
Formal Description of the UML Architecture and Extendibility

Yingxu Wang

*Dept. of Electrical and Computer Engineering
University of Calgary
2500 University Drive, Calgary, Alberta, Canada T2N 1N4*

Yingxu.Wang@acm.org

ABSTRACT: *Since its emergence in 1995, the Unified Modeling Language (UML) has become part of the mainstream of object-oriented software development in a wide range of applications. This paper presents a formal description of UML technologies for visualized specification and modeling of software systems, and analyzes the usability of UML views and diagrams. Requirements and extension of UML capability to real-time software system specification and description are explored, and a real time process-algebra-based approach for extending UML's descriptivity for real-time system modeling is provided. In addition, findings and improvement approaches on UML applications in real-world projects are discussed.*

RÉSUMÉ: *Depuis son émergence en 1995, UML est devenu prépondérant dans les développements de logiciels orienté-objets et pour une grande variété d'applications. Ce papier présente une approche formelle des techniques UML pour la présentation visuelle des spécifications et la modélisation de systèmes logiciels. Il analyse également l'utilisation des vues et des diagrammes UML. Les besoins et les extensions d'UML vers les systèmes temps-réel sont explorés et une algèbre de processus temps-réel est proposée. Finalement des références et des améliorations d'UML pour des applications ou des projets réels sont discutées.*

KEY WORDS: *Software engineering, specification, description, modeling, UML, formal description, extension of UML, real-time system, formal methods, CSP*

MOTS-CLÉS: *Génie Logiciel, spécification, description et description formelle, modélisation, UML, système temps-réel, CSP, méthodes formelles*

1. Introduction

Object-oriented (OO) programming emerged in the 1980s is one of the significant achievements of software engineering after structured programming. Object technologies have absorbed merits of structured programming and abstract data types (ADTs), and combined them with several new properties such as encapsulation, inheritance, reusability and polymorphism. The Unified Modeling Language (UML) [UML 99, RUM 98] is one of the widely accepted methodologies for object-oriented software specification and description using visual notations.

Object is an abstract model of a real-world entity and/or a conceptual artifact that is packaged by an integrated structure of interface and implementation, and is described by methods for its functions and by data structures for its attributes. For a well packaged object, the only access means to it is via the interface of the object. Its implementation of methods and related data structures are hidden inside the object, which enables the implementation to be changed independently without affecting the interface of the object.

Object-orientation is a type of system design, analysis and implementation methodology which supports integrated functional-and-data-oriented programming and system development. Object-oriented technologies were originally designed for programming. However, as OO programming became broadly accepted, it was found that OO technologies could be used not only in programming, but also in system analysis and design known as OOAD.

The evolution of OOAD technologies in software engineering has resulted in the development of UML in 1995. UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML 1.1 was released in 1997 and recognized as a *de facto* OO modeling standard supported by the Object Management Group (OMG). The current version of UML, Version 1.3, was published in 1999 that ameliorated the legacy problems of UML 1.1 and rectified some bugs discovered in UML 1.2. A collaborate international effort for revision of UML towards Version 2.0 [KOB 99] in 2001 and beyond is going on actively.

Parallel to UML, similar technologies exist such as OML – the Open Modeling Language based OPEN (Object-oriented Process, Environment and Notation) [OPE 99]. Henderson-Sellers and Firesmith [HEN 99] compared two of the third-generation OO modeling approaches: UML and OML. OML is one of the premier third-generation, public domain, fully object-oriented methodology/process. OPEN provides a metamodel-based framework that can be tailored to individual domains or projects taking into account personal skills, organizational culture and requirements peculiar to each industry domain. OML models a pair of two-dimensional matrices which provide a tailoring capability in term of probabilistic links between activities of the lifecycle and tasks which are the smallest unit of work within OML. A second two-dimensional matrix then links the task to the techniques that provide the way the goal can be achieved.

Barbier and Henderson-Sellers [BAR 00] reviewed and provided perspectives on OO technology development for the next generation object modeling languages. They predicted that current alterations and evolution will lead to new versions of UML and OML, possibly to the creation of a new fourth-generation modeling language with strong differences from current ones.

Although, the UML semantics formalization effort and the Analysis and Design Platform Task Force (PTF) for UML 2.0 has recognized a number of needs for real-time extension of the UML capability as a matured object-oriented system modeling language [OMG 99b], alternative approaches have not been decided yet.

