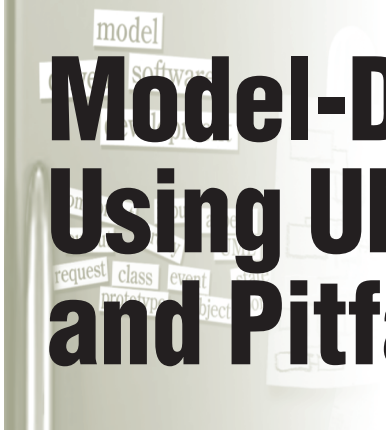

Model-Driven Development Using UML 2.0: Promises and Pitfalls

Robert B. France, Sudipto Ghosh, and Trung Dinh-Trong
Colorado State University

Arnor Solberg
SINTEF

The Object Management Group initiated the Unified Modeling Language 2.0 effort to address significant problems in earlier versions. While UML 2.0 improves over earlier versions in some aspects, its size and complexity can present a problem to users, tool developers, and OMG working groups charged with evolving the standard.

Experience indicates that effective complexity management mechanisms automate mundane development tasks and provide strong support for separation of concerns. For example, current high-level programming languages and integrated development environments provide abstractions that shield developers from intricate lower-level details and offer automated support for transforming abstract representations of source code into faithful machine-executable forms.

Advances in software development and information processing technologies have resulted in attempts to build more complex software systems. These systems have highlighted the inadequacies of the abstractions provided by modern high-level programming languages. This has led to a demand for languages, methods, and technologies that raise the abstraction level at which software systems are conceived, built, and evolved. The Object Management Group (OMG) has responded to this demand with the UML version 2.0[1] and the Model Driven Architecture (MDA) initiative (www.omg.org/mda).

The problems initially targeted by the architects of the UML 2.0 standard included the apparent bloat in earlier UML versions and the lack of well-defined semantics. During the standard's development, the requirements evolved to include support for model-driven development (MDD), particularly the MDA approach to MDD.

Widespread awareness of the problems associated with earlier UML versions, coupled with growing interest in MDD, raised expectations that the UML 2.0 architects would produce a version that consisted of a significantly reduced set of modeling concepts for concisely and conveniently describing a wide variety of applications. It was also anticipated that this version would have precise semantics that would facilitate the automation needed to move MDD beyond the realm of existing computer-aided software engineering tools. Some expected UML 2.0 architects to produce a modeling language silver bullet, or at least a close approximation.

Those unfamiliar with the inner workings of a community-driven language standardization effort found the size and complexity of the UML 2.0 standard surprising. Indeed, the end result seems at odds with the forces that initially motivated this major revision. From a detractor's viewpoint, the numerous modeling concepts, poorly defined semantics, and lightweight extension mechanisms that UML provides make learning and applying it in an MDD environment difficult.

These real problems must be addressed, but we should not be surprised that this first-generation modeling language is far from perfect. As some UML 2.0 architects point out, we are still in the "infancy of modeling language design."[2] We need constructive criticism of UML to drive the process of evolving MDD-related knowl-

0018-9162/06/$20.00 © 2006 IEEE

edge. In this sense, UML 2.0 can play an important role as an explicit form of modeling experience that stakeholders can dissect, analyze, and critique.

Advocates point out that UML 2.0 reflects best modeling practices and experience. They tout the following major improvements:

- Better support for the notion of UML as a *family of languages* through the use of profiles and semantic variation points that mark the parts of UML intentionally left without semantics to accommodate user-defined ones.
- Improved modeling expressiveness, including improved modeling of business processes, support for modeling reusable classifiers, and support for modeling the architectures of distributed, heterogeneous systems.
- Integration of the Action Semantics that developers can use to define the models' runtime semantics and provide the semantic precision required to analyze models and translate them into implementations.

UML 2.0 can play an important role as an explicit form of modeling experience.

Overzealous promotion of UML and associated MDD technologies can be as harmful as unfair criticisms. Such promotion can raise user expectations to a currently unattainable level. By cutting through the hype surrounding UML 2.0, we hope to provide some insights into how well it can support MDD.

