UML 2: A model-driven development tool

B. Selic

The Unified Modeling Language® (UML®) industry standard has recently undergone a major upgrade, resulting in a revision called UML 2. The primary motivation for this revision was to make UML better suited to model-driven development™ (MDD™), an approach to software development in which software models play a vital role. This requires a modeling language that is not only highly expressive but also capable of specifying models that are precise and unambiguous. In this overview article, we describe the key developments in UML 2 and the rationale behind them, and we explain how they help meet the needs of MDD. These new capabilities can be grouped into two distinct categories: (1) internal and architectural changes required to support MDD and (2) new modeling features. This paper is a revised version of a Web article, "Unified Modeling Language Version 2.0," which was published on March 21, 2005, by developerWorks®, IBM Corporation.

INTRODUCTION

The early part of the 1990s saw heightened interest in the object paradigm and related technologies. New programming languages based on this paradigm, such as Smalltalk, Eiffel, C++, and Java**, were defined and widely used. These were then accompanied by a prodigious and confusing glut of object-oriented software design methods and modeling notations. For example, in his very thorough overview of object-oriented (OO) analysis and design methods (covering over 800 pages), Ian Graham lists over 50 seminal OO methods.[1] Given that the object paradigm consists of a relatively compact set of core concepts, such as encapsulation, inheritance, and polymorphism, there was clearly a great deal of overlap and conceptual alignment across these methods, much of it obscured by notational and other differences of little or no consequence. This caused much confusion and needless fragmentation, which, in turn, impeded adoption of this extremely useful paradigm. Developers were forced to make difficult and binding choices between mutually incompatible languages, tools, methods, and vendors. The fragmentation also made it very difficult to find experts who were sufficiently fluent in the language chosen, leading to additional training costs.

For this reason, when the Unified Modeling Language** (UML**) initiative was announced by

Rational* Software, the reaction by the software development community was overwhelmingly positive. UML started as an amalgamation of the two most popular OO methods of the time: the OMT method of Rumbaugh et al.[2] and the Booch method.[3] The primary authors of these two methods, Jim Rumbaugh and Grady Booch, were later joined by Ivar Jacobson, whose OOSE method was noted for its seminal contribution to requirements-driven software construction processes, based on the now familiar notion of use cases.[4] Following this initial effort, which provided a homogenous conceptual and notational framework, a number of leading methodologists and thought leaders were invited to critique and to contribute to UML. Particularly notable was the contribution of David Harel, whose statechart formalism[5] was adapted and then adopted as one of the core elements of the language.

The intent behind UML was not invention but consolidation. The result was a synergistic blending of the best features of the various OO languages, methods, and notations into a single vendor-independent modeling language and notation. This open quality is one of the main reasons why UML very quickly became a *de facto* standard and, following its adoption by the Object Management Group (OMG**) in 1996, a bona fide industry standard.[6–8]

Since then, UML has been widely adopted and is supported by the majority of major modeling tool vendors. It has also been incorporated as an essential part of the computer science and engineering curricula in universities throughout the world and in various professional training programs. Last but not least, it is being used extensively by academic and research communities as a convenient lingua franca and is itself the subject of significant research.

UML has also helped raise general awareness of the value of software modeling as a means for coping with the complexity of modern software. Although this highly useful technique is almost as old as software itself (flowcharts and finite state machines are early examples of software modeling), the majority of practitioners have generally been slow in accepting it as anything more than a minor power assist. Because this is still the dominant attitude, model-based development methods are encountering a great deal of resistance.

Even though some of this can be ascribed to an irrational fear of change, there are some valid reasons why practitioners doubt the value of models. Probably the most important is that experience has shown that software models are often wildly inaccurate, sometimes obscuring fatal design flaws behind fancy but ambiguous graphics. Clearly, the practical value of any model increases with its accuracy. If a model cannot be trusted to tell us what we need to know about the software system that it represents, then it can be even worse than useless because it can lead to the wrong conclusions. The key, then, to increasing the value of software models is to narrow the semantic gap between them and the systems they are modeling. However, as we shall explain later, it turns out that this is far easier to do with software than with any other engineering medium.

Much of the inaccuracy of software models is due to the extremely detailed and sensitive nature of current programming languages. Minor lapses and barely detectable coding errors, such as misaligned pointers or uninitialized variables, can have enormous but generally unpredictable consequences. For instance, there is a well-documented case where a single missing 'break' statement in a C program resulted in the loss of long-distance telephone service for a large part of the United States. The economic damage this caused was estimated to be in the hundreds of millions of dollars.[9] This "chaotic" nature of modern software technologies, where a seemingly minute defect can have major effects on the overall system, makes it very difficult to model software systems accurately. After all, the essence of modeling lies in abstraction or the removal of unessential detail. Because it is difficult to predict which fragment of software is unessential, how is it possible to have a model of software that is accurate and yet abstract enough to be useful?

One very effective solution to this dilemma is to *formally* link a model with its corresponding software implementation through one or a series of *automated* model transformations. Perhaps the best and most successful exemplar of that approach is the compiler, which automatically translates a high-level language program into an equivalent machine language implementation. In this case, the "model"

is the high-level language program, which, like all useful models, hides irrelevant detail such as the idiosyncrasies of the underlying computing technology (e.g., internal word size, word orientation, the number of accumulators and index registers, the details of arithmetic and logic unit (ALU) programming).

Few if any engineering media other than software can provide such a tight coupling between a model and its corresponding engineering artifact. A model of any kind of physical artifact (automobile, building, bridge, etc.) inevitably involves an informal step of abstracting the physical characteristics into a corresponding formal model, such as a mathematical model or a scaled-down physical model. Similarly, implementing an abstract model with physical materials involves an informal transformation from the abstract into the concrete. The informal nature of this step can lead to inaccuracies that, as noted above, can render the models ineffective or even counterproductive. When it comes to software, however, the elements that are being modeled typically come from the world of ideas and are generally unfettered by intricate physical detail or constraints. By judicious selection of software abstractions and through the precise definition of their semantics, the transition from an abstraction to its software realization (and vice versa) can be automated without loss of accuracy. In this sense, software is an engineer's dream material, in which the model and its realization can be perfectly coupled to each other.

