Rational. software

What's New in UML™ 2.0?

Bran Selic
IBM Distinguished Engineer, IBM Rational Software
Kanata, Ontario, Canada
bselic@ca.ibm.com

Contents

Introduction

The early part of the 1990s saw a greatly heightened interest in the object paradigm and related technologies. New object-based programming languages, such as SmallTalk, Eiffel, C++, and Java, were devised and adopted. These were accompanied by a prodigious and confusing glut of object-oriented (OO) software design methods and modeling notations. Thus, in his very thorough overview of OO analysis and design methods (covering more than 800 pages), Graham lists more than 50 "seminal" methods [Graham01]. Given that the object paradigm consists of relatively few fundamental concepts, including encapsulation, inheritance, and polymorphism, there was clearly heavy overlap and conceptual alignment across these methods—much of which was obscured by notational and other differences of no consequence. This caused great confusion and needless market fragmentation, which, in turn, impeded the adoption of the useful new paradigm. Software developers had to make difficult and binding choices between mutually incompatible languages, tools, methods, and vendors.

For this reason, when Rational Software proposed the Unified Modeling Language™ (UML™) initiative, led by Grady Booch, Ivar Jacobson, and Jim Rumbaugh, the reaction was immediate and positive. Rational did not intend to propose anything new, but—through collaboration among top industry thought leaders—consolidated the best features of the various OO approaches into one vendor-independent modeling language and notation. Because of that, UML quickly became the first de facto standard and, following its Object Management Group adoption in 1996, a bona-fide industry standard [OMG03a] [OMG04] [RJB05].

Since then, the majority of modeling tool vendors have adopted and supported UML in their tools. The language has became an essential part of the computer science and engineering curricula in universities throughout the world and in various professional training programs; academic and other researchers use it as a convenient lingua franca.

UML also helped raise general awareness about the value of modeling when dealing with software complexity. Although this highly useful technique is almost as old as software itself (with flowcharts and finite state machines as early examples), most practitioners have generally been slow to accept it as anything more than a minor power assist. It is fair to say that this is still the dominant attitude, which is why so-called "model-driven" methods are encountering great resistance in this community.

There are valid reasons for this situation[1]. The main one is that software models can often be inaccurate in unpredictable ways. Clearly, any model's practical value is directly proportional to its accuracy. If we cannot trust the model to tell us true things about the software system it represents, then the model is worse than useless—it can foster false conclusions. The key to increasing a software model's value then is to narrow the gap between it and the system it is modeling. Paradoxically, as we shall discuss later, this is easier to do in software than in any other engineering discipline.

You can blame some of this model inaccuracy on the extremely detailed and sensitive nature of current programming language technologies. Minor lapses and barely detectable coding errors, such as misaligned pointers or uninitialized variables, can have enormous consequences. For instance, a well-documented case noted that one missing break in one case of a nesting switch statement resulted in the loss of long-distance telephone service for a large part

1. In addition, some not-so-valid reasons, such as general human distrust of innovation.

of the United States, causing immense economic losses [Lee92].
If such seemingly minute detail can have such dire consequences, how can we trust models to be accurate, since models, by definition, are supposed to hide or remove detail?

The solution to this conundrum is to formally link a model to its corresponding software implementation through one or more automated model transformations. Perhaps the best and most successful exemplar of that can be found in the concept of a compiler, which translates a high-level language program into an equivalent machine language implementation. Like all useful models, the model—in this case, a high-level language program—hides irrelevant detail, such as the idiosyncrasies of the underlying computing technology (internal word size, the number of accumulators and index registers, the type of ALU, etc.).

(Note that few, if any, other engineering media can provide such a tight coupling between a model and its corresponding engineering artifact.
This is because the modeled artifact is software rather than hardware.
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 or scale model.
Similarly, implementing an abstract model using 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. In software, however, this transformation can, in principle, be performed formally in either direction.)

The potential behind this powerful combination of abstraction and automation has led to the emergence of new modeling technologies and corresponding development methods, collectively referred to as model-driven development (MDD) [Brown04] [Booch04]. MDD's defining feature is that models have become primary artifacts of software design, shifting the focus away from the corresponding program code. Models serve as blueprints from which programs and related models are derived by various automated and semi-automated processes. MDD's degrees of automation today vary from simple skeleton code derivation to complete automatic code generation (which is comparable to traditional compilation). Clearly, the greater the levels of automation, the more accurate the models and greater the MDD benefits become.