This paper describes UML architecture and technologies in a formal and hierarchical approach. The usability of UML is analyzed by mapping the UML diagrams onto a set of UML views. Requirement and extension of the UML capability to real-time software system specification and description are explored, and a real time process-algebra-based approach [WAN 00a] for extending UML's descriptivity for real-time system modeling is provided. In addition, findings and improvement approaches on UML in real-world applications are discussed.

2. Formal Description of UML Architecture

Formal description technologies possess the properties of precise, neat, and less ambiguity. It is demonstrated in this paper that the architecture and usability of UML defined in a large volume of reference documents can be described rigorously in a few pages. This section describes the top level structure, views, diagrams, and usability of UML. A variation of EBNF (Extended Backus Naur Form) [AHO 74] is adopted to describe UML, where “|” and “||” means “or” and “and,” respectively.

2.1 The high level structure of UML

UML, as a visual specification and description language for software systems, consists of notations, syntaxes, and semantics as shown in Expression 1.

$$\begin{aligned}
 \text{UML} = & \text{Notation} \quad // \text{UML visual notations} \\
 & || \text{Syntax} \quad // \text{UML diagrams} \\
 & || \text{Semantics} \quad // \text{Natural language descriptions} \quad [1]
 \end{aligned}$$

As shown in Expression 1, the syntaxes of UML are described by a set of diagrams and their drawing roles in nature language. The semantics of UML are explained and interpreted in natural language. In addition, usage of each visual diagram and view is shown by examples and cases.

A UML model of a software system can be described by four layers known as the meta-metamodel, metamodel, model, and use object layers as shown in Expression 2.

$$\text{UML-Model} = \text{meta-metamodel} || \text{metamodel} || \text{model} || \text{use object}$$

$$= \textit{use-case} \parallel \textit{class} \parallel \textit{state} \parallel \textit{interaction} \\ \parallel \textit{implementation} \parallel \textit{deployment} \quad [2]$$

The UML model described by Expression 2 can be further extended and refined with definitions as shown in Expressions 3 – 6.

$$\textit{Meta-metamodel} = \textit{OMG MOF} \\ \parallel \textit{Defined by OMG meta object facilities (MOF)} \quad [3]$$

$$\textit{Metamodel} = \textit{Foundation} \quad \parallel \textit{A package defines syntaxes of static behavior} \\ \parallel \textit{models} \\ \parallel \textit{Behavior} \quad \parallel \textit{A package defines syntaxes of dynamic} \\ \parallel \textit{behavior models} \\ \parallel \textit{Management} \quad \parallel \textit{A package defines the semantics for grouping and} \\ \parallel \textit{managing model elements} \\ = (\textit{core} \parallel \textit{extension mechanism} \parallel \textit{data types}) \\ \parallel (\textit{use-case} \parallel \textit{activity} \parallel \textit{statechart} \parallel \textit{collaboration}) \\ \parallel (\textit{model management}) \quad [4]$$

$$\textit{Model} = \textit{class} \parallel \textit{attribute} \parallel \textit{operation} \quad [5]$$

$$\textit{Use object} = \textit{objects} \quad \parallel \textit{An instance of a model} \quad [6]$$

2.2 UML views

As a physical system may be projected by different viewpoints, a software system can be described by different views in different development phases, such as requirements acquisition, design, implementation and maintenance; or according to roles of actors in development, such as analysts, designers, users, and testers.

Definition 1. A UML view is a visual conceptual model of a system that illustrates aspects of the system by different angles and by different roles in the system development.

The architecture of a software system can be modeled by a number of views, such as the use-case, logical, process, implementation and deployment views [UML 99, RUM 98] as described by Expression 7:

A UML view can be defined by Expression 7 as follows:

$$\textit{UML-View} = \textit{use-case} \parallel \textit{logical} \parallel \textit{process} \parallel \textit{implementation} \\ \parallel \textit{deployment} \quad [7]$$

Each of the above UML views is a projection of a system aspect and a set of coherent behaviors.

Definition 2. The use-case view describes system behaviors that are seen by a specific point of view, such as end users, analysts, designers, and/or testers.

A use-case describes an actor's view of partial functions of a software system. The use-case view captures the static structure of a system by use case diagrams, and describes system dynamic behaviors by interaction, statechart, and activity diagrams.

Definition 3. The logical view describes functional requirements of a system that consists of classes, interfaces, and collaborations.

The static behaviors of the logic view are described by class and object diagrams. The dynamic behaviors of the logic view are described by interaction, statechart, and activity diagrams.

Definition 4. The process view describes the performance, scalability, and throughput of a system, which consists of tasks, threads, processes, and interactions.