This potent combination of *abstraction* and *automation* has inspired a set of modeling technologies and corresponding development methods collectively referred to as *model-driven development* (MDD).[10,11] The defining feature of MDD is that models have become primary artifacts of software design, shifting much of the focus away from program code. Models serve as blueprints from which programs and related models are derived by various automated and semiautomated means. The degree of automation being applied today varies from simple skeleton code derivation all the way through to complete automatic code generation (comparable to traditional compilation). Clearly, the greater the levels of automation, the more accurate the models and the greater the benefits of MDD. However, there are many factors that must be considered when selecting an optimal level of automation for a given project. This includes, for example, the availability of appropriate expertise and tools, the specific nature of the application, the amount and characteristics of legacy code, and so on.

Model-driven methods of software development are not particularly new and have been used in the past with varying degrees of success. The reason they are receiving more attention now is that the supporting technologies have matured to the point where automation can be used in practical applications to a much larger extent. This is because the new technologies are much more scalable, more efficient, and much more easily integrated with existing tools and methods than was the case in the past. The degree of maturity of these technologies has reached a point where many of their aspects can be standardized—resulting in a commoditization of much MDD tooling.

To that end, OMG, the industry consortium that first standardized UML, has launched an initiative to develop a body of standards that support MDD. Called Model-Driven Architecture** (MDA**), the initiative involved standards for modeling languages, such as UML,[7] standards for defining modeling languages like the Meta-Object Facility or MOF**,[12] standards for defining automated model transformations, standards for defining model-based software processes, and so on.

In the remainder of this article, we will examine how the latest version of the UML standard, UML 2, has been adjusted to meet the needs of MDD. First, we examine the forces that led to the revision of the original standard. This is followed by a summary of the major new language capabilities. For convenience, they have been grouped into five major categories of changes. Each of these is then described in a section of its own. The article concludes with a view of current and anticipated developments related to UML.

THE RATIONALE BEHIND UML 2

UML 2 is the first major revision of the UML standard, following a series of lesser revisions.[7,8] Why was it necessary to revise UML?

The original UML standard was primarily designed with the traditional development process in mind: the model was primarily a means for documenting

and communicating high-level design ideas. This did not require a precise modeling language. Nonetheless, a growing number of software architects wanted their UML models to be precise specifications that could serve as formal blueprints to be faithfully realized by the corresponding software implementation. Any ambiguity in such models could lead to misinterpretations and invalid realizations. This created pressure to define the semantics of UML much more precisely. Simultaneously, many programmers were beginning to see the benefits of more abstract graphical representations of their code, representations that were shorn of the noise of programming-language syntax and more clearly rendered its essence. For example, a graph-based rendering of a class hierarchy that shows relationships between classes visually is generally more easily understood than the corresponding textual representation. This quickly led to the requirement to allow the code to be manipulated in either graphical or textual form, whichever happened to be more convenient at the time. Therefore, it was necessary to define very precisely the formal relationship between the graphics and the code and also the semantics of UML diagrams.

Both tool vendors and users responded to this pressure by defining individual specializations of UML. Unfortunately, these custom variants differed from case to case and from project to project, often based on dubious or invalid interpretations of the underlying UML concepts. This threatened to lead to the same kind of fragmentation that the original standard was intended to eliminate. A new, more precise version of the standard was clearly necessary to reduce the ambiguities of the original standard. In addition, a more capable and more clearly defined mechanism was required to support domain-specific specializations of UML.

Whereas the pressure towards MDD was the primary motivator for UML 2, another key factor was the need to model important new technologies that had emerged since the first release of the standard, such as Web-based applications and service-oriented architectures. Although all of these could be represented by appropriate combinations of existing UML 1 concepts, there were obvious benefits to providing more direct ways of modeling these capabilities.

Finally, although we still lack a sound and systematic theory of modeling language design, much has been learned about suitable ways of defining, structuring, and using such languages. For example, new theories of meta-modeling and of model transformations have emerged over the past 10 years, which need to be incorporated into UML to ensure its applicability and longevity. Although UML might end up being the equivalent of FORTRAN in the domain of software modeling languages, it is worth recalling that FORTRAN is still an active language, almost 50 years after its inception.

WHAT IS NEW IN UML 2

The new developments in UML 2 can be grouped into the following five major categories, listed in decreasing order of significance:

1. *A significantly higher level of precision in the definition of the language*—This is a result of the need to support the higher levels of automation required for MDD. Automation implies the elimination of ambiguity and imprecision from models (and, hence, from the modeling language) so that they can be transformed and analyzed by specialized computer programs.
2. *An improved language organization*—This is characterized by a modularity that not only makes the language more approachable to new users but also facilitates inter-working between tools.
3. *Significant improvements in the ability to model large-scale software systems*—Some modern software applications represent integration of existing stand-alone applications into more complex systems of systems. This is a trend that will likely continue, resulting in ever more complex systems. To support such trends, flexible new hierarchical capabilities were added to the language to support software modeling at arbitrary levels of complexity.
4. *Improved support for domain-based specialization*—Practical experience with UML demonstrated the value of its extension mechanisms. These were consolidated and refined to allow simpler and more precise refinements of the base language.
5. *Overall consolidation, rationalization, and clarification of various modeling concepts resulting in a simplified and more consistent language*—This involved consolidation of concepts, removal of redundant concepts, refinement of definitions, and the addition of clarifications and examples.

Each of the these categories is described individually below.