Model-driven methods are not particularly new and have been used in software development with varying degrees of success. They are receiving much more attention today because the supporting technologies have matured to the point where you can automate much more than you could in the past. This is not just in terms of efficiency but also in terms of scalability, and the ability of such tools to be integrated with legacy tools and methods. The emergence of MDD standards that result in the commoditization of corresponding tools plus the obvious benefits to users reflect this maturation. One of these MDD standards is the Unified Modeling Language version 2.0.

The rationale

UML 2.0 is the standard's first major revision, following a series of lesser minor revisions [OMG04] [RJB05]. So why was it necessary to revise UML?

The primary motivation came from the desire to better support MDD tools and methods. In the past decade, several vendors had developed UML-based tools that supported significantly greater levels of automation than traditional CASE tools. To support these higher forms of automation, it was necessary to define UML in much more precise terms than the original standard did[2]. Unfortunately, these definitions varied from vendor to vendor, threatening once again to lead to the kind of fragmentation that the original standard was intended to eliminate. A new version of the standard could rectify this.

In addition, after close to a decade of practical UML experience and the emergence of important new technologies during that time, such as Web-based applications and service-oriented architectures, new modeling capabilities were identified. While practically all of these could be represented by appropriate combinations of existing UML concepts, there were clear benefits to introducing some of these as first-class built-in language features.

Finally, during the same extensive period, much has been learned about suitable ways of using, structuring, and defining modeling languages[3]. For example, there are now emerging theories of meta-modeling and of model transformations, which impose certain demands on how a modeling language should be defined. The OMG needed to incorporate these and similar developments into UML to ensure its utility and longevity.

2. In tune with the times, the original UML standard was primarily designed to serve as an auxiliary tool for informal capture and communication of design intent.

3. However, we still lack a consolidated and systematic theory of modeling language design that is comparable to the current theory of programming language design.

The highlights of UML 2.0

We can group the new developments in UML 2.0 into the following five major categories, listed in order of significance:

1. *A significantly increased degree of precision in the language's definition. This addresses the need to support the higher levels of automation that MDD requires. Automation implies the elimination of model ambiguity and imprecision (and, hence, from the modeling language) so that computer programs can transform and manipulate models.*
2. *An improved language organization, characterized by a modularity that not only makes the language more approachable to new users but that also facilitates inter-working between tools.*
3. *Significant improvements in the ability to model large-scale software systems. Some modern software applications represent integrations of existing stand-alone applications into more complex systems of systems. This trend will likely continue, resulting in ever-more complex systems. To support such trends, the OMG added flexible new hierarchical capabilities to the language to support software modeling at arbitrary levels of complexity.*
4. *Improved support for domain-specific specialization. Practical experience with UML demonstrated the value of its so-called "extension" mechanisms. The OMG consolidated and refined these to allow simpler and more precise refinements of the base language.*
5. *Overall consolidation, rationalization, and clarifications of various modeling concepts resulting in a simplified and more consistent language. This involved consolidating and, in a few cases, removing redundant concepts, refining numerous definitions, and adding textual clarifications and examples.*

We now delve into each of these in more detail.

Degree of precision

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, since most modeling languages were used either for documentation or for what Martin Fowler referred to as design "sketching" [Fowler04]. The idea was to convey a design's essential properties, leaving developers to work out details during implementation.

However, this often led to confusion because different individuals could—and often did—interpret models expressed in such languages quite differently. Further, unless these individuals explicitly discussed model interpretation up front, such differences could remain undetected, until later in the development stage when costs to fix resulting problems are much greater.

To minimize ambiguity as well as in contrast to most other modeling languages of the time, the first standardized UML definition 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 an elementary subset of UML[4] and was supplemented by a set of formal constraints written in the Object Constraint Language (OCL). This combination represented a formal specification of UML's abstract syntax[5]; that is, it defined the set of rules that you can use to determine whether a given model is well formed. For example, such rules would inform us not to connect two UML classes by a state machine transition.

4. This subset of UML, primarily comprising concepts defined in UML class diagrams is called the Meta-Object Facility (MOF). This subset was chosen such that you could use it to define other modeling languages.

5. It is called "abstract" because it is independent of the actual notation or "concrete syntax" (e.g., text, graphics) that is used to represent models.

However, 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 [Stevens02]). In particular, the specification of the semantics, or meaning, of the UML modeling concepts remained inadequate for such MDD-oriented activities as automatic code generation or formal verification.