UML 2.0

The UML 2.0 standard contains a large set of modeling concepts that are related in complex ways. In defending the size and complexity of UML 2.0, its architects point out that the language is intended to support modeling in a variety of domains. To cope with this complexity, designers organized the standard into four parts:

- *Infrastructure*—defines base classes that provide the foundation for UML modeling constructs;
- *Superstructure*—defines the concepts that developers use to build UML models;
- *Object constraint language*—defines the language used for specifying queries, invariants, and operation specifications in UML models; and
- *Diagram interchange*—defines an extension to the UML metamodel that supports storage and exchange of information pertaining to the layout of UML models.

The UML 2.0 Superstructure describes the standard's externally visible parts—the concepts used to describe systems. To manage complexity, the Superstructure concepts are organized into language units. A language unit is used to model systems from a particular viewpoint. For example, the Interactions language unit is used to model interactions among behavioral elements, while the Activities language unit is used to describe application workflows.

In addition, some of the more complex language units are organized into increments, with one increment adding detail to a smaller increment. For example, the Activities language unit consists of the following activity increments: Fundamental, Structured, and Complete Structured Activities. Structuring UML 2.0 into relatively independent language units lets users pick only the appropriate parts to learn and understand.

The relative novelty of UML modeling can require users to understand most if not the entire language before determining what models are appropriate for their environments. It is a mistake, however, to view the standard as a textbook description of UML. It specifies the concepts independently of the methods and technologies that will use the language. Tool developers, methodologists, and UML textbook writers are the standard's intended readers. We know of no books that comprehensively cover UML 2.0, but expect these will appear soon now that the Superstructure standard has been finalized.

UML 2.0 VIEWPOINTS

UML provides good support for separating concerns by letting users model systems from four viewpoints:

- Models produced from the *static structural* viewpoint describe the system's structural aspects. Class models are examples of descriptions produced using this viewpoint.
- Developers use the *interaction* viewpoint to produce sequence and communication models that describe the interactions among a set of collaborating instances.
- The *activity* viewpoint is used to create models that describe the flow of activities within a system.
- The *state* viewpoint is used to create state machines that describe behavior in terms of transitions among states.

These viewpoints are not completely orthogonal: Concepts used in one viewpoint often depend on concepts used in another. For example, participants in an interaction must have their classifiers defined in a static structural model. Such dependencies are specified in the UML metamodel, and tools can use them to determine information consistency across system views.

The UML 2.0 metamodel's size and complexity pose a potential pitfall in this respect. Tool developers and methodologists face a difficult challenge when trying to

identify all the specified dependencies among concepts. Further, determining whether the metamodel captures all required dependencies is difficult. Tools that query and navigate the metamodel will help developers use it effectively.

Developers of complex systems might want the ability to model systems from user-defined viewpoints. UML 2.0 currently does not provide mechanisms for creating models using such viewpoints other than those defined in the language. Work on UML-based *aspect-oriented modeling* (AOM) attempts to provide some support for describing systems from user-defined perspectives.[3] An aspect model describes a system from a user-defined viewpoint. The model is a slice of a UML system model that contains only information pertinent to the viewpoint.

UML 2.0 makes the user responsible for defining and communicating appropriate semantics plugged into variation points.

Developers use AOM techniques to produce a design model that consists of aspect models and a base model. Aspect models are composed with the base model to produce an integrated view of the design. AOM requires tools for creating multiple, possibly overlapping diagrams of a particular type—for example, class diagrams—and for composing diagrams of the same type. Commercial versions of these tools currently do not exist.

TAILORING UML 2.0

UML 2.0's designers use semantic variation points to acknowledge that a variety of useful semantics can be associated with some UML concepts. For example, semantic variation points are used to tackle the poorly defined notion of aggregation found in earlier UML versions. Papers critiquing earlier versions of the aggregation concept[4,5] identify a variety of semantics that can be associated with the relationships between the whole and parts in an aggregation structure, particularly relationships in weak aggregation structures. UML 2.0 counters this problem by not associating specific semantics to weak aggregation. It also avoids specifying how parts of an aggregate are created.

Examples of other semantic variation points in UML 2.0 include the following:

- semantics associated with the reception of events in state machines,
- the means by which the system delivers messages to recipients during an interaction, and
- how use case extension points are defined.