INCREASED PRECISION OF LANGUAGE DEFINITION

Most early software modeling languages were defined informally, with little attention paid to precision. More often than not, modeling concepts were explained using imprecise and informal natural language. This was deemed sufficient at the time because the majority of modeling languages were used either for documentation or for what Martin Fowler refers to as design "sketching".[13] The idea was to convey the essential properties of a design, leaving detail to be worked out during implementation.

This, however, often led to confusion because models expressed in such languages could be—and often were—interpreted differently by different individuals. Furthermore, unless the question of model interpretation was explicitly discussed up front, such differences could remain undetected, to be unmasked only in the latter phases of development when the cost of fixing the resulting problems was much greater.

In contrast to most other modeling languages of the time, to minimize ambiguity the first standardized definition of UML was specified using a *metamodel*. This is a model that defines the characteristics of each UML modeling concept and its relationships to other modeling concepts. The metamodel was defined using what is, in essence, an elementary subset of UML called MOF, consisting primarily of concepts defined in UML class diagrams and supplemented with a set of formal constraints written in the Object Constraint Language (OCL). This combination represented a formal specification of the *abstract syntax* of UML (in contrast to its *concrete syntax* or notation). The abstract syntax is the set of rules that can be used to determine whether a given UML model is *well formed*. For example, such rules would allow us to determine that a model in which two UML classes are joined by a state machine transition is illegal.

Nonetheless, the degree of precision used in this initial UML metamodel proved insufficient to support the full potential behind MDD (see, for example, the discussion in Reference 14). In particular, the specification of the *semantics*, or meaning, of the UML modeling concepts remained inadequate for MDD-oriented activities such as automatic code generation or formal verification.

Consequently, the degree of precision used in the definition of UML 2 was increased significantly. This was achieved by the following means:

- *A major refactoring of the language metamodel*—The metamodel of UML, specified using the MOF language,[12] defines the formal rules to which a well-formed (i.e., syntactically correct) UML model must adhere. For UML 2, the core of this metamodel was broken up into a set of fine-grained low-level modeling concepts and patterns that are, in most cases, too rudimentary or too abstract to be used directly in modeling software applications. However, their relative simplicity makes it relatively easy to be precise about their semantics and the corresponding well-formedness rules. These finer-grained concepts are then combined to produce the more complex user-level modeling concepts. For instance, in UML 1, the notion of ownership (i.e., elements owning other elements), the concept of namespaces (named collections of uniquely named elements), and the concept of classifier (elements that can be categorized according to their features), were all inextricably bound into a single semantically complex notion. (This also meant that it was impossible to use any one of these without implying the other two.) In the new UML 2 metamodel, these concepts were separated, and their syntax and semantics were defined separately.
- *Extended and more precise semantics descriptions*—Defining the semantics of the UML 1 modeling concepts was problematic in a number of ways. The level of description was highly uneven, with some areas having extensive and detailed descriptions (e.g., state machines), whereas others had little or no explanations. The UML 2 specification puts much more emphasis on the semantics and, in particular, in the key area of basic behavioral dynamics (see below). (A more detailed discussion of the semantics of UML can be found in Reference 15.)
- *A clearly defined dynamic semantic framework*—The UML 2 specification clarifies some of the critical semantic gaps in the original version, including a clear specification of the relationship

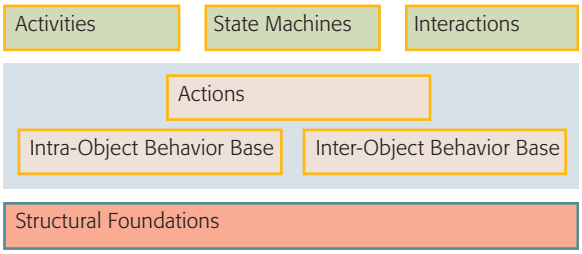

Figure 1
The UML 2 semantics framework

between structure and behavior as illustrated in *Figure 1*.

Note that the bottom (foundational) layer in this framework deals with the semantics of structure. This is because UML is, at its core, based on the object paradigm, wherein all behavior is assumed to emanate from the actions of objects. This core layer covers the essential properties of the structural concepts of UML, such as objects, variables, and links, which provide the setting for behavior. Overlaid on the foundational layer there is another shared semantic layer. This layer, represented by the middle area in Figure 1, is concerned with how the core structural elements are created and manipulated (the Intra-Object Behavior Base) as well as with how objects communicate with each other (the Inter-Object Behavior Base). The semantics of primitive actions that can cause this behavior are also part of this layer. The two shared layers in this framework provide the shared foundation on which the dynamic semantics of higher-level formalisms, such as state machines and interactions, are based. This ensures that objects can interact with each other regardless of which formalism is used to describe their behavior. More details can be found in References 15 and 7.

THE NEW LANGUAGE ARCHITECTURE

One of the immediate consequences of the increased level of precision in UML 2 is that the language definition has gotten bigger, even without accounting for the new modeling capabilities. This would normally be of concern, especially given that the original UML was criticized as being too rich and, therefore, too cumbersome to learn and use. Such criticisms typically ignore the fact that UML is intended to address some of today's most complex software problems and that such problems demand sufficiently powerful tools. (Successful technologies such as automobiles and electronics have never gotten simpler; it is a part of human nature to persistently demand more of our machinery, which, ultimately, implies more sophisticated tools. No one would even contemplate building a modern skyscraper with basic hand tools.)

To deal with the problem of language complexity, UML 2 was modularized in a way that allows selective use of language modules. The general form of this structure is shown in *Figure 2*. It consists of a foundation comprising shared structural and behavioral modeling concepts, such as classes and associations, on top of which is a collection of vertical "sub-languages" or *language units*, each one suited to modeling a specific form or aspect (see *Table 1*). These vertical language units are generally independent of each other and can, therefore, be used independently. This was not the case in UML 1, where, for example, the activities formalism was based entirely on the state machine formalism.