The static behaviors of the process view are described by class and object diagrams. The dynamic behaviors of the process view are described by interaction, statechart, and activity diagrams.

Definition 5. The implementation view describes system components, files, configuration, and management.

The static behaviors of the implementation view are described by UML component diagrams. The dynamic behaviors of the implementation view are described by interaction, statechart, and activity diagrams.

Definition 6. The deployment view describes system hardware platform configuration and topology, such as distribution, installation, and delivery of system components.

The static behaviors of the deployment view are described by deployment diagrams. The dynamic behaviors of the deployment view are described by interaction, statechart, and activity diagrams.

2.3 UML diagrams

UML diagrams are visual tools for describing system design views, components, processes, and human-computer interactions. UML diagrams consist of visual elements that support modeling and visualization of a software system.

Definition 7. A UML diagram is a visual modeling tool for describing a view of a system's architecture, components, and its static and dynamic behaviors.

A summary of types of UML diagrams can be defined by Expression 8:

```

UML-Diagram = class      // Definition 8
              | object   // Definition 9
              | sequence // Definition 10
              | collaboration // Definition 11
              | statechart // Definition 12

```

<i>activity</i>	// <i>Definition 13</i>	
<i>use-case</i>	// <i>Definition 14</i>	
<i>component</i>	// <i>Definition 15</i>	
<i>deployment</i>	// <i>Definition 16</i>	[8]

Definition 8. A class diagram is a set of classes, interfaces, and collaborations, and the description of their relationships by links.

Definition 9. An object diagram is a set of objects and the description of their relationships by links.

Definition 10. A sequence diagram is an interaction diagram that describes the order and relative timing of sequential and parallel objects and processes.

Definition 11. A collaboration diagram is an extended sequence diagram that describes the interaction and organization of objects that exchange messages.

Definition 12. A statechart diagram is a visual representation of a state machine of a software system. A state machine consists of state, events, transitions, and activities.

Definition 13. An activity diagram is a special kind of statecharts that describes the flow of activities within a system.

Definition 14. A use-case diagram is a set of use cases oriented to actors of the system and their relationships.

Definition 15. A component diagram is a set of components with descriptions of their relationships.

Definition 16. A deployment diagram is the configuration of all run-time components and their relationships.

2.4 Usability of UML

The usage of UML can be defined by a mapping between the UML views and the UML modeling technologies known as visual diagrams. A detailed mapping of the views onto related diagrams, or vice versa, for both static and dynamic behaviors, is shown in Table 1.

Table 1. Mapping between UML views and visual technologies

Technology \ View	Usage	Visual Modeling Diagrams									
		Class	Object	Sequence	Collabo-ration	State chart	Activity	Use case	Compo-nent	Deploy-ment	
Use-case	User requirements capture and interpretation	S				✓			✓		
		D			✓		✓	✓	✓		
Logical	Static and dynamic behaviors	S	✓	✓							
		D			✓		✓	✓			
Process	Message flows	S	✓	✓							
		D			✓	✓	✓	✓			
Implemen-tation	Work units and configuration	S								✓	
		D			✓	✓	✓	✓			
Deployment	Processes and component allocation	S									✓
		D			✓	✓	✓	✓			

Note: S – Static behaviors, D – Dynamic behaviors

3. Extension of the UML Descriptivity to Real-Time Systems

Real-time software systems are inherently complicated because they are two-dimensional in specification with a logic dimension and a time dimension. Unlike the non real-time software within that if the system logic is correct then the system's behaviors are correct, a real-time software requires that both system logic and timing should be correct in implementation and at run-time. From a functional point of view, real-time system specification requires a notation system that possesses precisely timing control (absolute and relative timing), event capture, process dispatch, direct device/memory/port access, dynamic memory management, etc.

Although an OMG real-time analysis and design initiative was established in 1998 [OMG 99a], the modeling of the above techniques for real-time systems have yet to be implemented in UML. The current development of UML V2.0 has intended to address the extensibility of UML for real-time applications by openly call for proposals [OMG 99b].

The flexibility of UML is that it provides an extensibility mechanism consistent with a four-layer metamodel architecture. Via the UML profile specifications, extensions and user demands for language extension and customization can be supported.

This section describes a new approach to extend UML to real-time system modeling and specification by using an extended real-time process algebra [WAN 00a]. Algebra is a form of mathematics that simplifies difficult problems by using symbols to represent constants, variables, calculations and their relations. Algebra enables complicated problems to be expressed and investigated in a formal and

rigorous way. Hoare [HOA 85] and Milner [MIL 89] developed a way to represent computer communicating and concurrent processes by algebra, known as process algebra. Wang et al. extended Hoare's CSP-based process algebra to real-time process algebra [WAN 00a].

