What Models Mean

Ed Seidewitz, *InteliData Technologies*

If today's software developers use models at all, they use them mostly as simple sketches of design ideas, often discarding them once they've written the code. This is sufficient for traditional code-centric development. With a model-driven approach, however, the models themselves become the primary artifacts in the development of software. In this case, a clear, common understanding of the semantics of our modeling languages is at least as important as a clear, common understanding of the semantics of our programming languages.

There has been, and continues to be, a great deal of discussion within the software community on modeling and metamodeling and the relationships between modeling languages and metamodeling languages. Such relationships' circular nature makes them particularly hard to discuss clearly, often hiding many important but subtle issues at the foundation of what we do when we model.

To address these issues as clearly as possible, I examine a set of questions whose answers, taken together, will help us understand the fundamental question, "What do models mean?" The result is a set of careful definitions for key terms that will let us cut through some of this topic's more confusing aspects.

To understand the intent behind these definitions, we must also look at how other disciplines use models. After all, we should realize that, with a true model-driven approach to software development, we are using models in much the same way that other scientific and engineering disciplines use them. So, I've illustrated the definitions with simple (nontechnical) discussions of modeling in Newtonian physics and its application to engineering.

The article's primary goal is, however, to discuss the modeling of software. Since the Object Management Group first specified the Unified Modeling Language in 1997, UML has become the common modeling language of the software development community. (UML's current version is 1.5;[1] the draft specifications for Version 2.0 are entering the OMG "finalization" process.[2,3]) Thus, any discussion of software models must employ UML terms, as I do in this article.

Nevertheless, we begin first not with an in-

A model's meaning has two aspects: the model's relationship to what's being modeled and to other models derivable from it. Carefully considering both aspects can help us understand how to use models to reason about the systems we build and how to use metamodels to specify languages for expressing models.

0740-7459/03/$17.00 © 2003 IEEE

quiry into the modeling language but with an even more basic question.

What is a model?

A *model* is a set of statements about some *system under study*. Here, *statement* means some expression about the SUS that can be considered true or false (although no truth value has to necessarily be assigned at any particular point in time).

We can use a model to describe an SUS. In this case, we consider the model *correct* if all its statements are true for the SUS.

Models are usually descriptive in traditional scientific disciplines. A common kind of model in Newtonian physics might describe a set of physical objects—say the planets in the solar system, which would be the SUS in this case. Such a model makes statements on the positions, velocities, and masses of the planets at some point in time as they orbit the sun. The model is correct if those statements correspond to observations of the actual planets.

We can similarly use a UML model to describe an existing software system's structure and operation. Suppose that the SUS is an object-oriented software system. We could use a UML class model to make statements about the system's classes and how they're related. Additional models could make statements on how we expect instances of those classes to interact, resulting in state changes over some period. These models would be correct if the actual system's structure and behavior are consistent with those described in the models.

Alternatively, we can use a model as a *specification* for an SUS or a class of SUS. In this case, we consider a specific SUS *valid* relative to this specification if no statement in the model is false for the SUS.

Models usually serve as specifications in traditional engineering disciplines. For example, the physical objects in a Newtonian model could be spacecraft rather than planets. In this case, the model might specify beforehand what trajectories the spacecraft should take, rather than describe their trajectories after the fact. If the spacecraft deviate from the intended trajectories, the spacecrafts' design and operation, not the model, are likely invalid.

When we construct software, we also generally use models as specifications. So, the same UML models we used descriptively might also serve as a design specification for the OO software system. In this case, if the as-built software's structure or behavior deviates from the design specification, it's the software that's invalid (at least relative to this specification).

How is a model interpreted?

An *interpretation* of a model is a mapping of the model's elements to elements of the SUS such that we can determine the truth value of statements in the model from the SUS, to some level of accuracy. Colloquially, an interpretation of a model gives the model meaning relative to the SUS. If this mapping is invertible, so that we can map the SUS elements to model elements, then we can also construct a model as a *representation* of an SUS, such that all the statements in the representation are true for the SUS under the interpretation mapping.

An interpretation of a model gives a model meaning.

Newtonian physics represents a particle's position and velocity in 3D space as vectors of three numbers. The interpretation of, say, the position vector, is that some element of the universe identified as a particle will be found at the x, y, z Cartesian coordinates given by the vector's three numbers, relative to some origin point. Such an interpretation is a common, generally accepted convention among scientists and engineers who use Newtonian models (although it is not the only convention—you could instead use r, θ, ϕ polar coordinates, for example).

