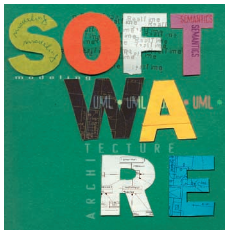
COVER FEATURE

Meaningful Modeling: What's the Semantics of "Semantics"?



Much confusion surrounds the proper definition of complex modeling languages, especially the Unified Modeling Language. At the root of the problem is insufficient regard for the crucial distinction between syntax and true semantics and a failure to adhere to the nature and purpose of each.

David Harel
Weizmann Institute of Science

Bernhard Rumpe
Technische Universität Braunschweig

The Unified Modeling Language (UML) is a complex collection of mostly diagrammatic notations for software modeling, and its standardization has prompted an animated discussion about UML's semantics and how to represent it. As the "Wrong Ways to View Semantics" sidebar describes, authors have quite different ideas of what constitutes semantics for UML subsets and adaptations. Worse, implicit assumptions often influence these definitions and results. Comparing published research on UML semantics is thus very difficult, since the comparison must take into account the subsets dealt with, the kind of systems assumed, the relationships among constructs, the definitions' detail level, and the notations and representations used.

Obviously, a multitude of concepts surround the proper definition of complex modeling languages, and many people are confused about what these concepts—both crucial and marginal—really mean and how to understand and use them. This confusion is particularly apparent in the context of UML, a multifaceted effort whose followers are ever growing, but it is also characteristic of other modeling approaches.

We have thus set out to clarify some of the notions involved in defining modeling languages, with an eye toward the particular difficulties arising in defining UML. We are primarily interested in distinguishing a language's notation, or *syntax,* from its meaning, or *semantics,* as well as recognizing the differences between variants of syntax and semantics in their nature, purpose, style, and use.

ELEMENTS OF A LANGUAGE DEFINITION

Much has been said about the distinction between the purist notion of information and its syntactic representation as data. The literature generally agrees that data is used to communicate, but extracting or understanding the information behind it requires an interpretation—a mapping that assigns a meaning to each (legal) piece of data.

A major source of confusion is the mixing of the data and information notions. In one case, two pieces of data might encode the same information, for example, "June 20, 2000" and "The last day of the first spring in the second millennium." In another case, the same piece of data might have several meanings and therefore denote different information for different people or applications. The information the reader derives from "John's birthday," for example, depends on the context. Deeply understanding the difference between syntax and semantics helps avoid confusion.

Just as people use natural languages to communicate with each other, machines use machine-readable languages for communication. Both kinds of language—whether they are natural, artificial, pro-

Published by the IEEE Computer Society

gramming, or hardware description languages—contain a great variety of meaningful language elements. Communication stakeholders must thus agree on the language, which in turn fixes the data set that they can communicate.

Accordingly, a language consists of a syntactic notation (syntax), which is a possibly infinite set of legal elements, together with the meaning of those elements, which is expressed by relating the syntax to a semantic domain. Thus, any language definition must consist of the syntax semantic domain and semantic mapping from the syntactic elements to the semantic domain.

Syntax

Depending on the language type, syntactic elements can be words, sentences, statements, boxes, diagrams, terms, models, clauses, modules, and so on. In our description and the "Two Language Examples" sidebar, we use "expression" to represent these terms.

Textual languages are symbolic in spirit, and their basic syntactic expressions are put together in linear character sequences. In contrast, the basic expressions in iconic languages are small pictorial signs that visually depict elements. An iconic language can be more intuitive than a textual language, but only if the designer resists abusing the icons.

In diagrammatic languages, or visual formalisms,[1] basic expressions include lines, arrows, closed curves and boxes, and composition mechanisms involve connectivity, partitioning, and "insideness." Despite some well-known critiques,[2,3] diagrammatic languages are proving extremely helpful in software and systems development. In a theoretical sense, textual languages and visual or diagrammatic ones have no principal difference, but when rigor and formality are called for, properly defining diagrams seems much harder.

Moreover, although semantics actually describes a language's meaning, computer tools make it impossible to manipulate semantics directly. Instead, everything on paper or the screen is a syntactic representation. This is also true of the machine's internal representation, the so-called abstract syntax or metamodel.

Because a rigid syntax is critical to correct language interpretation, any attempt to compromise it could be disastrous. Writing `read(data)` in a language in which the input commands are of the form `input(data)` will result in a syntax error, for example. And a computer can't exactly recognize the command, "How about getting me a value for *K*?" Thus, a formal, concise, and rigid set of syn-

Wrong Ways to View Semantics

It is an understatement to say that different people view semantics differently across software and systems engineering. After listening to numerous presentations and reading even more papers, we have identified both specializations of the general concept and downright misuses. The following are some common erroneous views, many in the context of UML.