3.1 Meta-Processes

This subsection describes a set of meta-processes where “meta” means the elementary and primary processes in a system. Complex processes can be derived from the meta-processes by a set of process combinatory roles.

3.1.1 System Dispatch

Definition 17. System dispatch is a meta-process that acts at the top level of a process system for dispatching and/or executing a specific process according to system timing or a predefined event table.

A system dispatch process, *SYSTEM*, can be denoted by:

$$SYSTEM = \{ t_i \Rightarrow P_j \vee e_i \Rightarrow P_j \}, i, j = 1, 2, 3, \dots \quad [9]$$

where $t_i \Rightarrow P_j$ means a system timing t_i triggers a process P_j , and $e_i \Rightarrow P_j$ means an event e_i triggers the process P_j .

3.1.2 Assignment

Definition 18. Assignment is a meta-process that assigns a variable x with a constant value c , i.e.:

$$x := c \quad [10]$$

3.1.3 Get System Time

Definition 19. Get system time is a meta-process that reads the system clock and assigns current system time t_i to a system time variable t .

A get-system-time process, $@T$, can be denoted by:

$$@T = t := t_i \quad [11]$$

3.1.4 Synchronization

Synchronization between processes can be classified into two types: time synchronization and event synchronization.

Definition 20. Time synchronization is a meta-process that holds a process's execution until moment t of the system clock.

A time synchronization process, *SYNC-T*, can be denoted by:

$$SYNC-T = @(t) \quad [12]$$

Definition 21. Event synchronization is a meta-process that holds a process's execution until event e occurs.

An event synchronization process, $SYNC-E$, can be denoted by:

$$SYNC-E = @(e) \quad [13]$$

3.1.5 Read and Write

Definition 22. Read is a meta-process that gets a message from a memory location or system port.

A read process, $READ$, which gets a message m from a memory or port location l can be denoted by:

$$READ = l ? m \quad [14]$$

Definition 23. Write is a meta-process that puts a message into a memory location or system port.

A write process, $WRITE$, which puts a message m into a memory or port location l can be denoted by:

$$WRITE = l ! m \quad [15]$$

3.1.6 Input and Output

Definition 24. Input is a meta-process that receives a message from a system I/O channel which connects the system to other systems.

An input process, IN , which receives a message m from channel c can be denoted by:

$$IN = c ? m \quad [16]$$

Definition 25. Output is a meta-process that sends a message to a system I/O channel which connects the system to other systems.

An output process, OUT , which sends a message m to a channel c can be denoted by:

$$OUT = c ! m \quad [17]$$

3.1.7 Stop

Definition 26. Stop is a meta-process that terminates a system's operation. A stop process is denoted by $STOP$.

3.2 Process relations

This subsection develops a set of relational operations for describing relationships between processes. The relational operators, such as of sequential, branch, parallel, iteration, interrupt, and recursion, define the rules to form combinatorial processes from simple and meta-processes.

3.2.1 Sequential process

The sequential relation is the simplest relation between processes. This subsection discusses two types of sequential processes: serial and pipeline processes.

Definition 27. Serial is a process relation in which a number of processes are executed one by one.

A relational operator “;” is adopted to denote the serial relation between processes. Assuming two processes, P and Q , are serial, their relation can be expressed as follows:

$$P ; Q \quad [18]$$

Expression 18 reads, “P followed by Q.”

Definition 28. Pipeline is a process relation in which a number of processes are interconnected to each other, and a process takes the output of the other process(es) as its input.

A relational operator, \gg , is adopted to denote the pipeline relation between processes. Assuming two processes, P and Q , are pipelined, their relation can be expressed as follows:

$$P \gg Q \quad [19]$$

Expression 19 reads, “P output to Q.”

3.2.2 Branch process

The branch relation describes the selection of processes based on a conditional event. This subsection discusses three types of branch processes: the event-driven choice, the deterministic choice, and the nondeterministic choice.

Definition 29. The event-driven choice is a process relation in which the execution of a process is determined by the event corresponding to the process.

A relational operator, $|$, is adopted to denote an event-driven choice between processes. Assuming process P accepts event a as input, and Q accepts b , an event-driven choice can be expressed as follows:

$$(a \rightarrow P \mid b \rightarrow Q) \quad [20]$$

Expression 20 reads, “a then P choice b then Q.”

Definition 30. The **deterministic choice** is a process relation in which a set of processes are executed in an externally determinable order.