Semantic variation points support the notion of UML as a family of languages. UML 2.0 makes the user responsible for defining and communicating appropriate semantics plugged into variation points. It does not provide default semantics or a list of possible variations, nor does it formally constrain the semantics that can be plugged into variation points. This can lead to pitfalls such as users assigning semantics inconsistent with the semantics of related concepts or failing to communicate particular semantics to model readers and tools that analyze models. This can lead to misinterpretation and improper model analysis. Users must have knowledge of possible variations that can be plugged in to tailor the UML 2.0 semantics appropriately.

Semantic variation points support a form of *tailoring-in-the-small*. In contrast, profiles support *tailoring-in-the-large*. A UML profile describes how UML model elements are extended to support usage in a particular modeling context. For example, a profile can be used to define a UML variant suited for modeling Enterprise JavaBeans applications.

In a profile, UML model elements are extended using stereotypes that define additional element properties. The properties defined in an extension introduced by a stereotype must not contradict the properties associated with the model element. A profile can introduce new constraints and define additional attributes in classes, but it cannot introduce new classes of model elements or remove existing classes of elements. For this reason, profiles are considered lightweight extension mechanisms. Profile examples include the OMG's UML Profile for Schedulability, Performance, and Time, and the SysML profile for system modeling (www.uml.org/#UMLProfiles).

Unfortunately, the UML 2.0 profile mechanism does not provide a means for precisely defining semantics associated with extensions. For this reason, developers cannot use profiles in their current form to develop domain-specific UML variants that support the formal model manipulations required in an MDD environment.

When UML does not provide the base elements for modeling domain-specific concepts, developers can use extension mechanisms to change the UML metamodel. These heavyweight mechanisms use the Meta-Object Facility (MOF) to define the metamodel for the UML variant.

Several tool vendors are developing technologies that support the development of domain-specific modeling languages through heavyweight extensions. Language engineers will use these technologies to develop MDD environments that consist of domain-specific modeling languages and supporting application development tools. The application of these technologies will place the burden of language development on users.

STRUCTURAL MODELING CONCEPTS

Class modeling concepts are the most widely used UML concepts. Some development projects create only class models. UML 2.0 provides some minor but notable changes to class modeling concepts. One improvement in

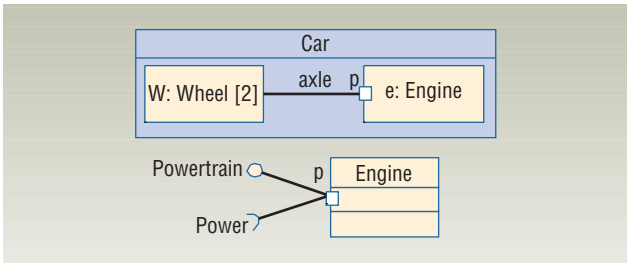

***Figure 1. UML 2.0 example of a structured class, taken from the UML 2.0 Superstructure specification. The class* Car *consists of two parts:* W: Wheel *and* e: Engine. *The part* W *consists of two wheels and* e *consists of an engine, an instance of the structured class* Engine*, which has a port named* p *that implements an interface named* Powertrain.**

the class diagram notation is the introduction of a new association navigability marker. The new notation lets developers distinguish when a navigation is explicitly prohibited from when a design decision has not been made to allow or disallow navigation. An X at an association end indicates that navigation via that association end is prohibited; lack of an adornment—an arrowhead or an X—indicates that no decision has yet been made with respect to navigation at the association end.

To better support software architecture modeling and to support reuse of classifiers in multiple environments, UML 2.0 introduces the *structured classifier* and *port* concepts. A structured classifier has an internal structure described by a structure of parts. A part specifies a collection of instances. For example, a structured class has an internal structure that is a configuration of objects. A composite structure diagram describes a structured classifier's internal structure.

A structured classifier can be associated with ports representing points of interactions with the classifier. A port can be associated with *provided* and *required* interfaces that specify interactions supported by the port. Provided interfaces specify the incoming communications that the classifier or its constituent parts handle, while the required interfaces specify the communications the classifier sends out. The provided interfaces thus determine the services the classifier provides through a port while the required interfaces determine the services the classifier expects the environment to provide.