The interpretation of, say, a UML class model is not as conventionally fixed as in the case of Newtonian models. Instead, the interpretation might be given by an accepted community *profile* for using UML, or it might be simply the consequence of (perhaps implicit) local design conventions. For example, a UML profile for Java might interpret a UML class model as specifying that there are Java classes corresponding to the UML classes and that references between instances of those Java classes implement associations in the class model. On the other hand, a UML profile for business modeling would have a very different interpretation of a class model, in terms of business entities such as workers and resources.

How is a model used?

A *theory* is a way to deduce new statements about an SUS from the statements already in some model of the SUS. We can consider this as a way to extend the original model or as a

Given a model that conforms to a theory, we can make deductions.

way to determine whether the model *conforms* to the theory. In the latter case, any statements deduced from some subset of the model must be *consistent* with any other statements in the model. Two statements in a model are consistent if they can both be true for the SUS (that is, the truth of one statement does not necessarily imply the falsity of the other).

Given a model that conforms to a theory, we can use an interpretation of the model to make deductions about an SUS by making deductions within the model using the theory. If we're modeling descriptively, we consider a theory correct if deductions made about the SUS using the theory correspond to what is actually observed of the SUS (to some level of accuracy). On the other hand, for a specification model, we assume that the theory is correct, so that all statements deducible from the specification also effectively become part of the specification. That is, not only can no statement in the specification model itself be false for a valid SUS, but also no statements deducible from the specification can be false.

For example, given the positions, velocities, and masses of two particles at a particular point in time, we can use Newton's theory of gravity to deduce the particles' trajectories over time under the influence of their mutual gravitational attraction. Alternatively, given a model of the two particles' trajectories, we can determine whether the trajectories are consistent with the Newtonian theory of gravity. These models predict how actual particles will move. Newtonian theory is correct up to the level of accuracy at which relativistic effects become significant.

Similarly, given a UML class model of an OO system and the system's initial state, the "theory of UML class modeling," such as it is given by UML semantics, lets us deduce how instances of those classes will interact and how their states will evolve under a given set of external stimuli. Also given a UML interaction model for the system, such a theory lets us deduce whether this model is consistent with the class model. We could use these models to predict the operation of the software system (objects as instances of classes). For descriptive models, UML semantics are correct to the extent that, for example, hardware limitations and failures can be ignored. For specification models, the software system is valid if it doesn't violate any prediction.

How is a model expressed?

A *modeling language* lets us express the statements in models of some class of SUS.

Because computer science has traditionally focused so much on languages, the concept of a modeling language sometimes receives more prominence than the concept of a model itself. This is not usually so in other scientific and engineering disciplines. Nevertheless, the languages used for modeling in those disciplines are just as important as the languages used in software modeling.

In Newtonian physics, vector calculus is the primary modeling language. Although you don't find books on "Newton's Modeling Language," you can certainly find textbooks on vector calculus, even ones targeted at practicing scientists and engineers rather than mathematicians. Indeed, Newton more or less had to invent differential calculus to first present his theory.

For software modeling, UML has become the de facto modeling language. In fact, the class of SUS covered by UML has grown beyond just software systems. UML now covers any SUS for which it's useful to make statements about the data maintained or the behavior the system exhibits relative to its environment—including such things as businesses and even hardware.

What is a metamodel?

A *metamodel* is a specification model for a class of SUS where each SUS in the class is itself a valid model expressed in a certain modeling language. That is, a metamodel makes statements about what can be expressed in the valid models of a certain modeling language.

Computer science's language focus leads to the concept of a metamodel being much more prominent in software modeling than in other modeling disciplines. It does, however, appear in other disciplines, implicitly or explicitly (although the term *metalanguage* is more common than *metamodel* in, say, mathematical logic). For example, mathematics provides the metamodel for vector calculus, and hence for Newtonian physics, usually as expressed in various mathematical textbooks.

The metamodel concept for software modeling is also particularly important because it forms the basis for the UML definition. The UML specification document is indeed a metamodel for UML. That is, it includes a set of statements that must not be false for any valid

UML model. (This metamodel, in its entirety, includes all the concrete graphical notation, abstract syntax, and semantics for UML.)

How is a metamodel interpreted?