Furthermore, the vertical language units are hierarchically organized into up to three levels, with each successive level adding more modeling capabilities to those available in the levels below. This provides an additional dimension of modularity so that, even within a given language unit, it is possible to use only specific subsets.

This architecture means that users can learn and use only the subset of UML that suits them best. It is no more necessary to become familiar with the full extent of UML in order to use it effectively than it is necessary to learn all of English to use it effectively. As they gain experience, users have the option of gradually introducing more powerful modeling concepts as necessary.

As part of the same architectural reorganization, the definition and structure of compliance has been significantly simplified in UML 2. In UML 1, the basic units of compliance were defined by the packages of the metamodel, with literally hundreds of possible combinations. This meant that it was highly unlikely to find two or more modeling tools that could interchange models, because each would likely support a different combination of packages.

In UML 2, only three levels of compliance are defined, and those correspond to the hierarchical

language unit levels already mentioned and depicted in Figure 2. (The infrastructure of UML defines an additional two levels, but those are not visible or of particular interest to the general UML user.) These compliance levels are defined in such a way that models at level n (n = 1,2) are compliant with the higher compliance levels; that is, a tool compliant with a given level can import models produced by tools that are compliant with any level equal to or below its own, without loss of information.

In addition to the compliance levels, four distinct types of compliance are also defined that cut across each of the compliance levels:

1. *Compliance with the abstract syntax*—This means compliance with the well-formedness rules of UML as defined by the UML 2 metamodel. It also includes the ability to interchange models with other tools.
2. *Compliance with the concrete syntax*—This means support for the UML 2 notation as defined in the UML standard. In principle, it is possible for tools to comply with the notation without necessarily supporting the abstract syntax. This form of compliance is intended for relatively simple tools whose primary purpose is to assist in the drawing of UML diagrams.
3. *Compliance with both abstract and concrete syntax*—This type of compliance combines the two forms of compliance listed above and is presumed to be supported by most tools. Compliance of this type means compatibility with either of the previous two types (at a given level of compliance).
4. *Compliance with both the abstract and concrete syntax and the diagram interchange standard*—This form of compliance adds support for the diagram interchange standard,[16] which ensures the preservation of graphical information related to a model, such as font selections, position and sizing of graphical elements, and so forth, when models are exchanged between compliant tools.

For example, a given tool might provide abstract syntax compliance up to level 2 but concrete syntax compliance only up to level 1. This means that it does not support the standard notation for all the level-2 concepts which it provides (e.g., it may use a vendor-specific notation for some or all of the level-2 concepts, and at the same time, use the standard UML notation for the level-1 concepts).

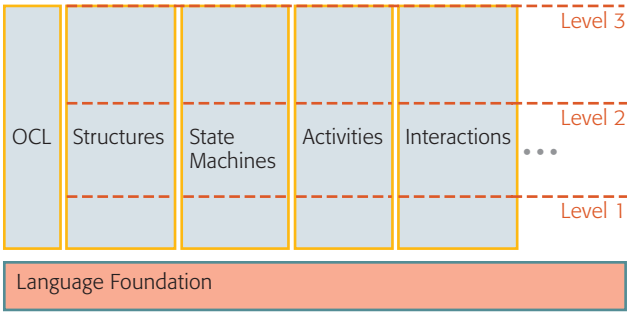

Figure 2
The language architecture of UML 2

Table 1 The language units of UML 2

Language Unit	Purpose
Actions	(Foundation) modeling of fine-grained actions
Activities	Data and control flow behavior modeling
Classes	(Foundation) modeling of basic structures
Components	Complex structure modeling for component technologies
Deployments	Deployment modeling
General Behaviors	(Foundation) common behavioral semantic base and time modeling
Information Flows	Abstract data flow modeling
Interactions	Inter-object behavior modeling
Models	Model organization
Profiles	Language customization
State Machines	Event-driven behavior modeling
Structures	Complex structure modeling
Templates	Pattern modeling
Use Cases	Behavioral requirements modeling

This matrix of three compliance levels and four types yields 12 different forms of compliance with varying degrees of capability, such that certain less capable forms are upward compatible with certain more capable forms. Consequently, in UML 2, model interchange between compliant tools from different

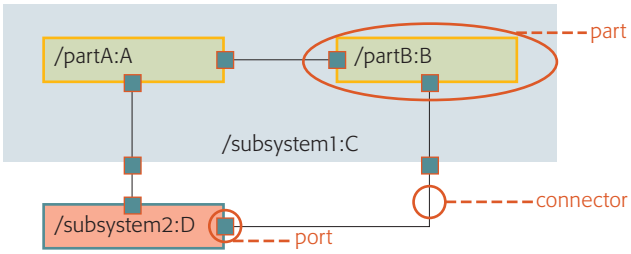

Figure 3
Example of the use of new structure modeling concepts

vendors is now more than just a theoretical possibility.

MODELING OF LARGE-SCALE SYSTEMS

The number of features added in UML 2 is intentionally relatively small in order to avoid the infamous "second system" effect,[17] whereby a language becomes bloated by an excess of new features demanded by a highly diverse user community. In fact, the majority of new modeling capabilities are essentially simply extensions of existing features that allow them to be used for modeling large-scale software systems. Moreover, these extensions were all achieved by using the same basic approach: recursive application of the same basic set of concepts at different levels of abstraction. This means that model elements of a given type could be combined into units that, in turn, would be used as the building blocks to be combined in the same way at the next level of abstraction, and so on—analogous to the way that procedures in programming languages could be nested within other procedures to any desired depth.

Specifically, the following modeling capabilities are extended in this way:

- Complex structures
- Activities
- Interactions
- State machines

The first three of these capabilities account for more than 90 percent of the new features added to UML 2.

