Chapter XIII
Software Modeling Processes: UML–xUML Review

Roy Gelbard
Bar-Ilan University, Israel

ABSTRACT

Applications require short development cycles and constant interaction with customers. Requirement gathering has become an ongoing process, reflecting continuous changes in technology and market demands. System analysis and modeling that are made at the initial project stages are quickly abandoned and become outmoded. Model driven architecture (MDA), rapid application development (RAD), adaptive development, extreme programming (XP), and others have resulted in a shift from the traditional waterfall model. These methodologies attempt to respond to the needs, but do they really fulfill their objectives, which are essential to the success of software development? Unified modeling language (UML) was created by the convergence of several well-known modeling methodologies. Despite its popularity and the investments that have been made in UML tools, UML is not yet translatable into running code. Some of the problems that have been discovered have to do with the absence of action semantics language and its size. This chapter reviews and evaluates the UML evolution (UML2, xUML), providing criteria and requirements to evaluate UML and the xUML potential to raise levels of abstraction, flexibility, and productivity enhancement. At the same time, it pinpoints its liabilities that keep it from completely fulfilling the vision of software development through a continuous exactable modeling process, considered to be the future direction for modeling and implementation.

INTRODUCTION

In his book, Evitts describes the beginnings of UML tools (Evitts, 2000). The context prompting the development of UML was the increasing complexity of software which began in the 90s, when technologies (tools) that could deal with a network and information-driven world did not yet exist. In 1991, Malone and Rockart described expectations that would soon emerge from all quarters. They noted that whenever people work together, there is a need to communicate so as to make decisions, allocate resources, and provide and receive products and services at the right time and place. However, in the early 90s, methodologies were rarely supported, either by common modeling tools, traditional methodologies (based upon process charts, ERD, and DFD), or object oriented methodologies. The semi-standard development process, the "waterfall," was convenient, albeit unperfected, whereas object-oriented provided none of these comforts, and the general opinion was that very few of its efforts had any real advantages over mainstream approaches.

In early 90s, the rise of Java, the standardization of C++, the birth and rebirth of CORBA, and the emergence of pattern languages for software design attracted a great deal of attention and popularity to UML. In June 1996, Rational released the 0.9 revision of UML, and then later on January 1997, Rational's 1.0 spec reached the market. In September 1997, Rational's UML 1.1 was combined with the OMG's UML proposal to create the final product that was called UML 1.0.

The current chapter evaluates the extent to which the UML can be used to support the modeling process, providing not only better communication among system analysts and developers. Primarily, it examines productivity enhancement through generating capabilities of wider range of software elements based upon modeling definitions.

BACKGROUND REVIEW

A. From UML 1 to UML 2.0

The scope of the UML has recently broadened. It is no only longer used to describe software systems, but now also business processes. With the service-oriented architect (SOA) and model driven architecture (MDA) initiatives, it has evolved to describe and automate business processes (activity diagram is a UML variation of the traditional process diagram), as well as become a language for developing platform-independent systems.

Earlier versions of the UML standard did not describe what it meant to support the standard. As a result, UML tool vendors were free to support incomplete UML features, and converting models from one tool to another was often extremely difficult, if not impossible.

UML 2.0 defines 38 compliance points (Ambler, 2004; Bjorkander & Kobryn, 2003). A compliance point is an area of UML, such as use cases. All implementations are required to implement a single compliance point, the kernel. The other 37 compliance points are currently optional. Evaluating modeling tools in light of these compliance points helps clarify which model elements are supported, and to what extent. For each compliance point, there are four compliance options. A compliance option determines how compliant a given implementation is. The four options are as follows:

- **No compliance**—the implementation does not comply with the syntax, rules, and notation for a given compliance point.
- **Partial compliance**—the implementation partially complies with the syntax, rules, and notation for a given compliance point.
- **Compliant compliance**—the implementation fully complies with the syntax, rules, and notation for a given compliance point.
- **Interchange compliance**—the implementation fully complies with the syntax, rules, notation, and XMI schema for a given compliance point.

However, UML 2.0 does not address any of UML 1.x's significant deficiencies, namely the lack of **business rule** Modeling, **workflow** modelling, and **user interface** modeling, although there is a business rule working group within the OMG. Several methodologists have suggested approaches to user interface flow modeling and design modeling using UML, but no official effort to develop a common profile exists.

B. Executable UML (xUML)

xUML is a subset of the UML, incorporating action language that allows system developers to build executable domain models and then use these models to produce system source code. Hence, xUML is an executable version of the UML.