Consequently, the degree of precision used in the definition of UML 2.0 was increased significantly. This was achieved by the following means:

- *A major refactoring of the metamodel infrastructure. UML 2.0's "infrastructure" comprises a set of low-level modeling concepts and patterns that are in most cases too rudimentary or too abstract to use directly in modeling software applications. However, their relative simplicity makes it easier to be precise about their semantics and their corresponding well-formedness rules. These finer-grained concepts are then combined in different ways to produce 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 you can categorize according to their features), were all inextricably bound into one semantically complex notion. (Note that this also meant that you could not use any one of these without implying the other two.) In the UML 2.0 infrastructure, these concepts were separated and their syntax and semantics defined separately.*
- *Extended and more precise semantics descriptions. The semantics definition 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), while others had little or no explanations. The UML 2.0 specification puts more emphasis on the semantics and, in particular, in the key area of basic behavioral dynamics (see below)[6].*

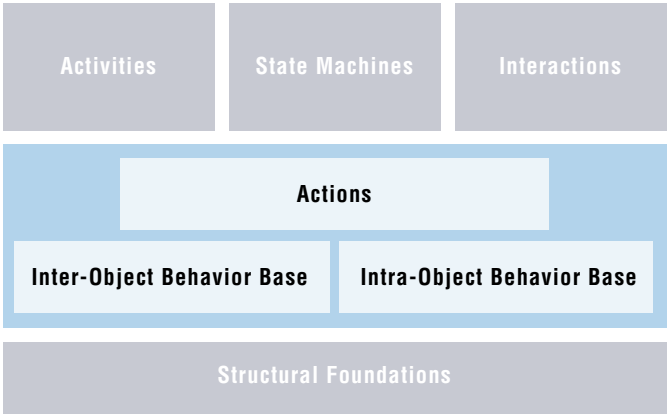
6. For a more detailed discussion of the semantics of UML 2.0, refer to [bs1].

- *A clearly defined dynamic semantic framework. The UML 2.0 specification clarifies some of the critical semantic gaps in the original version. This framework is depicted in Figure 1 and is described in more detail in [Selic04].*

  In particular, this framework addresses explicitly the following issues:

  - *The structural semantics of links and instances at runtime*
  - *The relationship between structure and behavior*
  - *The semantic underpinnings or causality model shared by all current high-level behavioral formalisms in UML (i.e., state machines, activities, interactions) as well as potential future ones. This also ensures that objects whose behaviors are expressed using different formalisms can interact with each other.*

Figure 1. The UML 2.0 semantics framework

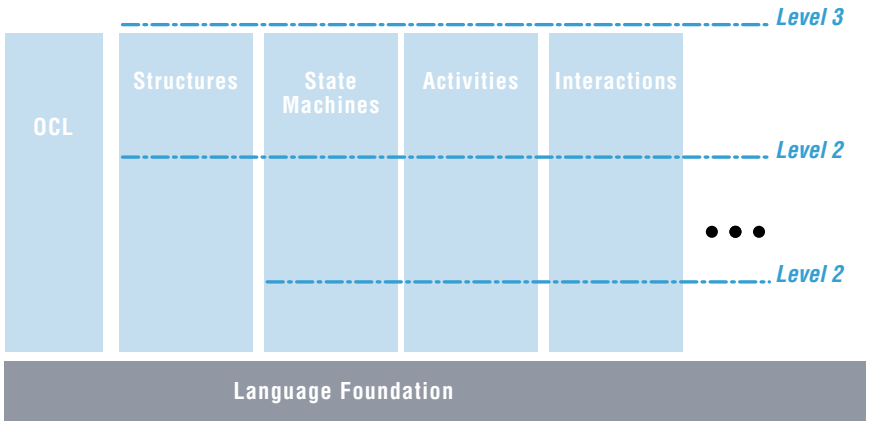


New language architecture

One immediate consequence of UML 2.0's increased level of precision is that the language definition has grown—even without accounting for the new modeling capabilities. This is a concern, especially given that the industry criticized the original UML for being too rich and, therefore, too cumbersome to learn and use. However, 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 not become simpler over time; 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 using basic hand tools.)

To deal with the language-complexity problem, the OMG modularized UML 2.0 in a way that allows developers to selectively use language modules. Figure 2 shows the general form of this structure. It consists of a foundation comprising shared 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 I). These vertical language units are generally independent of each other; therefore, you can use them independently. (Note that this was not the case in UML 1, where, for example, the activities formalism was based entirely on the state machine formalism.)

Figure 2. The language architecture of UML 2.0



Further, the vertical language units are hierarchically organized into as many as 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, you can only use specific subsets.

This architecture means that users can learn and use only the UML subset 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 to learn all of English to use it effectively. As you gain experience, you have the option of gradually introducing more powerful modeling concepts as necessary.