Complex structures

The basis for this set of features comes from long-term experience with various architectural description languages, such as UML-RT,[18] Acme,[19] and SDL.[20] These languages are characterized by a relatively simple set of graph-like concepts: basic structural nodes called *parts* that may have one or more interaction points called *ports* and that are interconnected by communication channels called *connectors*. Aggregates of this type may be encapsulated within higher-level units, which can have their own ports so that they can themselves be combined with other similar units into yet higher-level compositions, as shown in ***Figure 3***.

In the example, a collaboration structure consisting of internal parts and connectors is nested within a class specification. This means that, upon creation, all instances of this class will have an internal structure specified by the class definition. For example, in Figure 3, parts/partA:A and/partB:B are nested within part/subsystem1:C. The latter represents an instance of the composite class C. Note that other instances of class C have the same structural pattern including all the ports, internal parts, and interconnections.

The rudiments of this type of structural composition based on parts and connectors existed in the UML 1 collaboration diagrams. However, it was not possible to easily use the concepts recursively to construct multilevel structural decomposition hierarchies. Also, the crucial port concept was missing. This important concept serves a dual purpose.

First, a port allows an object to distinguish between different potentially concurrent collaborators, based on which port is used for a given interaction. In principle, each port could present a different interface, depending on the type of interaction taking place through that port. This type of interface separation is particularly useful when modeling complex architectural-level components, which are often involved in multiple interactions. In addition, ports act as intermediaries, relaying information back and forth between the internal entities of the component and its environment. If all external interactions of a component occur through its ports, then its internal entities are fully isolated from any direct knowledge of any external entities. Consequently, the same component definition can be reused in many different applications without any modification. In other words, ports enable true two-way encapsulation of components by preventing direct coupling between component internal entities and external entities in either direction.

It turns out that by simple recursive application of these three simple concepts (ports, parts, and connectors), it is possible to model arbitrarily complex software systems.

Activities

Activities in UML are used to model flows of various kinds: signal or data flows as well as algorithmic and procedural (i.e., control) flows. There are numerous domains and applications that are most naturally rendered by such flow-based descriptions. In particular, this formalism was embraced by business-process modelers and by systems engineers, who tend to view many of their systems as interconnecting signal processors.

The UML 1 version of activity modeling had a number of serious limitations in the types of flows that could be represented. Many of these were due to the fact that activities were overlaid on top of the basic state-machine formalism and were, therefore, constrained to the semantics of state machines.

UML 2 replaced the state-machine underpinning with a much more general semantic foundation based on Petri nets, which eliminates all of these restrictions. In addition, inspired by a number of industry-standard business-processing formalisms, including notably BPEL4WS,[21] a very rich set of new and highly refined modeling features were added to the basic formalism. These include the ability to represent interrupted activity flows, sophisticated forms of concurrency control, and diverse buffering schemes. The result is a very rich modeling toolset that can represent a wide variety of flow types.

The integration of the UML action semantics specification into the new semantic foundations provided for activities is an important new development. UML action semantics were first introduced as a separate OMG specification, which was subsequently included in the UML 1.5 revision as an addendum. Action semantics provide a language-neutral facility for specifying detail-level behavior in the context of a UML model (see also Figure 1). This includes the definition of actions that create and destroy objects, that read and write object attributes and variables, that invoke operations and send signals, and so forth. In effect, action semantics complement the higher-level modeling capabilities of UML to the extent that it is possible to use UML as a fully-fledged implementation language. The rules for combining UML actions using control and data flows are essentially the same as the composition rules for combining activities, so that the consolidation of these two areas resulted in a significant overall simplification.

As with other complex structures, activities and their interconnecting flows can be recursively grouped into higher-level activities with clearly defined inputs and outputs. These can, in turn, be combined with other activities to form more complex activities, up to the highest system levels.

Interactions

Interactions in UML 1 were represented either as sequenced message annotations on collaboration diagrams or as separate sequence diagrams. Unfortunately, two fundamental capabilities were missing:

1. The ability to reuse sequences that may be repeated in the context of more extensive (higher level) sequences. For example, a sequence that validates a password may appear in multiple contexts in a given application. If such repeated sequences cannot be packaged into separate units, they have to be defined numerous times, not only adding overhead but also complicating model maintenance (e.g., when the sequence needs to be changed).
2. The ability to adequately model various complex control flows that are common in representing interactions of complex systems, including repetition of subsequences, alternative execution paths, and concurrent and order-independent execution.

Fortunately, the problem of specifying complex interactions was extensively studied in the telecommunications domain, where a standard was evolved based on many years of practical experience in defining communications protocols.[22] This formalism was used as a basis for representing interactions in UML 2.

The key innovation was the introduction of an interaction as a separately named modeling unit. Such an interaction represents a sequence of inter-object communications of arbitrary complexity. It may even be parameterized to allow the specification of context-independent interaction patterns.

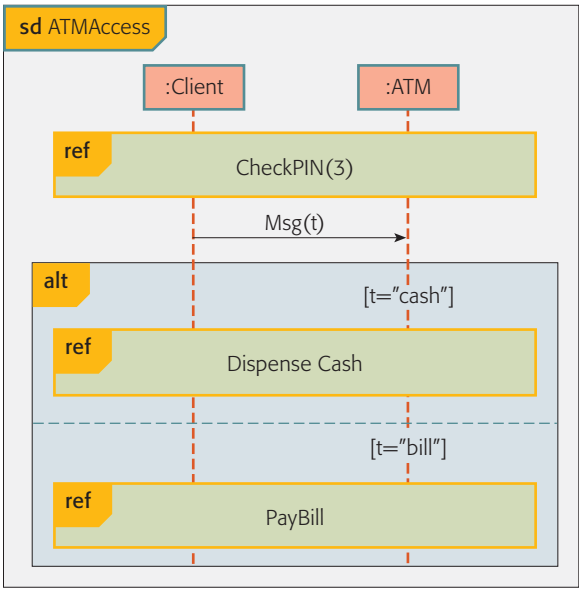

Figure 4
Example of a complex interaction model

These "packaged" interactions can be invoked recursively from within higher-level interactions analogous to macro or subroutine invocations (the "ref" block in ***Figure 4*** labeled CheckPIN). Just like macros and subroutines, they provide both a reuse facility and an abstraction facility.