The xUML process involves action specification language (ASL) (Raistrick, Francis, Wright, Carter, & Wilkie, 2004). The resulting models can be independently executed, debugged, viewed, and tested. Multiple xUML models can be assembled together to form complex systems with their shared mappings, expressed using ASL (Raistrick et al., 2004). Executable models can then be translated into target implementations. The execution rules of the xUML formalism means that the same models can be translated into a wide variety of target architectures without introducing changes into the models.

The xUML model enables modeling independency concerning hardware and software organization; in the same way typical, a compiler offers independency concerning register allocation and stack/heap organization. Furthermore, just as a typical language, compiler makes decisions about register allocation and the like for a specific machine environment, so a xUML model compiler makes decisions about a particular hardware and software environment, deciding, for example, to use a distributed Internet model with separate threads for each user window, HTML for the user interface displays, and so on.

The xUML methodology suggests a system partitioned into domains or subject matters (Miller & Mukerji, 2003). Each domain is modelled separately. Bridges are defined between domains, with some requirements placed between one domain and another and connector points defined for their exchange of information. The various domains and their dependency relationships are commonly displayed using package diagrams.

According to the OMG MDA approach (Miller & Mukerji, 2003), each xUML model is a platform-independent model (PIM). The mappings between such models are PIM-to-PIM mappings. The translation approach makes use of PIM to platform specific model (PSM) and platform specific implementation (PSI) mappings, in order to achieve executable modeling, large-scale reuse, and pattern-based design.

The xUML does not use all of the UML diagrams and constructs, as many are thought to be redundant. The xUML is intended to precisely model a system; any construct or building blocks that could introduce ambiguity should be left out of the model (Raistrick et al., 2004).

The most fundamental modeling diagrams in xUML are the class and state chart diagrams. Classes should be modeled with some degree of precision. A state machine is attached to each class to describe its dynamic behavior and lifecycle. In other words, in xUML, each class is thought to have a state machine that responds to events. Actions that are taken in response to events or about a certain state are specified precisely using some sort of action language. The specification 1.4 of the UML includes the specification of action semantics even though no concrete syntax for these semantics is specified. Typically, the xUML tool will provide an explanation about the action language syntax or syntaxes that the tool can interpret.

Use case and activity diagrams **are not** an integral part of the xUML but they are recommended as methods for gathering requirements before the model is constructed. Activity diagrams are employed to show the sequence of use cases and branching possibilities. Collaboration and sequence diagrams can be used to gain insight into the system or in some cases for visualizing aspects of the system after it has been built. They, too, are not executable.

C. Model Driven Architecture (MDA)

Model driven architecture initiated by the object management group OMG) aims to place models at the core of the software development process, that is, model driven architecture of both the system and the software. The MDA claim can be regarded as a mere recommendation for how the system should be built, with little guarantee that the system will actually be built as specified. If some design decisions have to be revised, and decisions are taken to build the system differently during the implementation phase, rarely are the design models updated to reflect the decisions that make design models ineffective in the same manner as static documents, which are all ineffective as regards future maintenance.

RAD and other agile methods are based upon the same concept. The typical situations where analysis and design models often end up serving their intended purpose rather poorly, has led to iterative methods, having short iterations that repeat all modeling steps from analysis up to implementation.

Another MDA concept is platform independency. As technology progresses at a fast rate, new platforms are quickly introduced. Software written for a certain platform has very little use when transferred to other platforms—meaning most of the software must then be rewritten from scratch. If we were able to create a platform-independent model that could be translated to fit diverse platforms, many of the problems arising from platform instability could be avoided.

The MDA approach is based upon separation between a computation independent model (CIM), which is a precise business model, stakeholder oriented, uncommitted to specific algorithms or system boundaries, also known as a domain model and platform independent model, which is created at the analysis phase of software development. The PIM therefore is a long-term asset. A platform specific model is generated from the PIM in the design phase with some degree of [automation] autonomy. The PIM and the PSM concepts are not new, but the way they are modeled is. According to the MDA, there are four levels of modeling:

- **M0**—Objects living and interacting in a real system.
- **M1**—Models that define the structure and behavior of those objects. M1 models, on the other hand, are written in a language. These language constructs have to be defined somewhere.
- **M2**—Meta-models, or models about how M1 models are built. For instance, at level M2, one can find the UML meta-model, or a model about how M1 models are written by the language. Meta-models themselves have to be written in a language—that is what the level M3 stands for.
- **M3**—Meta-meta-modeling, defines how meta-models for language modeling can be built, or a language for meta-model refinement.