Table 1. The language units of UML 2.0

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	Informal behavioral requirements modeling

As part of the same architectural reorganization, the definition and structure of compliance has been significantly simplified in UML 2.0. In UML 1, the basic units of compliance were defined by the metamodel packages, with literally hundreds of possible combinations[7]. This meant that it was highly unlikely to find two or more modeling tools that could interchange models, since each would likely support a different package combination.

In UML 2.0, only three levels of compliance are defined and those correspond to the hierarchical language unit levels already mentioned and depicted in Figure 2[8]. These are defined in such a way that models at level (n) are compliant with models at any of the higher levels (n+1, etc.). That is, a tool compliant to a given level can import models, without loss of information, from tools that were compliant to any level equal to or below its own.

Four types of compliance are defined:

- *Compliance to the abstract syntax*
- *Compliance to the concrete syntax (i.e., the UML notation)*
- *Compliance to both abstract and concrete syntax*
- *Compliance to both the abstract and concrete syntax and the diagram interchange standard [OMG03b]*

This means that there is a maximum of only 12 different compliance combinations with clear dependency relationships between them (e.g., abstract and concrete syntax compliance is compatible with only concrete syntax compliance or only abstract syntax compliance). Consequently, in UML 2.0, model interchange between compliant tools from multiple vendors becomes more than just a theoretical possibility.

7. In fact, because UML 1 formalized the notion of "incomplete" compliance to a given compliance point, the possible number of different capability combinations that allowed a vendor to claim compliance was orders of magnitude greater.

8. Formally, UML 2 also defines a fourth level (Level 0), but this is an internal level intended primarily for tool implementers.

Large-scale system modeling capabilities

Relatively few features were added to UML 2.0. This was intentional to avoid the infamous "second system" effect [Brooks95], whereby a language gets bloated by an excess of new features demanded by a highly diverse user community. In fact, the majority of new modeling capabilities are, in essence, simply extensions of existing features that allow you to use them to model large-scale software systems. Moreover, these extensions were all achieved using the same basic approach: recursive application of the same basic set of concepts at different levels of abstraction. This means that you could combine model elements of a given type into units that, in turn, you would use as the building blocks for the next level of abstraction and so on; this is analogous to the way that you could nest procedures in programming languages within other procedures to any desired depth.

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

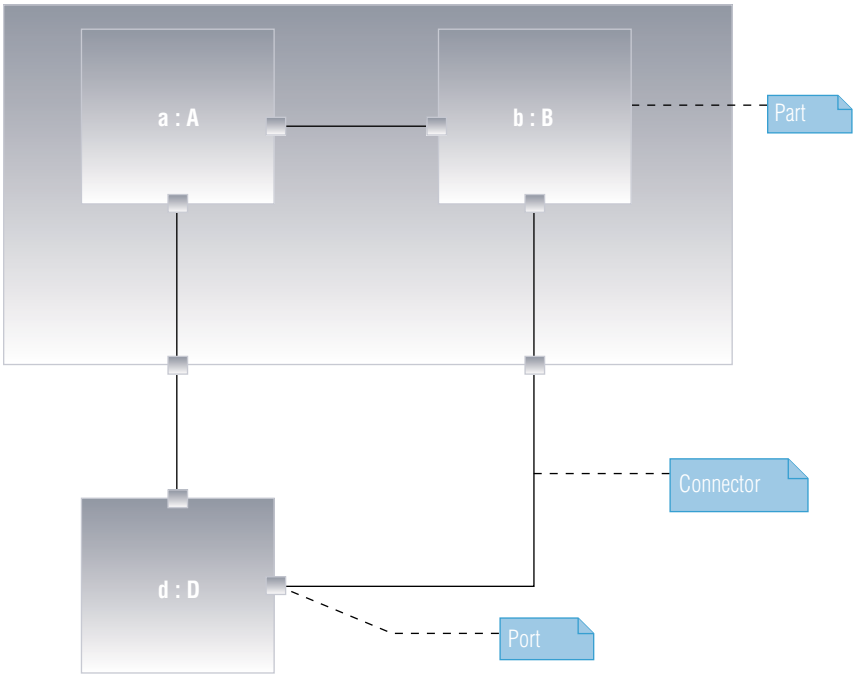
- *Complex structures*
- *Activities*
- *Interactions*
- *State machines*

The first three of these account for more than 90 percent of UML 2.0's new features.