Because a metamodel is a model of a modeling language, an interpretation of a metamodel is a mapping of the metamodel's elements to the modeling language's elements, such that we can determine the truth value of statements in the metamodel for any model expressed in the modeling language. Because a metamodel is a specification, a model in the modeling language is valid only if none of these statements are false.

If the interpretation mapping for a metamodel is invertible, we can also uniquely map elements of the modeling language back to elements of the metamodeling language. In this case, given any model, we can invert the interpretation mapping to create a metamodel representation of the model—that is, a set of true statements about the model in the metamodeling language.

To continue the example of mathematics as the metamodel for vector calculus, textbook definitions provide the mapping of general mathematical concepts to the elements of vector calculus. A statement in vector calculus is valid if it is *well formed* relative to these definitions. (Interestingly enough, the more formal these definitions, the less meaning they actually impart—indeed, the term *formal* indicates the importance of form over meaning.)

The *abstract syntax* metamodel for UML is basically a data model of how to store or interchange data on a UML model's syntactic structure. We interpret this metamodel by mapping instances of the *metaclasses* in the metamodel (where both the metaclasses and their instances are expressed in the metamodeling language) to the syntactic elements of the UML graphical notation (that is, elements of the modeling language). This is an invertible mapping, so that we can also map syntactic elements of the UML graphical notation back to their representations as instances of metaclasses (which is indeed how the mapping is usually specified).

How is a metamodel used?

A theory of a metamodel is a way to deduce new statements about a modeling language from the statements already in the modeling language's metamodel. Because a metamodel is a specification, a valid model in the modeling language must not violate any statement deducible using the theory from the explicit metamodel statements.

One way to look at this is to consider the metamodel's statements as *postulates* about the modeling language. Then, given the metamodel representation of a model, we can determine, using the theory, whether the model's representation is consistent with the metamodel (in the sense defined in the earlier discussion of *theory*). If it's consistent, the model is valid; otherwise, the model isn't.

For example, mathematical theories are rigorous systems for making deductions about mathematical statements. In formal mathematics, the deduction system is actually more important than any meaning the statements might have.

In the UML metamodel, the abstract syntax constrains the allowable structure of and relationships between model elements represented as instances of metaclasses. This is the theory of the syntactic metamodel. If the instances in the metamodel representation of a UML model meet these constraints, the UML model is well formed; otherwise it isn't.

It's common mental shorthand to identify a UML model element directly with its abstract-syntax metamodel representation (for example, the class `X` with its representation as an instance of the metaclass `Class`) and to loosely refer to the model element as being directly "an instance of" the metaclass (for example, class `X` "is an instance of" the metaclass `Class`). However, strictly speaking, the concept of "instance of" has meaning only within the theory of the metamodeling language, not between a metamodeling-language element (the metaclass `Class`) and a modeling-language element (the class `X`).

How is a metamodel expressed?

Because a metamodel is a model, we express it in some modeling language. A single modeling language might have more than one metamodel, with each expressed in a different modeling language.

Of particular interest for our purposes is when a modeling language's metamodel uses that same modeling language. That is, the statements in the metamodel are expressed in the same language the metamodel is describing. We call this a *reflexive metamodel*.

Given its metamodel representation, we can determine whether a model is valid.

Interpretation crosses metalayers; a theory resides in a single metalayer.

A *minimal* reflexive metamodel uses the minimum number of elements of the modeling language (for the purposes of that metamodel). That is, every statement about the modeling language that must be stated in the metamodel (for its purpose) can be expressed using this minimal set of modeling elements. But, if any element were removed from the set, some essential statements couldn't be expressed.

As Kurt Gödel showed, we can use statements of number theory to make statements about number theory itself (for example, "This proof is false"), given an appropriate encoding of number-theoretic statements as numbers.[4] This allows number theory to express a reflexive metamodel for itself. Because, in principle, we can reduce all mathematical statements to set theory, and mathematical statements can themselves be expressed using mathematics, set theory (in principle) provides a minimal reflexive metamodel for all of mathematics.

In a similar vein, we can use UML's object-modeling features to make statements about its abstract syntax, given an appropriate representation of UML's surface notation as object structures. This abstract-syntax representation is, essentially, the abstract parse tree for any concrete surface notation. The abstract syntax's metamodel elements are then classes, and the interpretation mapping identifies model elements with instances of those classes. We can express the entire UML abstract syntax using a minimal subset of UML's static-structure modeling constructs (for UML 1.5, this subset includes classes, packages, associations, generalizations, and dependencies). Other metamodeling languages used in the UML specification are the Object Constraint Language and English.