If an instance of a classifier handles incoming communications through a port directly, the port is referred to as a *behavior port*. Alternatively, the constituent parts of the instance can handle incoming communications.

When the system creates an instance of a classifier with ports, it also creates instances of the ports. A *port instance*, also called an *interaction point*, creates a realization of its provided interfaces, which can constrain the interactions that occur across the port. For example, a realization can require that the interactions adhere to a protocol.

Figure 1 shows an example of structured classes. The structured class *Car* consists of two parts: *W:Wheel* and *e:Engine*. The part *W* consists of two wheels, while *e* consists of an engine that is an instance of the structured class *Engine*. The *axle* connector links these parts. The class *Engine* has a port named *p* that implements a provided interface named *Powertrain*, and which requires operations described in the required interface named *power*.

Ports let developers construct structured classifiers independently of the environments in which they will be used. The environments need only know and conform to the interaction constraints the ports impose. Detailed knowledge about the internal structure of these classifiers is not needed to use them.

When using classifiers with ports, developers will encounter a lack of guidance on how to map these classifiers to implementations. The major programming languages do not provide direct support for implementing ports. Developers thus must transform models with ports to equivalent models without ports before implementing their models. For example, one approach treats ports as classes with a strong aggregation relationship with their classifiers.

BEHAVIORAL MODELING CONCEPTS

A good MDD language should have precise operational semantics. This eases the transformation of models to implementations and enables the development of technologies for animating and testing models.

The runtime semantics informally described in the UML 2.0 standard are intended to facilitate the development of tools that execute UML models. The semantics are based on the assumption that the root cause of behavior in a model is an action executed by an active object, where an active object is one with its own thread of control. Actions form the basic behavior units that UML behavioral diagrams describe. These actions always execute in the context of some object. For example, method bodies can be described as a sequence of actions using an activity model.

To define an executable UML model, the language must describe actions precisely. UML 2.0 accomplishes this by incorporating concepts defined by the *Action Semantics*, which identifies actions and describes their effects. For example, it defines actions for creating and deleting objects and links and for getting and setting attribute values. UML 2.0 Action Semantics does not specify a concrete syntax for actions. Researchers and methodologists have defined different surface action languages, including the action semantics language (ASL; www.kc.com/download/index.php) and Java-Like Action Language (www.cs.colostate.edu/~trungdt/uml_testing/tool).

The Action Semantics does not significantly raise the level of abstraction above that provided by programming languages. Writing a method body using an action

language that does not provide a level of abstraction above primitive actions can require as much effort as writing the method body in some programming language. Defining action languages that use the primitive actions to build higher-level constructs that significantly raise the abstraction level for describing behavior will significantly enhance UML 2.0's ability to support MDD.

Sequence and activity diagrams have been significantly changed in UML 2.0. Sequence diagram changes were driven by the significant experience gained developing the Message Sequence Diagram (MSD) standard language. MSDs have been used for decades in the telecommunications industry. The standard has evolved to the point that it has formalized and documented semantics. Notable improvements to sequence diagrams include the ability to

- name and refer to fragments of interactions, and
- decompose lifelines of participants with internal structure into interactions that describe how their internal parts interact in the context of the sequence diagram.

UML 2.0 also introduces a new diagram type, the Interaction Overview Diagram, which shows the flow relationships among fragments and sequence diagrams.

Despite these improvements, it is still difficult to describe some scenarios using sequence diagrams. In particular, modeling scenarios that involve iterative or recursive interactions on collections of participants is problematic because lifelines are static: If a lifeline appears in a sequence diagram loop fragment, it refers to the same participant across the iterations. If the system accesses a different participant in the collection in each iteration, the participants must be explicitly represented as separate lifelines.

Figure 2 describes a scenario in which an *Engine* object sends *rotate()* messages to a collection consisting of two *Wheels*. In this case, the problem is not readily apparent because there are only two wheels. The problem becomes apparent in the cases where the number of wheels in the collection is large or can vary. Some users get around this problem by using the informally defined dynamic lifelines shown in Figure 3: *w[i]* is a dynamic lifeline set to *w1* in the first iteration, then to *w2* in the second iteration. This improvised notation is more concise and intuitive, but often the semantics associated with the dynamic lifelines are imprecisely defined.