Theoretically, we could continue this way until we reach an arbitrary modeling level. In other words, there could be level M4 defining a language for M3, and level M5 from which M4 models are instanced, and so on. However, the MDA initiative defined that the M3 model is written with the M3 language, so there are no higher modeling levels, and M3 language is the metadata object facility (MOF), which is one of the OMG standards. The MOF model is written in MOF itself. Meta-models or models defining modeling languages are instances of the MOF model and thus written in the language defined by it. For instance, there is a meta-model for UML and it is written according to the one specified in the MOF model.

Another OMG standard is the XML metadata interchange or XMI. XMI defines model coding and meta-models in XML format. The standard defines XML generating for any MOF compliant language or meta-models. Moreover, because the MOF meta-model is written in MOF, the MOF meta-model itself can also be coded and exchanged in XML, enabling any vendor to make built-in variations or adaptations at the core of UML or any modeling concept.

Profile (UML Profiles) is also an MDA standard. UML profiles are basically a way to define UML dialects using UML standard extensibility mechanisms (stereotypes, tags, etc.). A number of profiles have been created or are being created for some popular platforms; these profiles serve as convenient PSM language for these platforms.

xUML is essentially related to MDA. xUML can be thought of as one way of implementing the MDA concept, with one notable exception (Raistrick et al., 2004). While MDA recommends that a platform-independent model be transformed into a platform-specific one before it is translated into code, xUML skips this intermediate step. Most xUML tools will translate models straight into code without generating a platform-specific model. On the other hand, the xUML model compiler can be thought of as being analogous to a transformation definition, where the rules about the transformation are declared.

COMPARISON ANALYSIS

The current chapter reviews and evaluates UML evolution (UML2, xUML), providing criteria and requirements to evaluate UML and the xUML potential to raise levels of abstraction, flexibility, and productivity enhancement, while pointing out the disadvantages that prevent it from completely fulfilling the vision of software development through a continuous exactable modeling process.

The following table presents these requirements, noting whether a requirement is supported or not. Based on the following data, the missing pieces that still need to be resolved in the upcoming versions of UML can be determined.

The (+) and (–) symbols note the presence and the absence of the relevant required feature.

Table 1. Requirements for executable software model

#	Required Feature	UML 2	xUML
1	Visualization	+	+
2	System Specification	+	+
3	System Documentation	+	+
4	Automatic Update	-	+
5	System Construction	+ (Partial)	+
6	Code Generation	+ (Partial)	+
7	Standardization	+	+
8	Modeling of classes with attributes, operations, and relationship	+ (Enhanced)	+ (Different relationships between classes in diagrams, state machine associated to each class)
9	Modeling of states and behavior of individual classes	+	+ (though the modeling is for a higher tier)
10	Modeling of packages of classes and their dependencies	+	+ (Extended to domains)
11	Modeling of system usage scenarios	+ (Enhanced)	+ (Enhanced)
12	Modeling of object instances with actual attributes in a scenario	+ (Enhanced)	+ (Enhanced)
13	Modeling of actual behavior of interacting instances in a scenario	+ (Enhanced)	+ (Enhanced)
14	Modeling of distributed component deployment and communication	+ (Enhanced)	+ (Enhanced)
15	Modeling of exceptions	+	+
16	Extension: Stereotypes	+	+
17	Extension: Profiles	+ (Enhanced)	+ (Enhanced)

continued on following page

Table 1. continued

#	Requirements	UML 2	xUML
18	Extension: meta-model	+	+
19	Modeling test cases	+	+
20	Scalability and precision of diagrams	+ (Enhanced but still partial)	+ (Limited support by available tools)
21	Gap reduction between design and implementation	-	+ (PIM, PSM)
22	Multiple views of the system	+	+
23	Domain, platforms, and process customization	+	+ (PIM, PSM)
24	Supporting visualization of user interfaces	-	-
25	Supporting logical expressions required for business logic, detailed definition, and design	-	-
26	Supporting organization and authorization structures	-	-
27	Supporting processes and workflow simulation	-	-
28	Supporting technical requirements		
29	Ability to represent the relationship between the design and specific platform	-	+ (PSM)
30	Describing structural and behavioral issues in a way that is easier to survey than in ordinary textual programming languages	+	+ (Enhanced functionality and action specification language, i.e., ASL, support)
31	Affecting translation into code: 1. Compiler is fast and reliable 2. Generated code is fast and robust	-	+ (PIM, PSM)
32	Generating diagrams from code	+	N/a (We always work on the model itself—all updates done on model and the code is generated from the model)
33	Possibility to under-specify unwanted or unavailable properties still to be defined	-	+ (PSM)
34	Possibility to transfer UML models between models	-	+ (partial)
35	Possibility to transfer UML models from one target language to another	-	+ (PSM)

FUTURE TRENDS: CURRENT UML AND XUML HOLDBACKS

As shown in the table, there are major areas of analysis which are not yet covered by the UML, such as user interface, business logic, organization and authorization structured, and so forth. Some of these areas are not adequately supported by the object oriented methodology itself, while others are mainly technical issues not methodological ones, such as affecting translation into code (requirement 31).