A relational operator, $[\]$, is adopted to denote the relation of deterministic choice between processes. Assuming two processes, P and Q , are related to each other by deterministic choice, their relation can be expressed as follows:

$$P [\] Q \quad [21]$$

Expression 21 reads, “P choice Q.”

Definition 31. The **nondeterministic choice** is a process relation in which a set of processes are executed in a nondetermined or random order dependent on run-time conditions.

A relational operator, \sqcap , is adopted to denote the relation of nondeterministic choice between processes. Nondeterministic choice is also known as “or”. Assuming two processes, P and Q , are related to each other by nondeterministic choice, their relation can be expressed as follows:

$$P \sqcap Q \quad [22]$$

Expression 22 reads, “P or Q,” or “P nondeterministic choice Q.”

3.2.3 Parallel process

Parallel describes the simultaneous and concurrent relation between processes. This subsection discusses three types of parallel processes: the synchronous parallel, the concurrency, and the interleave processes.

Definition 32. The **synchronous parallel** is a process relation in which a set of processes is executed simultaneously according to a common timing system.

A relational operator, \parallel , is adopted to denote a synchronous parallel relation between processes. Assuming two processes, P and Q , are synchronous parallel between each other, their relation can be expressed as follows:

$$P \parallel Q \quad [23]$$

Expression 23 reads, “P in parallel with Q.”

Definition 33. **Concurrency** is an asynchronous process relation in which a set of processes are executed simultaneously according to independent timing systems, and each such process is executed as a complete task.

A relational operator, $[\]$, is adopted to denote a concurrent relation between processes. Assuming two processes, P and Q , are concurrent between each other, their relation can be expressed as follows:

$$P \parallel Q \quad [24]$$

Expression 24 reads, “P concurrent with Q.”

Definition 34. Interleave is an asynchronous process relation in which a set of processes are executed simultaneously according to independent timing systems, and the execution of each such process would be interrupted by other processes.

A relational operator, \parallel , is adopted to denote an interleave relation between processes. Assuming two processes, P and Q , are interleave-related between each other, their relation can be expressed as follows:

$$P \parallel Q \quad [25]$$

Expression 25 reads, “P interleave Q.”

3.2.4 Iteration process

The iteration relation describes the cyclic relation between processes. This subsection discusses two types of iterative processes: the repeat and the while-do processes.

Definition 35. Repeat is a process relation in which a process is executed repeatedly for a certain times.

A relational operator, $()^n$, is adopted to denote the repeat relation for iterated processes. Assume a process, P , is repeated for n times, the combinatorial process can be expressed as follows:

$$(P)^n \quad [26]$$

where $n \in \mathbb{Z}$, $n \geq 0$, and P can be a simple process or a combinatorial process. Expression 26 reads, “repeat P for n times.”

Definition 36. While-Do is a process relation in which a process is executed repeatedly when a certain condition is true.

A relational operator, $*$, is adopted to denote the while-do relation for iterated processes. Assuming a process, P , is iterated until condition γ is not true, the while-do process can be expressed as follows:

$$\gamma * P \quad [27]$$

where P can be a simple process or a combinatorial process. Expression 27 reads, “while γ do P .”

3.2.5 Interrupt process

The interrupt relation describes execution priority and control-taking-over between processes. This subsection discusses interrupt and interrupt-return processes.

Definition 37. **Interrupt** is a process relation in which a running process is temporarily held before termination by another process that has higher priority, and the interrupted process will be resumed when the high priority process has been completed.

A relational operator, \nearrow , is adopted to denote the interrupt relation between processes, and between processes and the system. Assuming process P is interrupted by process Q , the interrupt relation can be expressed as follows:

$$P \nearrow Q \quad [28]$$

A special case of interrupt is between a process P and the operating environment $SYSTEM$, i.e.:

$$P \nearrow SYSTEM \quad [29]$$

In such a case, process P is interrupted by the system dispatcher, $SYSTEM$, and will not automatically return from the interruption, except the system invokes P for a new mission.

Expressions 28 and 29 read, “ P interrupted by Q ,” or “ P interrupted by $SYSTEM$ ”, respectively.

Definition 38. **Interrupt return** is a process relation in which an interrupted process resumes its running from the point of interruption.

A relational operator, \searrow , is adopted to denote an interrupt return between processes. Assuming the running condition of an interrupted process P is regained from process Q , the interrupt return relation can be expressed as follows:

$$Q \searrow P \quad [30]$$

Expression 30 reads, “ Q interrupt returned to P .”

