

The B-Method prescribes how to check the specification for consistency (preservation of invariant) and how to check designs and implementations for correctness (correctness of data refinement and correctness of algorithmic refinement).

The B-Method further prescribes how to structure large designs and large developments, and promotes the re-use of specification models and software modules, with object orientation central to specification construction and implementation design.
3.9 Communicating Sequential Processes (CSP)

Communicating Sequential Processes, or CSP, is a language for describing patterns of interaction. It is supported by an elegant, mathematical theory, a set of proof tools, and an extensive literature. CSP is a process-based specification language that models concurrent systems as a collection of processes which interact via a 'handshake' mechanism. However CSP is not ideally suited to the specification of higher-level functionality [12]. It supports concurrency but not very keen with inheritance and classes.

3.9.1 TROLL

TROLL Stands for Textual Representation of Object Logic Language. The TROLL approach supports the declarative specification of conceptual models. It integrates concepts for modelling of dynamic, structural and process aspects. It combines the intuitive diagrammatic notation OMTROLL with a textual one. TROLL allows the modeling of concurrent object system.

4 Summary of Formal Methods properties 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 properties.

<table>
<thead>
<tr>
<th>Language</th>
<th>Concurrency</th>
<th>Inheritance</th>
<th>Animation</th>
<th>Tool</th>
<th>Model-based</th>
<th>Refinement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>NO</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Object-Z</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>VDM</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>VDM++</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Estelle</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Larch</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>LOTOS</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>B-Method</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>CSP</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>TROLL</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

In this table, concurrency and simulation or refinement have been used as main determinants to evaluate formal methods suitable for formalising UML semantics and other constraints. However CSP and Object-Z have emerged to be the greatest contenders. But of the two, Object-Z has emerged the most favourite in that some aspects found in Object-Z are not found in CSP whereas many aspects found in CSP are also found in Object-Z.
5 Features of Component-Based Software Development

A Component is a software object, meant to interact with other components encapsulating certain functionality or a set of functionalities [5]. It is an abstract unit of software instructions and internal state that provides a transformation of data via its interface. On the other hand, a software component is a unit of composition, with contractually specified interface and explicit context dependencies only and can be deployed independently and is subject to composition by third part. The following three features are very important in Component-Based Software Development:

1. Reusability
   A software component can be reused in many different applications or in building other large components and allow cross language reuse.

2. Replacement
   A software component can be interchanged or replaced with any other component that provides the same interfaces without breaking the system.

3. Separation of interface from implementation This is different from those found in many programming languages.

6 Definitions of Real-Time, Reactive and Embedded Systems

A Real-Time system is a system (device) that has performance deadlines on its computations and actions. 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.

7 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 [15]. 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. 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.

8 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:

\[
\text{MessageType} ::=
\begin{align*}
\text{Simple} & \quad \text{– simple message} \\
\text{Asynchronous} & \quad \text{– Asynchronous message} \\
\text{Synchronous} & \quad \text{– Synchronous message}
\end{align*}
\]

The IntMessage class or schema will be displayed as follows:

\[
\begin{array}{l}
\text{IntMessage} \\
\text{sender} : \text{Interrupt} \\
\text{receiver} : \text{CPU} \\
\text{ready?} : \mathbb{B} \\
\text{sequence} : \mathbb{N} \\
\text{MType} : \text{MessageType} \\
\text{MType} = \text{Asynchronous} \\
\text{ready} \Rightarrow \text{receiver}.\text{send}() \\
\text{sequence} = 1
\end{array}
\]

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:

\[
\begin{array}{l}
\text{CPUInvokes} \\
\text{sender} : \text{CPU} \\
\text{receiver} : \text{ISR} \\
\text{ready?} : \mathbb{B} \\
\text{sequence} : \mathbb{N} \\
\text{MType} : \text{MessageType} \\
\text{MType} = \text{Synchronous} \\
\text{ready} \Rightarrow \text{receiver}.\text{save}()\text{sequence} = 2
\end{array}
\]

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

### 8.1 Conclusion

We believe that in order to match one language to another, the two languages must be able to map semantically. This is evident with Object-Z. Since Object-Z is an object-oriented formal specification and UML a semi-formal object-oriented modeling language, most of their common syntactic and well-formedness
rules as well as dynamic semantics can be mapped with ease. Time and space would not permit us to demonstrate this fact, but it is possible to do so.

9 How UML facilitates Component-Based Software development

There are two kinds of diagrams in UML that are used to model aspects of the implementation of computer system namely:

1. Component diagrams

- Component diagrams are used to model the physical software components and the relationship between them.
- They are used to model source code and the relationships between them.
- They are used to model the structure of releases of software.
- They are used to specify the files that are compiled into an executable [2].

They can also be used to model many other software objects that make up a computerized system, for example the relationship between help files and the program files that invoke them or the structure of tables in a database.

2. Deployment Diagrams

- These are used to model the physical hardware elements and the communication paths between them.
- They are used to plan the architecture of the system.
- They are used to document the deployment of software component on hardware nodes.

Besides the features of CBSD mentioned above, CBSD calls special concepts or needs on Interfaces. With interfaces, components represent the physical packaging of logical components. Another important aspect of CBD is the component framework. At present CORBA (Common Object Request Broker Architecture) and DCOM (Distributed Component Object Model) are among popular frameworks.

The Unified Modeling Language (UML) provides component diagrams and interface modeling mechanism for the basic CBD requirements. UML also offers extension mechanisms namely: stereotypes, tagged values, and constraints, which provide the flexibility in extending UML semantics and tailoring it for use in framework-specific component-based system.
10 Related work and conclusion

The fact that UML’s semantics, notations and constraints are semi-formal, the models created become very difficult to understand. The models cannot be verified and consistency checking is almost impossible. We are currently working on the formalization of UML semantics and other constructs (both static and dynamic) using Object-Z and thereby building up a formal tool using the existing open source tool called ArgoUML as well as SAL (Symbolic Analysis Laboratory) for model checking and theorem proving. We believe that since UML is a defacto Object-Oriented modelling language, formalizing its semantics and notations using a suitable formal method such Object-Z, will not only be a positive step towards software component provability but also making UML what is supposed to be as a formal modelling language.

References