Complex structures

The basis for this set of features comes from long-term experience with various architectural description languages, such as UML-RT [SR98], Acme [GMW97], and SDL (Systems Description Language) [ITU02]. These languages are characterized by a relatively simple set of graph-like concepts: basic structural nodes called "parts" that may have one or more ports and which are interconnected via communication channels called connectors. You may encapsulate these aggregates within higher-level units that include their own ports; therefore, you can combine them with other higher-level units into yet even-higher-level units, and so on.

Figure 3. Complex structure modeling concepts



To a degree, you could already find these concepts in the UML 1 definition of collaborations, except that they were not applied recursively. To allow recursion, you nest a collaboration structure within a class specification, which means that all instances of that class will have an internal structure that the class definition specifies. For example, in Figure 3, parts a:A and b:B are nested within part c:C, which represents an instance of the composite structure class C. Other instances of that class would have the same structural pattern, including all the port, parts, and interconnections.

With these three simple concepts and their recursive application, you can model arbitrarily complex software architectures.

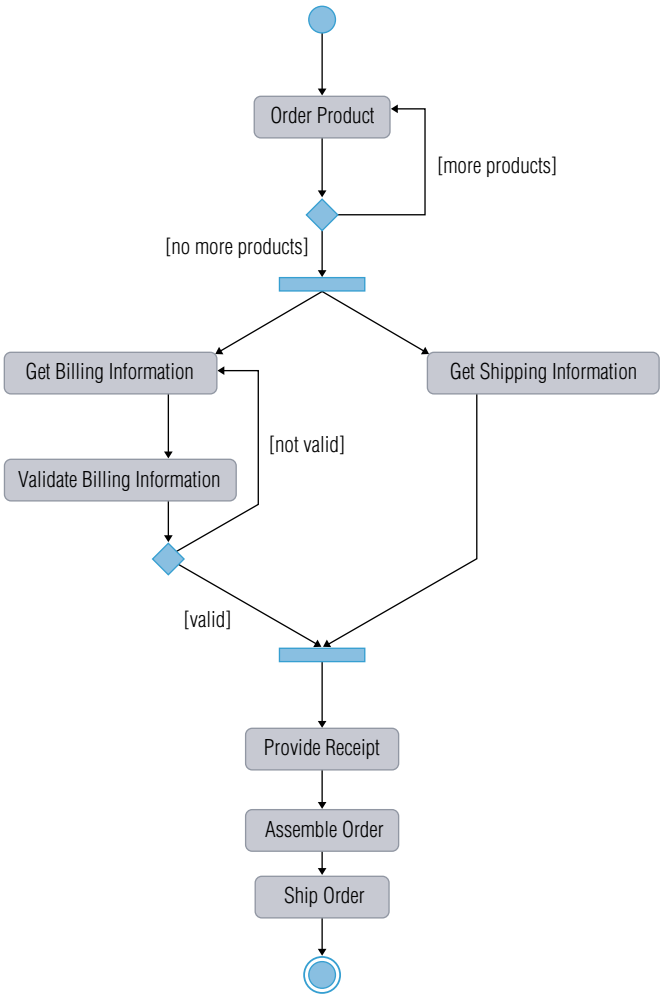
Activities

You use activities in UML to model flows of various kinds: signal/or data flows as well as algorithmic/procedural flows (see Figure 4). Needless to say, there are numerous domains and applications that are most naturally rendered by such flow-based descriptions. In particular, business-process modelers and also systems engineers (who tend to view their systems primarily as flow-through signal processors) embraced this formalism. Unfortunately, the UML 1 version of activity modeling had several serious limitations in the types of flows that it could represent. 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.

9. In fact, the semantic foundations are represented by a variant of generalized colored Petri nets [pet].

UML 2.0 replaced the state machine underpinning with a much more general semantic base[9] that eliminated all of these restrictions. In addition, inspired by several industry-standard business-processing formalisms, including notably BPEL4WS [BPEL03], a 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 rich modeling toolset that can represent a wide variety of flow types.

Figure 4. Activity modeling—purchasing a product(s)



As with complex structures, you can recursively group activities and their interconnection flows into higher-level activities with clearly defined inputs and outputs. In turn, you can combine these 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. Without the ability to package such repeated sequences into separate units, you had to define them numerous times, adding not only overhead but also complicating model maintenance (e.g., when you needed to change the sequence).*
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; concurrent and order-independent execution; and so forth.*

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 [ITU04]. This formalism was used as a basis for representing interactions in UML 2.0.

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

You can invoke these "packaged" interactions recursively from within higher-level interactions analogous to macro invocations (Figure 5). As you might expect, you can nest these to an arbitrary degree. Further, interactions can serve as operands in complex control constructs, such as loops (for example, you may have to repeat a given interaction several times) and alternatives. UML 2.0 defines several convenient modeling constructs of this type, providing a rich facility for modeling complex end-to-end behavior at any level of decomposition.