OMG has defined metamodels for languages other than UML, such as its Common Warehouse Metamodel,[5] used for modeling data warehouses. The common basis for all OMG metamodels is the OMG Meta-Object Facility. As the framework used for defining specific modeling languages, the MOF specification adopts a four-layer metamodeling architecture:[6]

- M0—What is to be modeled
- M1—Models (for example, a UML model)
- M2—Metamodels (for example, an abstract-syntax model in the UML specification)
- M3—The meta-metamodel

Layer M3 is a specification of the modeling language used to express metamodels in M2. In OMG, this is always the MOF metamodeling language. This metamodeling language is reflexively specified, so no higher layers are needed.

In terms of the OMG metamodeling layers, interpretation crosses metalayers. For example, the interpretation mapping for UML maps from the model elements, which are in M1, to the SUS elements, which are in M0. Similarly, there are interpretation mappings from metamodel elements in M2 to model elements in M1 and from meta-metamodel elements in M3 to metamodel elements in M2.

On the other hand, a theory resides in a single metalayer. For example, a theory of UML allows some models to be deduced from other models (for example, instance models from class models), entirely in M1. Similarly, a theory of the UML abstract syntax allows a UML model's validity to be determined entirely in M2, after the model is mapped to its metamodel representation.

How is a reflexive metamodel interpreted?

Because a reflexive metamodel is expressed in the same modeling language it describes, its interpretation provides a mapping of the modeling language onto itself. Generally, this mapping will be from the entire modeling language to a subset. We can then iterate this mapping, each time producing a smaller subset, until we reach the minimal reflexive metamodel that maps completely onto itself rather than a subset.

An interpretation of a minimal reflexive metamodel maps the metamodel onto itself. This means that any statement in the minimal reflexive metamodel can be represented in terms of elements of the minimal reflexive metamodel. However, the interpretation of this representation is itself expressed reflexively as a mapping to yet another representation in terms of the minimal reflexive metamodel. This circularity means that, for a minimal reflexive metamodel, the interpretation mapping really provides no useful expression of the metamodel's meaning.

The problem here is that, even though the interpretation mapping maps the set of modeling elements in a minimal reflexive metamodel into that same set, it does not map any given model into itself. Instead, on each iteration of the mapping, it generates increasingly compli-

cated representations of the original model. As I noted earlier, the fundamental thing we need is a theory that lets us use the metamodel representation of a model to determine whether the model is well formed. To break the circularity, however, this theory cannot depend on any reflexive interpretation mapping of the metamodel. Instead, it ultimately must be provided in terms of more basic concepts (perhaps via a formal mathematic semantics or an explicit set of axioms and deduction rules).

So, what *do* models mean?

A model's meaning has two aspects. The first is the model's relationship to the thing being modeled. This is *meaning* in the sense of "This class model means that the Java program must contain these classes." I've called this an *interpretation* of the model. Multiple interpretations of the same model can exist—for example, you could interpret a certain logical class model as the design for either a Java program or for a C++ program.

The second aspect is the model's relationship to other models derivable from it. This is *meaning* in the sense of "This class model means that instances of these classes are related in this way." I've called this a theory of the modeling language; this is often called the modeling language's "semantics" (although the definition of *interpretation* in this article is closer to how the term *semantics* is used in the study of natural languages). Once again, multiple theories for a single modeling language can exist—for example, one theory might provide the "standard execution semantics" for UML, while another is tailored to mirror Java semantics.

Both these aspects of meaning are important for the very reasons we do modeling.

If we're modeling descriptively, we'll generally create a model representation from a given SUS. The resulting model's interpretation is then exactly a description of the SUS. We create such a model so that we can analyze the SUS by reasoning about the model. To reason about the model, we need a theory; to relate the results of this reasoning back to the SUS, we need to use the interpretation mapping again.

If we're modeling as a specification, then we'll generally create one or more SUS intended to meet that specification. The specification's interpretation determines the constraints on how we may construct a valid SUS. We then want to be able to deduce from the specification the SUS's observable properties. This lets us test that the SUS is correctly built by observing whether it has these properties. To make these deductions, we need a theory; to relate the deduced properties back to the as-built SUS, we need to use the interpretation mapping again.