The current deficiencies of UML and xUML are listed.

1. One basic deficiency is language inability to support real-life facts and components that have to be defined and established in the system. This includes supporting visualization of user interfaces, logical expressions required for business logic and its detailed definition and design, organization and authorization structures, processes and workflow simulation, and technical requirements (such as performance, response time, etc.). Until we are able to do so, no system can be fully constructed. UML 2 provided enhanced diagrams, but the language was still not fully expressive. Therefore, there are still major significant pieces of code that must be hand-written or revised. There is no doubt that ideally, the entire system should be able to be constructed at the click of a button, but this cannot happen until the language is fully expressive. Until this happens, there is not much benefit in using xUML.

Further UML steps should address this issue and enable system analysts to define entire business functionality, interface components, and technical requirements. This is easier to state than to implement.

2. The action semantics for the UML is a semantic standard alone, not a syntactic standard. This was, presumably a marketing decision, enabling vendors to supply either AMD-CASE tools or development tools, which already contained fully-realized action languages with proprietary syntaxes. One point of view maintains that syntax does not matter; however, it is easy to see that competing syntaxes may constrain the ability of xUML to acquire growth rates. On the other hand, because action languages exist at such a high level of abstraction, they do not require such sharp learning curves as those of third generation languages. Therefore, after learning one action language syntax, modellers could easily learn another, as they would, in essence, be dealing with the same semantic set.

Further UML steps should address this issue in order to enable language and API transparency, and put aside political interests. Again, this is easy to say, but very complicated to apply, not so much because of technical reasons, but rather as a result of "human factors."

3 The original intent of the system analyst is impaired and lost in translation, when translating the UML model to PIM and then to PSM. The resulting code is sometimes awkward. Meta-models allow for the easy addition of new concepts. However, they do not ensure that these concepts will make semantic sense or that the different meta-models, such as UML profiles, will be consistent or orthogonal, with no contradictions with the original.

Further UML steps should be directed towards integrity validation mechanisms, as developers are "developing-oriented" not "version comparing-oriented," which is quite a cumbersome and tedious task.

4. Moreover, the modeller tends to focus not only on the properties s/he is modeling (Rumpe, 2002), but also on execution efficiency. Therefore, the resulting code must include functionality and business accuracy, as well as performance and efficiency.

Further UML steps should be directed towards supporting optimizing models and tools.

5. When using code generators that map UML to a target language, the semantics of the target language as well as its notational capabili-

ties tend to become visible on the UML level (Rumpe, (2002). For example, missing multiple-inheritance in Java may restrict xUML to single inheritance as well. Furthermore, the language's internal concurrency concept, message passing or exception handling may impose a certain dialect on xUML. This will cause UML dialects to be semantically incompatible. In particular, it will not be easy to make it possible to transfer UML models from one target language to another.

Further UML steps should address precise model transformation.

6. Last but not least are "model debugging" and the traceability-closure problem, that is, analysis error detection and closure between requirements and model components. Although the model, at the modeling stage, is only an outline, it has to be checked, debugged, optimized, and tested. Moreover, it has to enable detailed tracing in light of all requirements (both business and technical).

Further UML steps should be directed at "model debugging" and full traceability-closure, which has a huge impact not only on each one of the development cycles, but also over the entire system's life cycle.

CONCLUSION

The vision of generating an entire system at a single click is still not a reality, as is the vision of fully expressive modelling, which is also still not available. Furthermore, current field studies have shown that UML does not fulfill the "modelling vision" we all wish for. Dobing and Parsons studied "How UML is used" (Dobing & Parsons, 2006), and Davies, Green, Rosemann, Indulska, and Gallo published "How do practitioners use conceptual modeling in practice?" (Davies et al., 2006). Both found the same trends, namely that UML diagrams are not clear enough to users or to developers, and recommend that additional demonstrative methods should be employed. Modeling of business logic, user interfaces, requirements (such as performance, response time, etc.), and other system components are still not fully available. The same goes for "model debugging" and detailed tracing and closure, which are crucial to the modeling stage. Technical aspects are also crucial in order to ensure usability and performance, but this seems to be a temporal issue rather than a conceptual one.