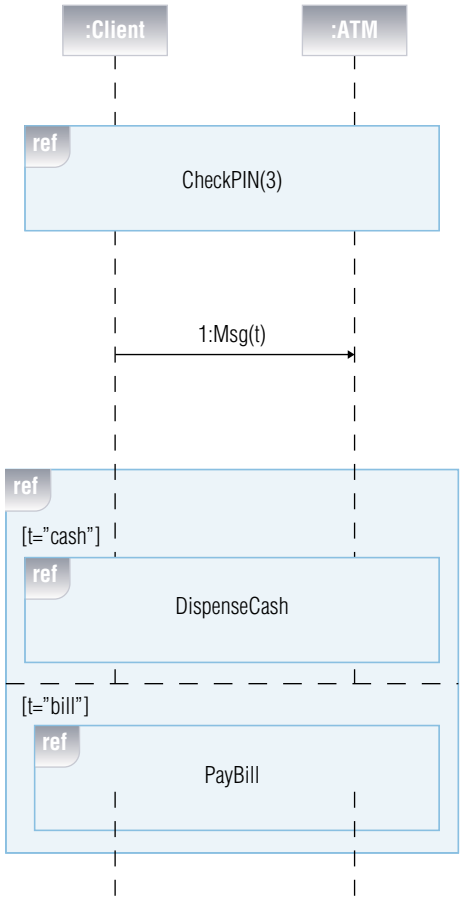


Figure 5. An example of an interaction model—ATMAccess.

Figure 5 shows an example of an extended interaction model. In this case, the interaction ATMAccess first "invokes" another lower-level transaction called CheckPIN (the diagram does not show this interaction's contents). Note that the latter interaction has a parameter (in this case, say, the number of times a user can enter an invalid PIN before the transaction is cancelled). After that, the client sends an asynchronous message specifying what kind of interaction it requires and, based on the value specified, it performs either the DispenseCash interaction or the PayBill interaction.

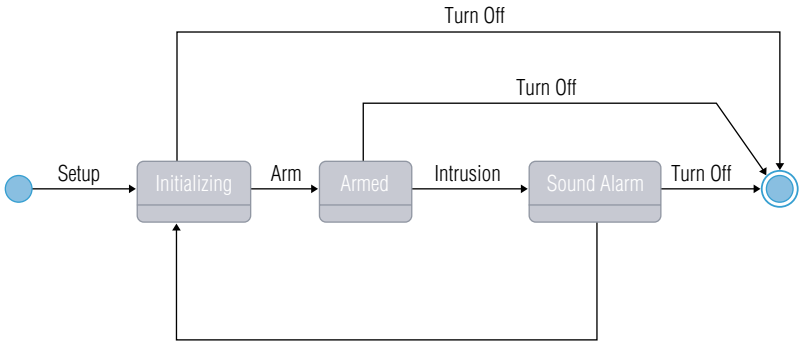
You can represent interactions in UML 2.0 by sequence diagrams as shown in the example above as well as by other diagram types, including the collaboration-based form defined in UML 1. There is even a non-graphical tabular representation.

State machines

The main new capability added to state machines in UML 2.0 is quite similar to the previous cases. The basic idea is that you can make a composite state fully modular with explicit points of transition entry and transition exit. This, in turn, allows you to define the internal decomposition of that state separately with a reusable state machine specification. That is, you can reuse the same specification in multiple places within the state machine or some other state machines. This simplifies the specification of shared behavior patterns in different contexts.

One other notable state machine innovation in UML 2.0 is a clarification of state machine inheritance between a class and its subclasses. See Figure 6 for a simple state machine model.

Figure6. State machine diagram for a simple burglar alarm



Language specialization capabilities

Experience with UML 1 indicated that a very common way of applying UML was to first define a UML profile for a particular problem or domain and then to use that profile instead of or in addition to general 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 a language definition that is optimally suited to the problem at hand. At first glance, this may seem the preferred approach to a DSL definition, but closer scrutiny reveals that there can be serious drawbacks to it.

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 will need to define the concepts of classes, attributes, associations, interactions, etc. 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.

But there is more than just conceptual reuse at play here. Because a UML profile, by definition, has to be compatible with standard UML[10]: (1) you can use any tool that supports standard UML to manipulate models based on that profile and (2) directly apply any knowledge of and experience with standard UML. Therefore, you can mitigate many of the fragmentation problems stemming from diversity or even avoid them altogether. This type of reasoning led the international standards body responsible for the SDL language [ITU02]—a DSL widely used in telecommunications—to redefine SDL as a UML profile [ITU00] [ITU03].