3.2.6 Recursive process

Recursive technology is frequently used in programming to simplify procedure structure. In software system modeling, recursive processes are very important to specify neat and provable system functions. For example, a simple everlasting clock, $CLOCK$, which does nothing but tick, i.e. $CLOCK = tick \rightarrow tick \rightarrow tick \rightarrow \dots$. Its recursive expression can be defined as simply as follows:

$$CLOCK = tick \rightarrow CLOCK \quad [31]$$

Definition 39. A generic recursive process, P , is defined as:

$$P = \mu X \bullet F(X) \quad [32]$$

where μX indicates a recursion of a local variable X that represents the given process P ; and $F(X)$ is a guarded expression of process X .

For example, a generic recursive representation of *CLOCK* defined in Expression 31 can be given as follows:

$$\begin{aligned} \text{CLOCK} &= \mu X \bullet F(X) \\ &= \mu X \bullet (\text{tick} \rightarrow X) \end{aligned}$$

3.3 An approach to extend UML for real-time system modeling

So far, a set of 10 meta-processes and 11 notations of process relations have been defined. A process relation set, R , can be summarized below:

$$R = \{;, \gg, |, [], \square, ||, [], ||, \mu X \bullet F(X), ()^n, \gamma * P\} \quad [33]$$

The extended real-time CSP (RT-CSP) [WAN 00a, HOA 85, WAN 99a] meta-processes and relations described above provides a set of fundamental mechanisms required in real-time system modeling. Adoption of these real-time mechanisms for UML will significantly increase the descriptivity of UML for real-time systems. Of course, corresponding visual icons will be developed and standardized for adaptation of the extended RT-CSP notations in UML.

4. Open Issues in Future UML Development

In order to present both sides of the coin, this section analyzes a number of open issues and limitations in applications of UML, which would influence or affect the future development of UML. In applications [WAN 00b, BAR 00, WAN 99b] the author found a number of limitations of UML in large-scale software specification and description as shown below. One reason of these issues is caused by the limitations of UML itself, while the others are because of the inherited properties of visual technologies for describing complicated systems.

4.1 Graphical notation vs. textual notation

It is observed in applications that a text version of UML notations (UML-TX), which corresponding to the current graphical notations (UML-GR), is required for establishing a neat, easy editing, rigor, and formal notation system. The UML-TX version will be identical in syntaxes and semantics with the counterpart UML-GR,

but more formal, neat, and less ambiguity, as well as easy on-line editing and processing.

It is widely accepted that visual technologies are intuitive, easy comprehension, and extremely useful in phases of system requirements capture and communication with customers. However, visual descriptions are difficult in generation, editing, and for preserving consistency in complicated systems. Therefore, from the usability point of view, UML-GR is more suitable to practitioners and users, while UML-TX is oriented to researchers and provable system analysts. UML-TX may provide further clarification for the UML-GR syntaxes and semantics, and may extend the capability of UML that was limited by inherited nature of visual technologies.

4.2 Object-orientation or non object-orientation

Object technologies are proven conceptual modeling technology and generically applicable in software system design, analysis and implementation. Object technologies provide a new approach to software component reuse by inheritance. While, in non object-oriented approaches, reuse may be implemented by clip and paste of code or by direct calls of specific software components from system software or user libraries.

However, good technologies may be implemented imperfectly when their disadvantages were extensively amplified. Current object technologies are one of such cases. For example, by using common OO languages, programmers must know details of a complicated structure of the foundational class hierarchy provided by a compiler in order to inherit or reuse a software component and/or a data type. This approach significantly increases the difficulty of mastering OO languages, and generates inherent complexity, subsequently, in OO software testing and maintenance.

Moreover, inheritance of a made class hierarchy in an OO language is not tailorable. Programmers have to inherit anything that contained in a given class and its ancestors, even just a small portion of functions of the class intended to be reused. This inefficient implementation of object technology results in a special phenomenon so called the 'fatware,' of which only an empty object encapsulation in common OO languages costing more than 10 Kbytes in implementation. Also, inheritance from the root of a class system provided by vendors causes difficulty in testing and maintenance. For example, a recent project reported that more than half of the bugs in porting a Borland C++ based software into MS C++ environment were caused by the incompatibility between the two foundational class structures.

On the basis of the above discussions it is considered that the new trend in component-based software engineering may not necessary in the way of object-orientation. Therefore, the future UML development may need consider a more generic approach to software system modeling for covering both object and non-object technologies.