Sequence diagrams describe only the interactions between instances. Developers can use state machine models and activity diagrams to describe how participants in an interaction respond to messages sent to them. State machines in UML 2.0 resemble state machines in the previous version and have been used as the basis for executable variants of UML.[6] Using them to completely describe the behavior of reactive systems is appropriate,

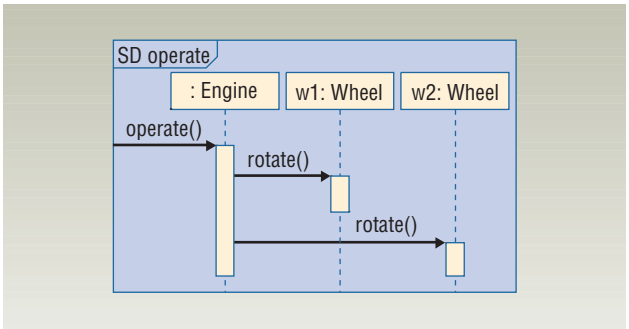

***Figure 2. A loop modeled using UML 2.0 notation. Here, an* Engine *object broadcasts a* rotate() *message to a collection consisting of two* Wheels.**

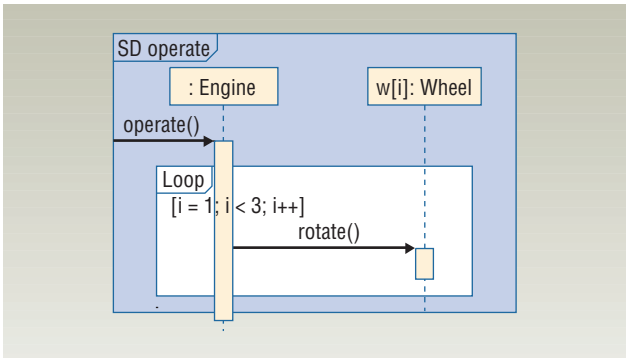

Figure 3. A loop modeled using a dynamic lifeline. Here,* w[i] *is a dynamic lifeline set to* w1 *in the first iteration, then to* w2 *in the second iteration.

but describing the behavior of information systems and other system types using only states and transitions can contribute to accidental complexity.

Activity diagrams provide a more appropriate vehicle for defining behaviors in information systems. Previous UML versions treated activity diagrams as a specialized form of state machines. Activity diagrams now have a Petri-net-like flow semantics to better support modeling of activity flows.

An activity diagram is a graph of nodes, where a node represents an action, data store, or control flow element. Tokens flowing across edges that connect actions determine when the actions are executed. Rules that determine how tokens flow in an activity diagram have been defined in the standard. Unlike previous versions of activity diagrams, subflows can be stopped without stopping other flows. This lets developers model activity flows that could not be modeled previously. More importantly, activity diagrams can now be used to describe method bodies procedurally. This paves the way for developing tools that animate and test models.

TESTING UML DESIGNS

In MDD, models provide the basis for understanding system behavior and generating implementations. The

ability to animate models can help developers better understand modeled behavior, while testing models can help uncover errors before developers transform the models into implementations.

Both novices and experienced developers will benefit from the visualization of modeled behavior that model animators provide. Model animation can give quick visual feedback to novice modelers, helping them identify improper use of modeling constructs. Experienced modelers can use model animation to better understand the designs that other developers create.

Models must be validated so that faults do not pass downstream to implementations. Currently, developers evaluate UML design models using walk-throughs, inspections, and other informal types of largely manual design-review techniques. Manual review of large UML designs can be tedious and error-prone. Existing formal verification techniques can be applied, but few of them scale to large system models. The runtime semantics associated with UML 2.0 models pave the way for the development of systematic model testing techniques that exercise executable models using test inputs.[7,8]

The UML 2.0 specification does not present a convenient overview of interactions models.

We have developed a prototype tool called the UML Animator and Tester UMLAnT,[9] which uses information from UML class, sequence, and activity models to animate and test UML design models. During testing, the UMLAnT prototype displays sequence diagrams that capture messages exchanged between objects. It also displays object diagrams that describe the configurations created as tests are executed.