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

10. A UML profile can only specialize in the standard UML concepts by defining constraints on those concepts that gives them a unique domain-specific interpretation. For example, a constraint may disallow multiple inheritance or it may require that a class must have a particular type of attribute.

With these considerations in mind, the profiling mechanism in UML 2.0 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 2.0 stereotype is defined as if it was 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 2.0 profile can also explicitly hide UML concepts that make no sense or are unnecessary in a given DSL. This allows you to define minimal DSL profiles.

Finally, you can also use the UML 2.0 profiling mechanism to viewa complex UML model from multiple, different domain-specific perspectives—something not generally possible with DSLs. That is, you can selectively "apply" or "de-apply" any profile 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 the model's elements. An automated performance analysis tool can then use these to determine a software design's fundamental performance properties. 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.

General consolidation

This item covers several areas, including the removal of overlapping concepts as well as numerous editorial modifications, such as adding clarifications to confusing descriptions and the standardization of terminology and specification formats.

The removal of overlapping concepts and the clarification of poorly defined concepts were two other important requirements for UML 2.0. The three major areas affected by this requirement were actions and activities, templates, and component-based design concepts.

Actions were introduced in UML 1.5. The conceptual model of actions was intentionally made general enough to accommodate both data-flow and control-flow computing models. This resulted in a significant conceptual similarity to the activities model. UML 2.0 exploits this similarity to provide a common syntactic and semantic foundation for actions and activities. From the user's point of view, these are formalisms that occur at different abstraction levels since they typically model phenomena at different granularity levels. However, the shared conceptual base results in overall simplification and greater clarity.

In UML 1, templates were defined very generally: you could make any UML concept into a template. Unfortunately, this generality impeded its application since it allowed for potentially meaningless template types and template substitutions. UML 2.0's template mechanism 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. You could use classes, components, or subsystems. These concepts had a lot 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 2.0, all 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 you could decide when to use which concept on the basis of objective criteria.

On the editorial side, the specification format was consolidated with the semantics and notation specifications for the modeling concepts combined for easier reference. Each metaclass specification was expanded with information that explicitly identifies semantic variation points, notational options, as well as its relationship to the UML 1 specification. Also, the terminology was 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.

Summary

UML 2.0 was designed to allow a gradual introduction of model-driven methods. You can still use it in the same informal way as UML 1 if you prefer it as a "sketching" tool. Moreover, since the new modeling capabilities are non-intrusive, , in most cases, you will not see any change in the language's look and feel.

However, the opportunity to move forward on the MDD scale is now available and standardized. The increased precision is also available for you to use, if desire, all the way through to completely automated code generation.

The standards body carefully reorganized the language structure to allow a modular and graduated approach to adoption: users only need to learn t he parts of the language that are of interest to them and can safely ignore the rest. As your experience and knowledge increases, you can selectively add new capabilities. Along with this reorganization, the definition of compliance to facilitate interoperability between complementary tools as well as between tools from different vendors is greatly simplified.

Only a small number of new features were added to avoid language bloat, and practically all of those are designed along the same recursive principle that enables modeling of 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 UML-based domain-specific languages. These languages have the distinct advantage that they 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 will help us develop more sophisticated software systems faster and more reliably—but still using the same type of intuition and expertise that is every software developer's bread and butter. In essence, it is still program design, only at a higher level—comparable to the step that occurred in hardware design when discrete components gave way to large-scale integration.

For more information

The UML diagrams in this whitepaper were created using IBM® Rational® Software Architect. Rational Software Architect is a design-and-construction tool for software architects and senior developers creating applications for the Java platform or in C++ that leverages model-driven development withthe UML and unifies all aspects of software application architecture. For a free trial version and additional information, visit our Rational Software Architect Webpage.

You can find information on the full portfolio of IBM Rational's modeling tools at our Design and Construction site.
http://www-306.ibm.com/software/rational/offerings/design.html

Learn more about Model-Driven Architecture® at our MDA® site.
http://www-306.ibm.com/software/rational/mda/

To find information on the UML, visit our UML Resource Center.
http://www-306.ibm.com/software/rational/uml/

For additional information visit the IBM Rational homepage.
http://www-306.ibm.com/software/rational/

References

[BPEL03] BEA, et al., Business Process Execution Language for Web Services (Version 1.1), 5 May 2003, (ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf).

[Brooks95] Brooks Jr., F., The Mythical Man-Month (1995 edition), Addison-Wesley, 1995.