4.3 Limitations of UML architecture as a notation system

Conventionally, a matured graphical engineering notation system provides a hierarchical diagram system. For example, in electrical and electricity engineering, graphical notations consist of principle schematic diagrams (at system conceptual level), logical circuit diagrams (at detailed level), and component diagrams (at extended detailed level) from a top-down hierarchy. More importantly, in the conventional engineering notation systems, each level of extension in detail is a refinement of a true subset of the description at the higher level(s).

However, UML provides a larger set of visual diagrams and views that can be used by multi-purposes, and the UML visual tools are functionally overlapped as analyzed in Section 2 and summarized in Table 1. These features seem to be continued in the future extension of UML [KOB 99]. Such contradiction to the convention of common engineering notation systems, at system conceptual, detail and extended-detail levels, results in a source of difficulty in consistency reservation in a UML specification for complicated systems. Perhaps this nature of UML is inherited from its history of development as a hybrid combination of various notations in OO modeling and description, especially the methods developed by Booch [BOO 86], Jacobson [JAC 92], Rumbaugh [RUM 91], SDL [CCI 88], and MSC [ITU 94]. Although, using different diagrams for different views and in different processes is an acceptable approach, the UML introduction of multiple diagrams with various syntaxes and semantics is considered lack of cohesion and consistency as a matured engineering notation system.

On the basis of the above analysis, it is considered that the future development of UML would be on elicitation of the fundamental descriptivity of software at conceptual, detail, and programming language levels, in supplement with the current focus on extension of types of UML diagrams.

4.4 Limitations of the UML use-cases

UML use-cases and views are a specific and subjective perception on an objective software system. The UML use-case was designed for easily capturing user requirements and for presenting system design ideas with viewpoints of various roles or actors in the system. However, it is found there is considerable difficulty to exhaust the use-case views for a complicated system. Therefore, the completeness and correctness of the use-cases for large-scale systems are very difficult to prove. In addition, most of the user-cases are often turned out to be highly overlapped or similar for describing a complicated system.

These observations identify a requirement for tailorability of the UML notations. It is suggested that a use-case should be a strict subset of the statechart or sequence diagram, rather than a separate diagram using additional syntax and with additional semantics. For example, a simplified deployment diagram of MS Word is shown as in Figure 1. A number of use-cases for various users, such as writers, editors, proof

readers, etc., should be derived as subsets of a unified UML logical diagram based on Figure 1 as marked by related boxes.

File	Edit	Format	Tools	Help
new	typing	page setup	find	contents
open	undo	font	view	index
close	clear	paragraph	spelling	context-dependent help
save	cut	columns	language	
save as	copy	tabs	word count	
delete	paste	change case	options	
print	insert	style		
exit	replace	tool bars		
	table			
	figure			
	track change			

Figure 1. High-level specification of MS Word as a use-case example

5. Conclusions

This paper has presented a formal description of UML technologies for visualization of software system specification and modeling, and analyzed the usability of UML views and diagrams. Requirements and extension of the UML capability to real-time software system specification and description have been explored. Limitations of the UML technologies in real-world applications are discussed.

Because UML is still a maturing technology toward Version 2.0 in the coming years [KOB 99, BAR 00], a number of open issues have been discussed in this paper based on the author’s work in IEEE SNPG. Major findings of this work are that:

- (a) UML can be extended to make it suitable for real-time system specification and modeling on the basis of existing work such as RT-CSP and process algebra. There are 10 meta-processes and 11 process relations identified in the real-time process algebra [WAN 00a] as described in Section 3, which is useful for UML extension in the future.
- (b) The UML graphical notations (UML-GR) may be balanced by textual notations (UML-TX), in order to provide formal and rigorous syntaxes and semantics for UML. The former is especially useful for practitioners and customers for requirement capture, and the latter is suitable for system analysts and researchers who are seeking rigorous and reliable solutions in complicated system specification and modeling.
- (c) From the architecture point of view, UML is a combination and integration of a larger set of visual diagrams and views, and each of these can be used by multi-

purposes. The philosophy of UML is that hybrid visual diagrams may be adopted in different development processes for improving flexibility and descriptivity. This is interestingly different from the common software engineering philosophy that requires system specification and design be carried out by stepwise refinement with a coherent notation in system design, detailed design and implementation.

- (d) The UML use-cases are found most suitable for small and medium sized applications. For large-scale complicated system specification by UML, completeness and consistency reservation become significantly difficult in applications. This may be improved by adopting a textual version of UML as discussed in point (b) of this Section.

