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, consume, or affect objects by changing their state. This simple yet powerful paradigm—the foundation of OPM—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 task is, however, counter-intuitive and extremely difficult if structure and behavior are forced into two (let alone nine) separate diagram types. Advocates of multi-diagram approaches argue for the "separation of concerns," or segregation into distinct models of the various system aspects, including structure, behavior, state transition, and hardware implementation. While a valid consideration, separating a system's structure from its behavior puts a major obstacle before systems developers, as it results in the model-multiplicity problem. The need to construct, maintain, and consult multiple models, each with its own symbol set, severely impedes architect productivity. It also impedes the ability of designers, already grappling with systems that are inherently complex, seeking to concurrently model structure and behavior.

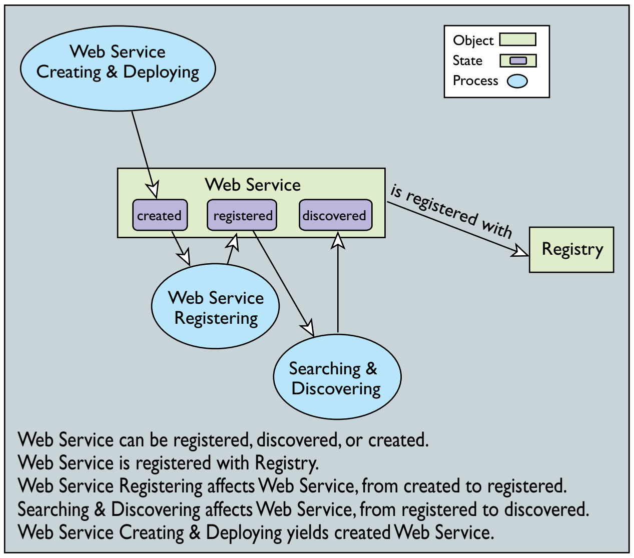
Trying to comprehend or design a complex system, any human's natural thought process focuses on the interaction between the system's structure and its behavior. As soon as the existence of an object, or component or subsystem (all objects of increasing complexity), is identified or conceived (the structural aspect), the question arises as to what processes transform (generate, consume, or affect the state of) the object (the behavioral aspect). Hence, the human thought process in system architecture, analysis, and design involves the constant interplay between a system's structure and its behavior. However, rather than supporting this thought process, UML's separation of the system model into different views, represented by different diagram types, dictates and enforces the damaging segregation of structure and behavior, thereby obscuring the developer's overall system comprehension.

Lack of UML support for integrating structure and behavior in a single model puts the intellectually demanding burden of flipping back and forth between at least two diagram types entirely on the developer's shoulders. That is, UML's aspect segregation unnecessarily strains developers' cognitive abilities by requiring they mentally integrate the various system models in a coherent holistic view without providing a proper integration framework for doing so. A complete mental picture of the modeled system must therefore be reconstructed from knowledge scattered across multiple diagrams with diverse graphical syntax and semantics.

Integrity Maintenance

Model multiplicity mandates integrity maintenance among a system's various models, increasing exponentially with the number of diagrams. The recursive ripple effect of changing any model, thus triggering the potential need to modify other diagrams, renders intractable the problem of keeping coherent all system views. Concurrent development of several sets of diagram types requires constant switching from one convention set to another. This is not merely a technical problem but a human factors problem; the solution is simplification. Hence, even with superb CASE tools, keeping the diagrams synchronized and preventing the introduction of contradictions and mismatches in the overall system model (which, in UML, exists only in the modeler's mind) become daunting tasks beyond anyone's cognitive ability.

This cognitive load severely hinders use of UML as a system-modeling tool. Indeed, my discussions with software developers have caused me to realize that many, if not most, of them struggle with UML's sheer complexity and inconsistencies, especially regarding the modeling of system dynamics in OO settings. Disliking it, they use it mainly because their organi-



Web Service can be registered, discovered, or created.
Web Service is registered with Registry.
Web Service Registering affects Web Service, from created to registered.
Searching & Discovering affects Web Service, from registered to discovered.
Web Service Creating & Deploying yields created Web Service.

A top-level Object-Process Diagram of a Web service life cycle and its equivalent, automatically generated Object-Process Language paragraph.

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.

zations follow the standard.

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 all 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:

"An attribute 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. C

References

1. Dori, D. *Object-Process Methodology: A Holistic Systems Paradigm.* Springer Verlag, Berlin, 2002.
2. Dori, D., Reinhartz-Berger, I., and Sturm, A. *OPCAT Object-Process CASE Tool, Version 2.02,* 2002; see iew3.technion.ac.il/~dori/opcat/index-continue.html.
3. Heylighen, F. *Principia Cybernetica Web.* Vrije Universiteit Brussel, Brussels, Belgium, 2001; see pespmc1.vub.ac.be/HEYL.html.
4. Object Management Group. *Sight Code Initial Submission Against the UML 2.0 Infrastructure RFP,* OMG Document ad/01-08-23, 2001; see www.omg.org/cgi-bin/doc?ad/2001-08-23.
5. Peleg, M. and Dori, D. The model multiplicity problem: Experimenting with real-time specification methods. *IEEE Transact. Soft. Engin. 26,* 8 (2000), 742–759.
6. Siau, K. and Cau, C. Unified Modeling Language: A complexity analysis. *J. Database Mgmt. 12,* 1 (Jan.–Mar. 2001), 26–34.
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.
8. Soffer, P., Golany, B., Dori, D., and Wand, Y. Modeling off-the-shelf information systems requirements: An ontological approach. *Requir. Engin. 6,* 3 (2001), 183–199.
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.