Reform may be too little too late to spare software engineers the cognitively overwhelming effort of applying UML to modeling system structure and behavior in a truly unified manner.

BY DOV DORI

WHY SIGNIFICANT UML CHANGE IS UNLIKELY

To become the system architect’s instrument of choice for evolving complex software systems, the Unified Modeling Language must experience a revolution. It must integrate structure and behavior in a single model while becoming simpler and more user friendly. The scope of the required reform is immense, but so is that of some stakeholder resistance. For this reason, despite the necessity, significant reform is unlikely, and the next UML generation will not differ significantly from the current one.

UML problems can be sorted into three main categories: model multiplicity resulting from excess diagram types and symbols; confused behavior modeling; and the obscuring influence of programming languages. Here, I discuss their nature and severity and propose possible remedy. I further suggest that the Object-Process Methodology (OPM) [1] may offer a fresh, user-friendly alternative to UML as a generic system architecting framework due to its single model expressed in intuitive graphics, translated on the fly into a subset of English.

While UML has undoubtedly contributed to streamlining software engineering practices since its commercial standardization some five years ago, the way it was conceived and adopted by the Object Management Group put it on a difficult track for becoming universally usable. UML agglomerates nine diagram types, also called views, or models, declared to be the unified standard. But such a declaration cannot replace unification of the concepts and symbol sets associated with the models, along with removal of the many redundant entities and overlapping notions.

A major problem with UML is the size of its alphabet of more than 150 symbols. No less disturbing is the number of its diagram types. One author has described UML’s mix of notations from different approaches as yielding a “confused unproductive picture” [7]; other authors have found that UML is up to 11 times more complex than other OO methods [6]. The associated model multiplicity problem [5] concerns the fact that not even one of the nine UML models clearly shows an integrated view of the two most prominent and useful system aspects: structure and behavior. Since UML has evolved bottom-up from OO programming concepts, it lacks a system-theoretical ontological foundation [8] encompassing observations about common features characterizing systems regardless of domain [3]. A plausible ontological foundation views systems as composed of objects transformed by processes, which generate, con-
When object-oriented programming (OOP) was first proposed, the goal was to provide a programming paradigm that would enable the design and implementation of large, complex software systems. This was achieved by introducing the concept of objects, which are instances of classes that encapsulate data and behavior. The key idea was to separate the structure of a program (what it does) from its behavior (how it does it), allowing for the modular design and reuse of code.

However, as systems became more complex, it became clear that OOP alone was insufficient to handle the increasing size and complexity of software systems. This led to the development of more advanced modeling techniques, such as UML (Unified Modeling Language), which was designed to provide a comprehensive framework for specifying, visualizing, constructing, and documenting the artifacts of software-intensive systems.

UML introduced a rich set of diagrams and notations that allowed developers to model the static structure of a system (what it is) and its dynamic behavior (how it works). These diagrams included class diagrams, object diagrams, and sequence diagrams, among others.

However, the adoption of UML has not been without its challenges. One of the major issues is the lack of integrated support for modeling both structure and behavior in a single diagram. UML requires developers to use multiple diagrams to represent different aspects of a system, such as class diagrams for structure and sequence diagrams for behavior. This fragmentation can lead to difficulties in maintaining consistency across diagrams and can be a source of confusion for developers.

Another issue is the cognitive load imposed by UML on developers. The use of multiple diagrams can be overwhelming, especially when trying to understand the interactions between different components of a system. This can lead to errors and inconsistencies in the model, which can be difficult to detect and correct.

The lack of integrated support for modeling structure and behavior in UML has led to the development of alternative modeling approaches that combine structure and behavior into a single model. One such approach is the OPM (Object Paradigm Architecture), which provides a modeling framework that integrates structure and behavior in a single, unified model.

OPM introduces a new paradigm for modeling that is based on the concept of OPM objects. These objects are instances of OPM classes that encapsulate both structure and behavior. This enables the unification of structure and behavior in a single graphical and natural language-based model.

The tight interdependence of structure (what the system is) and behavior (how the system changes) mandates that these two major system aspects be addressed concurrently. This can be difficult if structure and behavior are forced into two (or more) separate diagram types. Advocates of multi-diagram approaches argue for the separation of concerns, or segregation of the various system aspects, including structure, behavior, state transition, and hardware implementation. While a valid approach, this separation of concerns can lead to the model-multiplicity problem, where a system's structure and behavior are represented in multiple models, each with its own symbol set and syntax.