As one might expect, such references to other interactions can be nested to an arbitrary degree—yet another example of the use of recursion in UML 2 to achieve scalability. Furthermore, interactions can serve as operands in complex control constructs such as loops (for example, a given interaction may have to be repeated some number of times) and alternatives. UML 2 defines a number of convenient modeling constructs of this type, providing a very rich facility for modeling complex end-to-end behavior at any level of decomposition.

In Figure 4, we see an example of an extended interaction, specified in the form of a sequence diagram (sd), which models the use of an ATM machine, In this case, the interaction `ATMAccess` first references (i.e., invokes) another lower-level transaction called `CheckPIN` (the contents of this interaction are not shown in the diagram).

Note that the latter interaction has a parameter (in this case, say, the number of times an invalid personal identification number (PIN) can be entered before the transaction is canceled). After that, the client sends an asynchronous message specifying what kind of interaction is required and, based on the value in that message, one of two possible further execution paths is selected (i.e., either the `DispenseCash` interaction or the `PayBill` interaction is performed). For compactness, both alternatives are specified in the same diagram. This is indicated by enclosing them in the same "alt" (alternative) block.

Interactions in UML 2 can be represented by sequence diagrams as shown in the preceding example as well as by other diagram types—including the collaboration-based form defined in UML 1. There is even a nongraphical tabular representation.

State machines

The main new capability added to state machines in UML 2 is reminiscent of the aforementioned "ref" concept in interactions: the ability to define a generic state-machine pattern and then reuse it in different situations. The reusable state-machine pattern is called a *submachine*. It is like any other state-machine definition with one important difference: it may include one or more *entry* and *exit* pseudostates. These are points through which the submachine is bound to its invoking context. Specifically, entry points are points through which an incoming transition contained in the invoking state machine can enter the submachine, and exit points are points that can be bound to outgoing transitions in the invoking state machine.

An example can be seen in ***Figure 5***. Figure 5A shows the definition of the submachine `CheckPIN`, which specifies the behavior required to input a PIN on an ATM and validate it against a database of valid PINs. This submachine can then be invoked in higher-level state machines where appropriate. One example of such an invocation is shown in Figure 5B, where the state `CheckingPIN` represents an invocation of the `CheckPIN` submachine.

One other notable state-machine innovation in UML 2 is a clarification of the semantics of state-machine inheritance. In effect, a subclass inherits the state machine of its parents and may add new elements (e.g., states, transitions, triggers) or redefine existing elements in a compatible way.

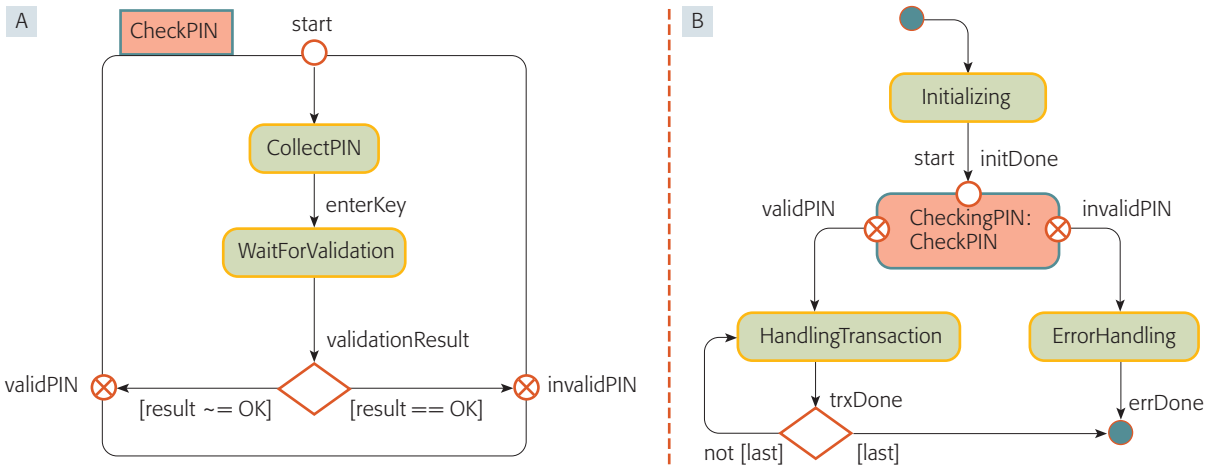

Figure 5
Example of a reusable submachine and its invocation

LANGUAGE SPECIALIZATION CAPABILITIES

From its inception, UML was conceived as a platform for a family of related modeling languages—languages that share a common semantics framework and, possibly, a common notation. This was achieved by providing so-called *semantic variation points* in the definition of the language. These are areas where the standard either provides a selection of alternatives (e.g., single or multiple inheritance) or leaves certain details unspecified (e.g., scheduling policy, method dispatching rules). The language is then customized by adding the necessary constraints and extensions. However, any such extensions must not violate the standard abstract syntax and semantics.

Experience with UML 1 has proven this to be a good design decision, because a very common way of applying UML is to first define a UML profile for a particular problem or domain and then use that profile instead of or in addition to standard UML. In essence, profiles are a way of producing what are now commonly referred to as *domain-specific languages* (DSLs).

An alternative to using UML profiles is to define a new custom modeling language using the MOF standard and tools. The latter approach has the obvious advantage of providing a clean slate, enabling the definition of a language that is optimally suited to the problem at hand. At first glance, this may seem preferable to DSL definition, but closer scrutiny reveals that there can be serious drawbacks to this approach.