Acknowledgments

This work is partially supported by a Swedish National Research Project PALBUS – Dependable Distributed System Specification and Implementation, and is related to the author's work in the IEEE Software Notation Planning Group (SNPG) toward the standardization of software engineering notation systems. The author would like to acknowledge the support of the funding organization, IEEE SNPG, and the industry partners.

The author would like to thank the anonymous reviewers for their valuable comments that have improved the presentation of this paper.

References

- [AHO 74] AHO A.V., HOPCROFT J.E, ULLMAN J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA., 1974.
- [BAR 00] BARBIER F., HENDERSON-SELLERS, B., The Next Generation on Object Modeling Languages for Software Engineering, in D. Patel and Y. Wang eds., Special Volume on Comparative Software Engineering, *International Journal of Annals of Software Engineering*, Baltzer Science Publishers, Oxford, Vol.10, 2000.
- [BOO 86] BOOCH G., Object-Oriented Development, *IEEE Transactions on Software Engineering*, IEEE Computer Society Press, Vol. 12, No.2, 1986.
- [CCI 88] CCITT, Recommendation Z.100 – *Specification and Description Language SDL*, Blue Book, Volume VI.20 – Vol.24, ITU, Geneva, 1998.
- [HOA 85] HOARE C. A. R., *Communicating Sequential Processes*, Prentice-Hall International, Englewood Cliffs, NJ., 1985.
- [HEN 99] HENDERSON-SELLERS B., FIRESMITH D.G., Comparing OPEN and UML: The Two Third-Generation Development Approaches, *Information and Software Technology*, Vol. 41, 1999, pp.139-156.

- [ITU 94] ITU-T, *Recommendation Z.120: Message Sequence Chart (MSC)*, Geneva, 1994, pp. 1-35.
- [JAC 92] JACOBSON I. et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.
- [KOB 99] KOBRYN C., UML 2001: A Standardization Odyssey, *Communications of the ACM*, Vol. 42, No.10, 1999, pp. 29 – 37.
- [MIL 89] MILNER R., *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ., 1989.
- [OMG 99a] OMG, UML Profile for Scheduling, Performance, and Time, *OMG Request for Proposal*, OMG ad/99-03-13, 1999.
- [OMG 99b] OMG, UML 2.0 Analysis and Design Platform Task Force (PTF), *OMG Request for Information (V1.0)*, ad/99-08-xx, 1999.
- [OPE 99] <http://www.open.org.au/>.
- [RUM 91] RUMBAUGH J. et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
- [RUM 98] RUMBAUGH J., JACOBSON I, BOOCH G., *The Unified Modeling Language Reference Manual*, ACM Press, New York, 1998.
- [UML 99] UML Revision Task Force, *OMG Unified Modeling Language Specification, V.1.3*, Document ad/99-06-08, Object Management Group, June, 1999.
- [WAN 00a] WANG Y., KING G., *Software Engineering Processes: Principles and Applications*, CRC Press, USA, ISBN: 0849323665, 2000, pp.1-746.
- [WAN 00b] WANG Y., KING G., FAYAD M., PATEL D., COURT I., STAPLES G., ROSS M., On Built-in Tests Reuse in Object-Oriented Framework Design, *ACM Journal on Computing Surveys*, Vol.32, No.1, 2000, March, New York.
- [WAN 99a] WANG Y., CHOULDURY I., PATEL D., PATEL S., DORLING A., WICKBERG H., KING, G., On the Foundations of Object-Oriented Information Systems, *L'Object: Logiciel Bases de Donnees Reseaux*, Vol.5, No.1, Feb., 1999, pp.9-27.
- [WAN 99b] WANG Y., KING G., PATEL D., PATEL S., DORLING A., On Coping with Software Dynamic Inconsistency at Real-Time by the Built-in Tests, Special Volume on Real-Time Software Engineering, *International Journal of Annals of Software Engineering*, Baltzer Science Publishers, Oxford, Vol.7, 1999, pp.1283-296.

Yingxu Wang is Professor of Software Engineering in the Dept. of Electrical and Computer Engineering at the University of Calgary, Canada. He was a Visiting Professor in Computing Laboratory at Oxford University during 1995. He received a PhD in software engineering from The Nottingham Trent University / Southampton Institute, UK, where he has been honored an academic title of Visiting Professor since 1999. He is a member of IEEE TCSE/SESC, ACM, and ISO/IEC JTC1/SC7. He has accomplished a number of European Commission and industry funded research projects as manager and/or principal investigator, and has published over 100 papers in software engineering and computer science. He is the lead author of a recent book on “Software engineering Processes: Principles and Applications.” He has won a dozen research achievement and academic teaching awards in the last 20 years.