[Brown04] Brown, A., "An Introduction to Model-Driven Architecture," IBM Rational developerWorks (http://www-106.ibm.com/developerworks/rational/library/3100.html), 2004.

[Fowler04] Fowler, M., UML Distilled (3rd edition), Addison-Wesley, 2004.

[GMW97] Garlan, D., Monroe, R., and Wile, D., "Acme: An Architecture Description Interchange Language," in Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research, Association for Computing Machinery (ACM), 1997.

[Graham01] Graham, I., Object-Oriented Methods: Principles and Practice (3rd edition), Addison-Wesley, 2001.

[ITU00] International Telecommunications Union, ITU Recommendation Z.109: SDL Combined with UML, ITU-T, 2000.

[ITU02] International Telecommunications Union, ITU Recommendation Z.100: Specification and Description Language (SDL), (08/02), ITU-T, 2002.

[ITU04] International Telecommunications Union, ITU Recommendation Z.120: Message Sequence Chart (MSC), (04/04), ITU-T, 2004.

[ITU05] International Telecommunications Union, "Study Group 17: Question 13/17—System Design Languages Framework and Unified Modeling Language," ITU-T Study Group 17, (http://www.itu.int/ITU-T/studygroups/com17/sg17-q13.html), 2003.

[Lee92] Lee, L. The Day the Phones Stopped Ringing, Plume Publishing, 1992.

[Booch04] Booch, G., et al., "An MDA Manifesto," with Frankel, D., and Parodi, J. (eds.), The MDA Journal, Meghan-Kiffer Press, 2004.

[OMG03a] Object Management Group, Unified Modeling Language (UML), Version 1.5, OMG document formal/03-03-01 (http://www.omg.org/cgi-bin/doc?formal/03-03-01), 2003.

[OMG03b] Object Management Group, UML 2.0 Diagram Interchange, Final Adopted Specification, OMG document ptc/03-09-01 (http://www.omg.org/cgi-bin/apps/doc?ptc/03-09-01.pdf), 2004.

[OMG04] Object Management Group, UML 2.0 Superstructure, Available Specification, OMG document ptc/04-10-02 (http://www.omg.org/cgi-bin/apps/doc?ptc/04-10-02.zip), 2004.

[RJB05] Rumbaugh, J., Jacobson, I., and Booch, G., The Unified Modeling Language Reference Manual (2nd edition), Addison-Wesley, 2005.

[Stevens02] Stevens, P., "On the Interpretation of Binary Associations in the Unified Modeling Language," Journal of Software and Systems Modeling, vol.1, no.1, Springer-Verlag, September 2002.

[Selic04] Selic, B., "On the Semantic Foundations of Standard UML 2.0," with Bernardo, M., and Corradini, F. (eds.), Formal Methods for the Design of Real-Time Systems, Lecture Notes in Computer Science vol. 3185, Springer-Verlag, 2004.

[SR98] Selic, B. and Rumbaugh, J. "Using UML for Modeling Complex Real-Time Systems," unpublished whitepaper (http://www.rational.com/media/whitepapers/umlrt.pdf), Apr. 4, 1998.

(Footnotes)

1 In addition, some not-so-valid reasons, such as general human distrust of innovation.

2 In tune with the times, the original UML standard was primarily designed to serve as an auxiliary tool for informal capture and communication of design intent.

3 However, we still lack a consolidated and systematic theory of modeling language design that is comparable to the current theory of programming language design.

4 This subset of UML, primarily comprising concepts defined in UML class diagrams is called the Meta-Object Facility (MOF). This subset was chosen such that you could use it to define other modeling languages.

5 It is called "abstract" because it is independent of the actual notation or "concrete syntax" (e.g., text, graphics) that is used to represent models.

6 For a more detailed discussion of the semantics of UML 2.0, refer to [bs1].

7 In fact, because UML 1 formalized the notion of "incomplete" compliance to a given compliance point, the possible number of different capability combinations that allowed a vendor to claim compliance was orders of magnitude greater.

8 Formally, UML 2 also defines a fourth level (Level 0), but this is an internal level intended primarily for tool implementers.

9 In fact, the semantic foundations are represented by a variant of generalized colored Petri nets [pet].

10 A UML profile can only specialize in the standard UML concepts by defining constraints on those concepts that gives them a unique domain-specific interpretation. For example, a constraint may disallow multiple inheritance or it may require that a class must have a particular type of attribute.

IBM Corporation
Software Group
Route 100
Somers, NY 10589

Produced in the United States of America
05-05

G507-0992-00