As noted in the introduction, too much diversity leads to the kind of fragmentation problems that UML was designed to eliminate. In fact, this is one of the primary reasons why it was accepted so widely and so rapidly.

Fortunately, the profile mechanism provides a convenient solution for many practical cases. This is because there is typically a lot of commonality even between diverse DSLs. For example, practically any object-oriented modeling language needs to define the concepts of classes, attributes, associations, interactions, and so forth. UML, which is a general-purpose modeling language, provides just such a convenient and carefully defined collection of useful concepts. This makes it a good starting point for a large number of possible DSLs.

There is more than just conceptual reuse at play though. Because a UML profile has to be compatible with standard UML by definition, (1) any tool that supports standard UML can be used for manipulating models based on that profile, and (2) any knowledge of and experience with standard UML is directly applicable. Therefore, many of the fragmentation problems stemming from diversity can be mitigated or even avoided altogether. This type of reasoning led the international standards body responsible for the SDL language[20]—a DSL widely

used in telecommunication—to redefine SDL as a UML profile.[23–24]

This is not to say that all DSL can and should be realized as a UML profile; there are indeed many cases where UML may lack the requisite foundational concepts that can be cast into corresponding DSL concepts. However, given the generality of UML, it may be more widely applicable than might first appear.

With these considerations in mind, the profiling mechanism in UML 2 has been rationalized and its capabilities extended. The conceptual connection between a stereotype and the UML concepts that it extends has been clarified. In effect, a UML 2 stereotype is defined as if it were simply a subclass of an existing UML metaclass, with associated attributes (representing tags for tagged values), operations, and constraints. The mechanisms for writing such constraints using a language such as OCL have been fully specified.

In addition to constraining individual modeling concepts, a UML 2 profile can also explicitly hide UML concepts that make no sense or are unnecessary in a given DSL. This allows the definition of minimal DSL profiles.

Finally, the UML-2 profiling mechanism can also be used as a mechanism for viewing a complex UML model from multiple different domain-specific perspectives—something generally not possible with DSLs (i.e., a UML profile can be selectively "applied" or "deapplied" without affecting the underlying UML model in any way). For example, a performance engineer may choose to apply a performance-modeling interpretation over a model, attaching various performance-related measures to elements of the model. These can then be used by an automated performance analysis tool to determine the fundamental performance properties of a software design. At the same time and independent of the performance modeler, a reliability engineer might overlay a reliability-specific view on the same model to determine its overall reliability characteristics, and so on.

CONSOLIDATION OF CONCEPTS

The consolidation of concepts includes the removal of overlapping concepts and numerous editorial modifications, such as clarifying confusing descriptions and standardizing terminology and specification formats.

The removal of overlapping concepts and the clarification of poorly defined concepts have been other important requirements for UML 2. The three major areas affected by this are actions and activities, templates, and component-based design concepts.

The consolidation of actions and activities was described earlier. From the user's point of view, these are formalisms that occur at different levels of abstraction because they typically model phenomena at different levels of granularity. However, the shared conceptual base results in an overall simplification and greater clarity.

In UML 1, templates were defined very generally: any UML concept could be made into a template. Unfortunately, this generality was an impediment to the application of the concept because it allowed for potentially meaningless template types and template substitutions. The template mechanism in UML 2 was restricted to cases that were well understood: classifiers, operations, and packages. The first two were modeled after template mechanisms found in popular programming languages.

In the area of component-based design, UML 1 had a confusing abundance of concepts. One could use classes, components, or subsystems. These concepts had much in common but were subtly different in non-obvious ways. There was no clear delineation as to which to use in any given situation. Was a subsystem just a "big" component? If so, how big did a component have to be before it became a subsystem? Classes provided encapsulation and realized interfaces, but so did components and subsystems.

In UML 2, all of these concepts were aligned, so that components were simply defined as a special case of the more general concept of a structured class, and, similarly, subsystems were merely a special case of the component concept. The qualitative differences between these were clearly identified so that decisions on when to use which concept could be made on the basis of objective criteria.

On the editorial side, the format of the specification was consolidated with the semantics and notation

specifications for the modeling concepts, combined for easier reference.

Each metaclass specification has been expanded with information that explicitly identifies semantic variation points, notational options, and the relationship of the specification to the UML 1 specifications. Also, the terminology has been made consistent so that a given term (e.g., type, instance, specification, occurrence) has the same general connotation in all contexts in which it appears.

CONCLUSION

UML 2 was specifically designed to allow a gradual introduction of model-driven methods into software development. For those who prefer it as a "design sketching" tool, it can still be used in the same informal manner as UML 1. Moreover, because the new modeling capabilities are nonintrusive, in most cases such users will not see any change in the look and feel of the language.

The opportunity to use UML for more advanced forms of MDD is now open. The increased precision and enhanced semantics definition are available in the revised standard, to be used—if desired—with very sophisticated automatic code generation techniques.

Although the language has added some new features, its overall structure was carefully reorganized to allow a modular and graduated approach to adoption: users only need to learn the parts of the language that are of interest to them and can safely ignore the rest. As their experience and knowledge increases, they can selectively add new language modules. This reorganization of the language also includes a major simplification of the compliance strategy to facilitate interoperability between complementary tools as well as between tools from different vendors.

To avoid language bloat, only a small number of new features were added, and practically all of those features were designed along the same recursive principle that enabled the modeling of very large and complex systems. In particular, extensions were added to more directly model software architectures, complex system interactions, and flow-based models for applications such as business-process modeling and systems engineering.

The language extension mechanisms were slightly restructured and simplified for a more direct way of defining DSLs based on UML. These languages can directly take advantage of UML tools and expertise, both of which are abundantly available.

The overall result is a second-generation modeling language that has the potential to help developers construct more sophisticated software systems faster and more reliably. In essence, software development with UML 2 need not be different from traditional software design, except that it is based on higher levels of abstraction and automation. It requires the same types of intuition, skill, and expertise that are the bread and butter of every software developer.