In light of the mentioned, the vision of generating an entire system by a single click will take place if and only if modelling languages, theories, and methodologies can overcome their conceptual limitations. In the IT domain, technical limitations are usually temporal, whereas conceptual limitations can lead to a dead-end.

ACKNOWLEDGMENT

I would like to thank David Belostovsky and Daphna Tsur for their contribution to this work.

REFERENCES

Ambler, S. W. (2004, March). What's new in UML 2. *Software Development Online Magazine.* Retrieved December 2006, from http://www.sdmagazine.com/documents/s=815/sdm0402i/

Bjorkander, M. (2000). Graphical programming using UML and SDL. *Computer, 33*(12), 30-35.

Bjorkander, M., & Kobryn, C. (2003). Architecting dystems with UML 2.0. *IEEE Software, 20*(4), 57-61.

Davies, I., Green, P., Rosemann, M., Indulska, M., & Gallo, S. (2006). How do practitioners use conceptual modeling in practice? *Data &Knowledge Engineering, 58*(3), 358-380.

Di Nitto, E., Lavazza, L., Schiavoni, M., Tracanella, E., & Trombetta, M. (2002). Deriving executable process descriptions from UML. In *Proceedings of the 24th International Conference on Software Engineering* (pp. 155-165).

Dobing, B., & Parsons, J. (2006). How UML is used. *Communication of the ACM, 49*(5), 109-113.

Evitts, P. (2000). *A UML pattern language*. New Riders Publishing.

Fei, X., Levin, V., & Browne, J. C. (2001). Model checking for an executable subset of UML. In *Proceedings of the 16th International Conference on Automated Software Engineering* (pp. 333-336).

Fowler, M. (2003). *UML distilled: A brief guide to the standard object modeling language* (3rd ed.). Addison Wesley.

Hölscher, K., Ziemann, P., & Gogolla, M. (2006). On translating UML models into graph transformation systems. *Journal of Visual Languages & Computing, 17*(1), 78-105.

HongXing, L., YanSheng, L., & Qing, Y. (2006). XML conceptual modeling with XUML. In *Proceedings of the 28th international conference on Software engineering* (pp. 973-976).

Jia, X., Steele, A., Qin, L., Liu, H., & Jones, C. (2007). Executable visual software modeling—The ZOOM approach. *Software Quality Journal, 15*(1), 27-51.

Liu, L., & Roussev, B. (2006). *Management of the object-oriented development process*. Idea Group.

Miller, J., & Mukerji, J. (2003, May). MDA guide. *OMG Organization*. Retrieved December 2006, from http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf

Raistrick, C., Francis, P., Wright, J., Carter, C., & Wilkie, I. (2004). *Model driven architecture with executable UML*. Cambridge.

Rumpe, B. (2002). *Executable modeling with UML—A vision or a nightmare*? (Tech. Rep.). Munich University of Technology. Retrieved December 2006, from http://www4.in.tum.de/~rumpe/ps/IRMA.UML.pdf

Seidewitz, E. (2003). Unified modeling language specification v1.5. *IEEE Software, 20*(5), 26-32.

Thomas, D. (2004). MDA: Revenge of the modelers or UML Utopia?. *IEEE Software, 21*(3), 15-17.

KEY TERMS

Agile Software Development: A conceptual framework for undertaking software engineering projects that embraces and promotes evolutionary change throughout the entire life cycle of the project.

Executable UML (xUML): A software engineering methodology that graphically specifies a deterministic system using UML notations. The models are testable and can be compiled, translated, or weaved into a less abstract programming language to target a specific implementation. Executable UML supports MDA through specification of platform independent models (PIM).

Model-Driven Architecture (MDA): A software design approach launched by the object management group (OMG) in 2001.

Object Management Group (OMG): A consortium, originally aimed at setting standards for distributed object-oriented systems, and is now focused on modeling (programs, systems, and business processes) and model-based standards.

Platform-Independent Model (PIM): A model of a software or business system that is independent of the specific technological platform used to implement it.

Rapid Application Development (RAD): A software development process developed initially by James Martin in the 1980s. The methodology involves iterative development, the construction of prototypes, and the use of computer-aided software engineering (CASE) tools.

Software Engineering: The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

System Modeling: An abstraction or conceptual representation used for system illustration.

Unified Modeling Language (UML): A standardized specification language for object modeling. UML is a general-purpose modeling language that includes a graphical notation used to create an abstract model of a system,