In conclusion, the development of UML and other modeling languages has been instrumental in enabling the design and implementation of complex software systems. However, the lack of integrated support for modeling structure and behavior in a single diagram has led to challenges in maintaining consistency and reducing the cognitive load on developers. The development of alternative modeling approaches, such as OPM, that integrate structure and behavior into a single model offers a promising solution to these challenges.
The Object-Process Methodology may offer a fresh, user-friendly alternative to UML as a generic system architecting framework due to its single model expressed in simple graphics and natural language interpretation.

Systems of interest are inherently complex, so breaking them into different views may seem like a reasonable solution to the complexity management problem, but specifications of complex systems consist of much more than one diagram of each type anyway. Rather than breaking the system's model into various diagram types, OPM copes with complexity through flexible and selective refining/abstracting mechanisms, including in-zooming/out-zooming, unfolding/folding, and state expression/suppression.

These scaling options enable expression of the various system aspects with the same single diagram type, called the Object-Process Diagram (see the figure) while maintaining the readability of each individual diagram. Concurrent modeling of objects, processes, and state transitions can be done naturally in the same diagram at different levels of granularity.

The entire system is thus expressed in a set of diagrams that fully and clearly specify how a system's structure and behavior are related. UML could greatly benefit from adopting this single-model principle. Ideally, UML should have one truly unifying diagram type. If needed, this model, such as the one employed by OPM, can be the source of a number of views.

No less paramount than the symbol- and diagram-explosion problem is UML's modeling of system dynamics. UML's inherent lack of a unifying system dynamics concept calls for another comprehensive revision.

In OPM, Process, the key dynamics modeling concept, is a pattern of transformation experienced...
by one or more objects. A transformation can involve the generation or consumption of an object, or a change of the object’s state. The U2 Partners [9], who at one time contemplated including in their UML2 submission a proposal based on the concept [4], ultimately rejected it earlier this year. Their reason was that, due to “feature creep,” many new features had to be excluded, and the only ones left were those “users would find it surprising if missing.”

The figure outlines in simple graphics and text how processes change object states in a Web service system under development. The OPM diagram is shown at the top of the figure; it combines objects (boxes), processes (ellipses), and transitions of states (rounded-corner rectangles) in the single OPM model, which is concurrently expressed in two modalities: graphics and text. The sentences at the bottom of the figure are a subset of natural language—a formal yet intuitive self-explanatory representation of the system’s structure and behavior readily comprehensible to stakeholders involved in a system’s development.

Simple graphics explained in natural language make the model accessible to non-IT professionals, while the formality of the natural language subset represents a solid foundation for complete code and database schema generation. Either representation can be generated automatically from the other.

A note is also in order regarding the adverse effect of programming languages on UML. UML is intended for use by software professionals to generate software systems. To be accessible to non-IT professionals—the ultimate consumers of the systems and whose approval is required for the UML-based design—UML must avoid programming jargon, such as the following, which appears in the UML1 specification:

“A property is … a text string … The default syntax is:

visibility name : type-expression [ multiplicity ordering ] = initial-value [ property-string ]

Where visibility is one of: 1 public visibility # protected visibility 2 private visibility - package visibility.”

The abundance of such cryptic programming language constructs makes UML-based system specifications comprehensible to only avid IT professionals. How can the UML user community of professional system architects and software engineers expect ongoing collaboration with the prospective users of their systems if they have to master machine-oriented syntax sprinkled across a myriad of symbols and diagrams?

**Conclusion**

For UML to endure and thrive, no less than a revolution is in order, but tool-vendor investment and users’ installed base render it highly unlikely, if not impossible. Offering no remedy, the U2 Partners’ proposed specification [9], which, given the balance of power, is today the only serious alternative, further diverts UML from becoming the small agile language it should be.

However, despite the problems, the momentum UML has achieved may yet sustain its survival for years to come. But they are so profound and the readiness to solve them so limited that eventually the language risks collapsing under its own weight, ceasing to be the lingua franca of software systems modeling. Perhaps this is all for the best, since rendering UML irrelevant will enable its user community, including system architects, software engineers, and program developers, to seek and begin to embrace a fresh systems modeling paradigm.

---

The following references are cited in the text:

7. Simons, T. Dependencies and associations. Precise UML Group Email Forum (June 21, 2001); see www.cs.york.ac.uk/puml/puml-list-archive/0310.html.
9. U2 Partners. UML2 Draft Version 0.671 (Jan. 30, 2002); see www.u2-partners.org/.

Dov Dori (dori@ie.technion.ac.il, dori@mit.edu) is an associate professor of information systems engineering at the Faculty of Industrial Engineering and Management, Technion, Israel Institute of Technology, Haifa, Israel, and a research affiliate at MIT, Cambridge, MA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

© 2002 ACM 0002-0782/02/1100 $5.00