At the time of writing, the first minor revision of the original UML 2 specification has been finalized, resulting in UML 2.1. This revision adds fixes to the abstract syntax to eliminate minor inconsistencies and ambiguities. No significant feature additions to UML 2 are anticipated over the next several years. In general, such standards should not change too frequently, as it is difficult for users, tool vendors, and book authors to keep up. There should be enough of a pause to allow ample time for sufficient experience with the present version to accumulate and for new technologies and relevant theoretical developments to emerge. Only after these are well understood should another major revision be considered.

CITED REFERENCES

1. I. Graham, *Object-Oriented Methods: Principles and Practice (3rd edition)*, Addison-Wesley, Reading, MA (2001).
2. J. Rumbaugh, M. Blaha, W. Lorenson, F. Eddy, and W. Premerlani, *Object-Oriented Modeling and Design, Prentice Hall*, Upper Saddle River, NJ (1990).
3. G. Booch, *Object-Oriented Analysis and Design with Applications (2nd edition)*, Addison-Wesley Professional, Reading MA (1993).
4. I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering: A Use Case*

Driven Approach, Addison-Wesley Professional, Reading, MA (1992).

5. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming* **8**, No. 3, 231–274 (1987).
6. *Unified Modeling Language (UML), Version 1.5*, OMG document formal/03-03-01, Object Management Group (2003), http://www.omg.org/cgi-bin/doc?formal/03-03-01.
7. *UML 2.0 Superstructure Specification*, OMG document formal/05-07-04, Object Management Group, Inc. (2005), http://www.omg.org/cgi-bin/doc?formal/05-07-04.
8. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual (2nd edition)*, Addison-Wesley, Reading, MA (2005).
9. L. Lee, *The Day the Phones Stopped Ringing*, Plume Publishing (1992).
10. A. Brown, "An Introduction to Model Driven Architecture," developerWorks, IBM Corporation (2004), http://www-106.ibm.com/developerworks/rational/library/3100.html.
11. G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, and B. Selic, "An MDA Manifesto," in *The MDA Journal*, D. Frankel and J. Parodi, Editors, Meghan-Kiffer Press (2004).
12. *MetaObject Facility (MOF) 2.0 Core Specification*, Available Specification, OMG document ptc/04-10-15, Object Management Group (2004), http://www.omg.org/cgi-bin/doc?ptc/2004-10-15.
13. M. Fowler, *UML Distilled (3rd edition)*, Addison-Wesley, Reading, MA (2004).
14. P. Stevens, "On the Interpretation of Binary Associations in the Unified Modeling Language," *Journal of Software and Systems Modeling* **1**, No. 1, 68–79 (2002).
15. B. Selic, "On the Semantic Foundations of Standard UML 2.0," *Formal Methods for the Design of Real-Time Systems,* in *Lecture Notes in Computer Science* **3185**, M. Bernardo and F. Corradini, Editors, Springer-Verlag (2004), pp. 181–199.
16. *UML 2.0 Diagram Interchange, Final Adopted Specification*, OMG document ptc/03-09-01, Object Management Group (2004), http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-01.pdf.
17. F. Brooks, Jr., *The Mythical Man-Month (1995 edition)*, Addison-Wesley, Reading, MA (1995).
18. B. Selic, "Using UML for Modeling Complex Real-Time Systems," *Languages, Compilers, and Tools for Embedded Systems,* in *Lecture Notes in Computer Science* **1474**, F. Mueller and A. Bestavros, Editors, Springer-Verlag (1998), pp. 250–260.
19. D. Garlan, R. Monroe, and D. Wile, "Acme: an Architecture Description Interchange Language," *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, ACM, New York (1997), p. 7.
20. International Telecommunication Union, ITU Recommendation Z.100: Specification and Description Language (SDL), ITU-T (August 2002).
21. S. Thatte, *Business Process Execution Language for Web Services (Version 1.1)*, BEA Systems, Inc., IBM Corporation, Microsoft Corporation, SAP AG, and Siebel Systems (May 5, 2003), ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.
22. International Telecommunication Union, ITU Recommendation Z.120: Message Sequence Chart (MSC), ITU-T (April 2004).
23. International Telecommunication Union, ITU Recommendation Z.109: SDL Combined with UML, ITU-T (2000).
24. International Telecommunication Union, "Study Group 17: Question 13/17—System Design Languages Framework and Unified Modeling Language," ITU-T Study Group 17 (2003), http://www.itu.int/ITU-T/studygroups/com17/sg17-q13.html.

Accepted for publication December 26, 2005.
Published online July 11, 2006.

Bran Selic
IBM Rational Software, IBM Canada, 770 Palladium Dr., Kanata, Ontario, Canada, K2V 1C8 (bselic@ca.ibm.com).
Mr. Selic is an IBM Distinguished Engineer at IBM Canada working on the CTO team for IBM's Rational brand. He is also an Adjunct Professor of Computer Science at Carleton University in Ottawa, Canada. He has close to 40 years of experience in designing, implementing, and maintaining large-scale industrial software systems, working mostly with telecommunications, aerospace, robotics, and large financial systems. In the late 1980s, he pioneered the application of MDD methods and tools in the real-time domain and is the primary author of a reference text on this topic. In 1992, he cofounded ObjectTime Limited, a highly successful company that developed software tools for MDD. He is recognized as an expert in modeling language design and MDD and has written many papers and articles on this subject. A frequently invited speaker and lecturer at various technical conferences and symposia, he is currently chair of the OMG team responsible for maintaining the UML modeling language standard. He received a Dipl.Ing degree (1972) and a Mag.Ing. degree (1974) from the University of Belgrade in Belgrade, Yugoslavia. He has been living and working in Canada since 1977. ■