When we're metamodeling, the metamodel is a specification and the SUS is a modeling language. In the case of UML, we gain conceptual economy by using UML reflexively as the metamodeling language for its own abstract syntax. However, this reflexivity tends to lead, in many discussions, to a conflation of the two aspects of meaning, which should strictly be understood as distinct, even in reflexive use.

For example, the "instance of" relationship, which is formally defined only within the theory of the UML modeling language, tends to get identified with the interpretation mapping from metaclass instances to UML model elements. But simply providing the interpretation mapping (*meaning* in the first sense) doesn't provide a formal definition of what "instance of" means in the metamodel (*meaning* in the second sense). To provide an appropriate grounding for any deductions based on our metamodel, we must understand the meaning of "instance of" within the theory of our metamodeling language, independently of the interpretation mapping we happen to use between the metamodel and modeling elements.

This is, admittedly, a somewhat subtle, esoteric point. However, it's an example of the care we must take when discussing our modeling languages' underpinnings. Understanding the interplay between model and metamodel, and between theory and interpretation, is crucial as we create the foundations on which to build widespread model-driven development.

Such concerns are certainly not unique to our discipline, as I've tried to show. But they are, perhaps, even more important to us than to other communities, with the peculiar combination of mathematics, logic, and engineering that provides material for our field. And, of course, they remain for us areas of active research and inquiry, not yet resolved into commonly accepted tenets.

This article has attempted to take another step toward such a common foundational understanding. For, in the end, models are really just tools, and they mean what we need them to mean to build successful systems.

Understanding the interplay between model and metamodel, and between theory and interpretation, is crucial.

About the Author

Ed Seidewitz is the chief architect at InteliData Technologies Corporation, where he's responsible for the model-driven approach being used to architect and develop the company's InteliWorks line of banking software. He first became involved with development of object-oriented software systems and methodologies while working at NASA's Goddard Space Flight Center. He has used UML since the prestandardized 0.9 version and has been heavily involved in the OMG process subsequent to UML's adoption. He has a BS in aeronautics and astronautics and in computer science and engineering from the Massachusetts Institute of Technology. He's a member of the ACM. Contact him at InteliData Technologies, 11600 Sunrise Valley Dr., Reston, VA 20191; eseidewitz@intelidata.com.

Acknowledgments

I conceived this article while reflecting on UML 2.0 proposals and ongoing threads on OMG mailing lists. A particular inspiration was the "3C" proposal.[7] I greatly thank Ed Barkmeyer, Steve Cook, Andy Evans, William Frank, Dave Frankel, Steve Mellor, Joaquin Miller, Jim Rumbaugh, and Bran Selic for reviewing earlier versions and participating in lively email discussions of the article's topics. This enjoyable interaction culminated in the informal paper "What Do Models Mean?" made available to the OMG community.[8] Steve Mellor suggested I write a new version of the paper for this special issue. I thank Steve for his vote of confidence and for working with me to adapt the paper to a wider audience.

References

1. *OMG Unified Modeling Language Specification*, ver. 1.5, OMG document formal/03-03-01, Object Management Group, Mar. 2003; www.omg.org/technology/documents/formal/uml.htm.
2. *Unified Modeling Language: Infrastructure*, ver. 2.0, OMG document ad/03-03-01, Object Management Group, 2003.
3. *Unified Modeling Language: Superstructure*, ver. 2.0, OMG document ad/03-04-01, Object Management Group, 2003.
4. K. Gödel, *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*, Dover, 1962.
5. *Common Warehouse Metamodel (CWM) Specification*, ver. 1.1, OMG document ptc/02-01-07, Object Management Group, 4 Feb. 2002; www.omg.org/technology/documents/formal/cwm.htm.
6. *Meta Object Facility (MOF) Specification*, ver. 1.4, OMG document formal/02-04-03, Object Management Group, Apr. 2002, Section 2.2; www.omg.org/technology/documents/formal/mof.htm.
7. *OMG Unified Modeling Language Specification (revised submission), 3C: Clear, Clean, Concise*, ver. 2.0.13, OMG document ad/02-09-15, Object Management Group, 9 Sept. 2002.
8. E. Seidewitz, "What Do Models Mean?" OMG document ad/03-03-31, Object Management Group, Mar. 2003.