Developers can use UMLAnT to generate test inputs from a UML design's class and sequence models. A set of test criteria defines test objectives and is used by the tool to assess the adequacy of test input sets. The tool generates an executable form of the design model from the class and activity models. A model is tested by applying test inputs to the executable form. The tests include several checks to determine test failures. For example, UMLAnT can determine when object configurations produced during execution violate constraints stated in a class model and when operation postconditions remain unsatisfied after the operations are executed. UMLAnT's animation feature can be used to debug faulty models. Object state and message sequence information produced during animation can help in locating faults in models.

THE UML "METAMUDDLE"

Developers of model transformations and other MDD mechanisms that manipulate models often work at the metamodel level. For example, developers frequently express model transformations as mappings between metamodel elements. Further, MDD technologies sometimes extend metamodels and define new language constructs to better support domain-specific modeling and modeling of product families. Developers of UML-based MDD technologies must therefore have a good understanding of the UML metamodel. However, the complexity of the current UML 2.0 metamodel can make understanding, using, extending, and evolving the metamodel difficult.

In practice, developers use only a small subset of the diagram types that UML provides. Not all the concepts available in a diagram type are used for modeling an application or product family. Thus, developers seldom need to have full knowledge of the UML metamodel to specify model transformations.

Unlikely as it seems, this raises a problem: The task of identifying and extracting the required subset of concepts from the UML metamodel must currently be performed by manually navigating through the "metamuddle." To illustrate this, consider the task of extracting a simple view of the UML 2.0 metamodel that focuses only on the relationships among concepts used to create basic sequence models. An examination of the UML metamodel reveals that the required information is scattered across the metamodel.

The Interactions language unit is organized into a number of related packages. For example, the Basic Interaction package has direct dependencies on the Basic Behaviors, Basic Actions, and Internal Structures. These packages have further dependencies on other packages described elsewhere in the 1,000-plus-page Superstructure specification. Tracing dependencies through these packages to extract the metamodel substructure that describes basic sequence models is tedious.

The UML 2.0 specification does not present a convenient overview of interaction models. For example, we would expect that the relationship between message ends and lifelines would be easy to identify. Surprisingly, deriving this simple relationship requires navigating through several associations and inheritance hierarchies. Figure 4 shows a snapshot of the work involved in finding the simple interaction metamodel.

The shaded boxes indicate the interaction concepts of interest. The other concepts must be navigated through to derive the required relationships. In some cases, finding the derived relationships requires moving through inheritance hierarchies that span packages. For example, lifeline inherits from *NamedElement*, which is defined in another package described elsewhere in the Superstructure document.

USING THE UML METAMODEL

The problem of extracting a metamodel view of basic sequence models gives some insight into why the task of defining transformations using the UML metamodel is currently tedious and error-prone. These problems can be alle-

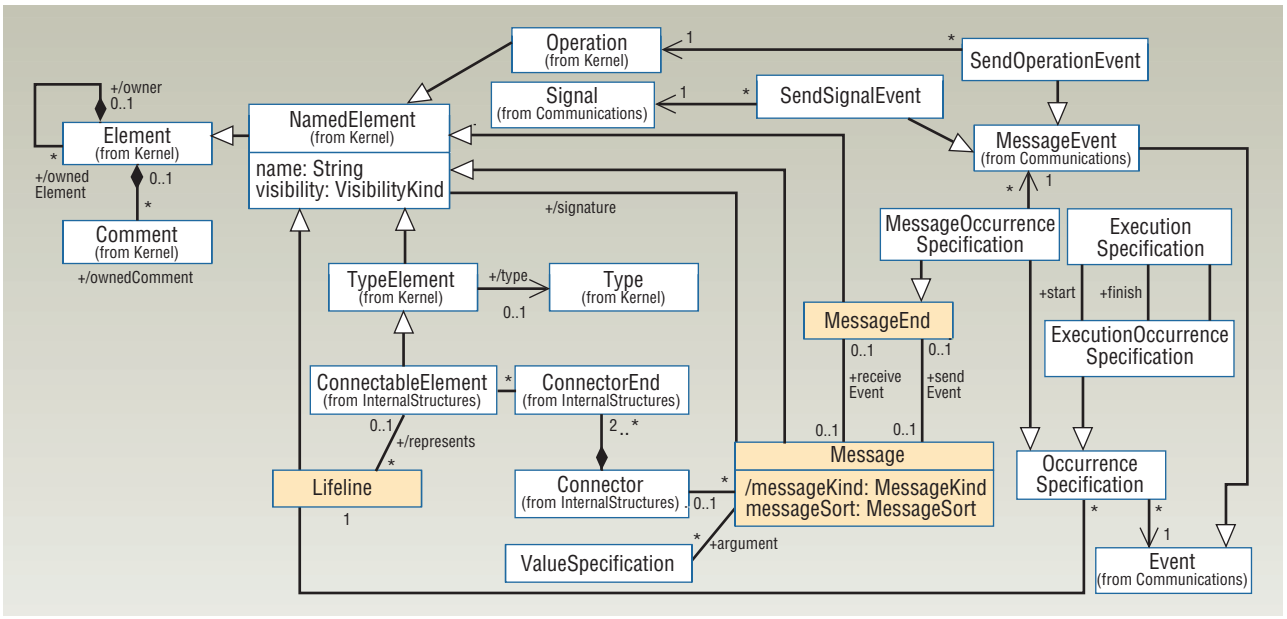

Figure 4. Finding the simple UML interactions metamodel. The shaded boxes indicate the interaction concepts of interest. Users must navigate through the other concepts to derive the required relationships.

viated by providing simplified metamodel views for each UML 2.0 model type. These views should describe only the concepts and relationships that appear in the models.

A more flexible and useful solution would provide tools that developers can use to query the metamodel and extract specified views from it. Query/extraction tools should be capable of extracting simple derived relationships between concepts and more complex views that consist of derived relationships among many concepts. Metamodel users can apply such tools to better understand the UML metamodel and to obtain views for specifying patterns and transformations. UML architects can use these tools to help determine the impact of changes to the UML and to check that the changes are made consistently across the UML metamodel.

Such tools can be developed using existing technology. UML tools that have basic facilities for accessing internal representations of the UML metamodel can be extended with query and extraction capabilities. Metamodeling tools, such as those developed by Xactium (www.xactium.com) and Adaptive Software (www.adaptive.com), and the megamodeling tools advocated by Jean Bézivin (www.sciences.univ-nantes.fr/lina/atl/contrib/bezivin),[10] have some of these capabilities. Another tool that can ease the task of using the UML metamodel is one that takes a UML model and produces a metamodel view that describes its structure. Developers can use such a tool to support compliance checking of models manipulated by transformations. For example, they can use the tool to check that source, target, and transformation models conform to their respective metamodels.

Evolving the complex UML 2.0 metamodel will be a challenge. It is expected that UML will change, and thus the standard's maintainers will be faced with assessing the impact of suggested changes, making changes consistently across the metamodel, and verifying the consistency and soundness of the changed metamodel. To better manage the evolution of the UML metamodel, we propose using AOM techniques to organize the metamodel into aspect models. For example, the following aspects can be defined:

- views for each of the diagram types in UML 2.0 that contain only the concepts visualized in diagrams; and
- views of abstract concepts reflecting language and UML-specific concerns such as namespace management, element typing, element connectivity, and execution semantics.

Each aspect model presents an uncluttered view of how the UML metamodel treats a concern. The aspect models need to be composed to produce an integrated view of the metamodel. Aspect treatment of the metamodel lets a developer make changes in a single aspect or create a new aspect, and then compose it with other aspects to determine the impact of the change on the metamodel. Using a composition mechanism also helps ensure that changes are consistently made across the metamodel. A prototype composition mechanism for UML class diagrams that can be used for this purpose has been developed.[11]

The purely structural nature of the UML 2.0 metamodel can hinder attempts at providing practical support for manipulating models. Specifying behavior at the metalevel can ease model manipulation tasks such as model transformation and model composition.

Operations in metamodels can describe and constrain how models are manipulated in MDD environments. Also, adding metaoperations for transforming related modeling concepts among different UML diagram types can ease consistency checks across different views and better support transformation between model types. In addition, metaoperations can be used to define standardized transformations that are based on relationships defined in the UML metamodel. For example, such metaoperations can be used to transform interaction models to class models based on the relationships between the *Lifeline* concept and the *Class* concept.

For UML 2.0 to support MDD, a framework that includes extensive facilities for tailoring the language and for manipulating models is needed. Many commercial UML tools claim to support MDD. The better ones tend to have limited support for defining and using UML model transformations and tend to restrict their users to a particular implementation platform. This is probably the best that can be done now.

UML 2.0's size and complexity present a problem not only to UML-based MDD tool developers but also to OMG working groups charged with evolving the standard. It will be extremely difficult to evolve UML 2.0 using only manual techniques. Evolving the standard can involve making changes to concepts that are scattered across the metamodel, verifying that the required changes are incorporated consistently across the metamodel, determining the impact that a change will have on other metamodel elements, and ensuring that the changes do not result in a metamodel that defines inconsistent or nonsensical language constructs. The metamodel needs to be restructured to ease evolution, and tools should be developed to help navigate the metamodel.

Evolving UML so that it better supports MDD will require at least addressing the problems we have identified and developing technologies that support practical development and use of profiles and use of the metamodel for managing and manipulating models.

UML's role in enabling MDD should not be undervalued. Ongoing UML-related research in academia and industry has added and will continue to add to the MDD knowledge base. We expect that UML will be the genesis of future MDD languages. ■

References

1. The Object Management Group, *Unified Modeling Language: Superstructure, Version 2.0*, OMG document formal/05-07-04, 2004.
2. B. Henderson-Sellers et al., "UML—The Good, the Bad or the Ugly? Perspectives from a Panel of Experts," *Software and System Modeling*, Feb. 2005, pp. 4-13.
3. R.B. France et al., "An Aspect-Oriented Approach to Design Modeling," *IEE Proc. Software,* special issue on Early Aspects: Aspect-Oriented Requirements Eng. and Architecture Design, Aug. 2004, pp. 173-185.
4. B. Henderson-Sellers and F. Barbier, "Black and White Diamonds," *Proc. UML99*, LNCS 1723, Springer-Verlag, 1999, pp. 530-565.
5. M. Saksena, R. France, and M. Larrondo-Petrie, "A Characterization of Aggregation," *Int'l J. Computer Systems Science & Eng.*, vol. 14, no. 6, 1999, pp. 363-371.
6. S. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley, 2002.
7. A. Andrews et al., "Test Adequacy Criteria for UML Design Models," *J. Software Testing, Verification and Reliability*, vol. 13, no. 2, 2003, pp. 95-127.
8. T. Dinh-Trong et al., "A Tool-Supported Approach to Testing UML Design Models," *Proc. 10th IEEE Int'l Conf. Eng. Complex Computer Systems* (ICECC05), IEEE Press, 2005, pp. 519-528.
9. T. Dinh-Trong et al., "UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs," to appear in *Proc. Eclipse Technology eXchange Workshop, Conf. Object-Oriented Programming, Systems, Languages and Applications* (OOPSLA), ACM Press, 2006.
10. J. Bézivin, F. Jouault, and P. Valduriez, "On the Need for Megamodels"; www.sciences.univ-nantes.fr/lina/atl/www/papers/OOPSLA04/bezivin-megamodel.pdf
11. Y.R. Reddy et al., "Directives for Composing Aspect-Oriented Design Class Models," to appear in *Trans. Aspect-Oriented Software Development*, 2006.

Robert B. France *is a professor in the Computer Science Department at Colorado State University. His research interests include model-driven development, aspect-oriented development, and formal methods. He received a PhD in computer science from Massey University, New Zealand. Contact him at france@cs.colostate.edu.*

Sudipto Ghosh *is an assistant professor in the Computer Science Department at Colorado State University. His research interests include software testing, aspect-oriented development, and component-based development. He received a PhD in computer science from Purdue University. Contact him at ghosh@cs.colostate.edu.*

Trung Dinh-Trong *is a doctoral student at Colorado State University. His research interests include software testing, software processes, and software modeling. He received an MS in computer science from Colorado State University. Contact him at trungdt@cs.colostate.edu.*

Arnor Solberg *is a research scientist at SINTEF. His research interests include model-driven development, quality of service, and aspect-oriented development. He is currently pursuing a PhD in computer science at the University of Oslo, Norway. Contact him at arnor.solberg@sintef.no.*