Semantics is the metamodel. This is a common misuse of the term. The metamodel is but a way to describe the language's syntax; it is a crucial precursor, but it is not the semantics itself. Knowing what a language looks like does not equate with understanding what it means.

Semantics is the semantic domain. Some people use the word semantics as shorthand for the statement, "The semantics is given in terms of a particular semantic domain or maps the syntax into that domain." Using semantics and semantic domain interchangeably is erroneous, since it avoids the most crucial part of the semantics—the semantic mapping.

Semantics is the context conditions. This use of the term has its roots in compiler theory, where everything beyond the basic context-free grammar is viewed as semantics. It seems to have had a great influence on the way Object Constraint Language constraints are used on top of UML's metamodel. This use of the term semantics is also erroneous, as it does not entail either a semantic domain or a semantic mapping. It simply further constrains the syntax. In the UML standardization documents, "static semantics" is used instead of "context conditions."

Semantics is dealing with behavior. Some of the most intricate languages deal with behavior, especially reactive behavior. Their semantics must prescribe the system's behavior for each allowed program/model/expression, so that for such languages, behavior and semantics are closely related. However, structure description languages, for example, don't talk about behavior, but they still need semantics. Hence, semantics and behavior are not to be confused.

Semantics is being executable. Taking the previous point one step further, some people equate having semantics with being executable. Clearly, if a language is executable, it probably has an adequate semantics, although that semantics might not have been given an adequately clear representation. However, not all languages specify behavior, and not all those that do so are (or need to be) executable. Also, even if the language is meant to be executable, it can have a nonexecutable, denotational semantics. Thus, in general, having adequate semantics has little to do with a language's ability to be executed.

Semantics is the behavior of a system. Sometimes people talk about the semantics of a particular system—the way it behaves, its reaction time, and so on. This is quite different from the semantics of the languages used to describe that system.

Semantics is the meaning of individual constructs. People often refer to the semantics of some part of the language, even just one construct. Clearly, there is much more to semantics than that.

Semantics means looking mathematical. When some people see that parts of a language definition have mathematical symbols, they are convinced that it is probably also precisely defined. This is simply not true.

Semantics is ________. Some people simply give a buzzword to indicate something about how the semantic definition goes, as in "the semantics is given by message-passing." This prompts others to think that the language is properly endowed with semantics. Sadly, the worst cases are when the people making this kind of statement actually believe it themselves.

Two Language Examples

To illustrate the difference between syntax and semantics, we offer two simple language examples, one based on arithmetic expressions and one based on dataflow diagrams.

Arithmetic expressions

A BNF-like grammar gives the syntax for simplified arithmetic expressions. The grammar's main composition rule is

```
<Exp> ::= <Number> | <Variable>
        | ( <Exp> ) | – <Exp>
        | <Exp> + <Exp> | <Exp> * <Exp>
        | foo ( <Exp> )
```

The language's basic expressions are the arithmetic operations, the function symbol foo (a symbol often used to illustrate examples of things with no particular meaning or importance), and the symbols used to define numbers and variables.

For the semantic domain *S* we choose all natural numbers ($S_{<\mathrm{Exp}>}$ = *Nat*), and the semantic mapping *M* associates a number with each expression: $M_{<\mathrm{Exp}>}$: <Exp> → *Nat*. Standard mathematics is a natural notation for describing the mapping.

In the same way that developers and mathematicians build expressions as syntax trees, the semantic mapping is defined inductively. The basic cases are arithmetic constants like *M*("42") = 42. Accordingly, variables need a variable assignment, which the environment provides.

In the inductive cases, expressions containing simpler expressions are combined using operators. The obvious mapping of "+" is to the mathematical operation of addition, for example. We could map the syntactic character "+" to something else, such as the modulo operation, but that would be strange. Therefore, we map "+" to the operation plus in the expected way. Thus, if an expression has the form *a* "+" *b*, and *a*,*b* ∈ <Exp>, the semantics of the combination is

```
M(a “+” b) = M(a) + M(b)
```

where the quotation marks distinguish the symbol's use as syntax from its more common mathematical use.

This kind of definition is perhaps annoyingly obvious, but it is extremely important, especially for functions that do not have a commonly agreed-on interpretation. For the function symbol "foo," for example, we choose

```
M( “foo(” a “)” ) = M(a) * M(a).
```

Dataflow diagrams

Figure A shows a sample expression in the dataflow language.

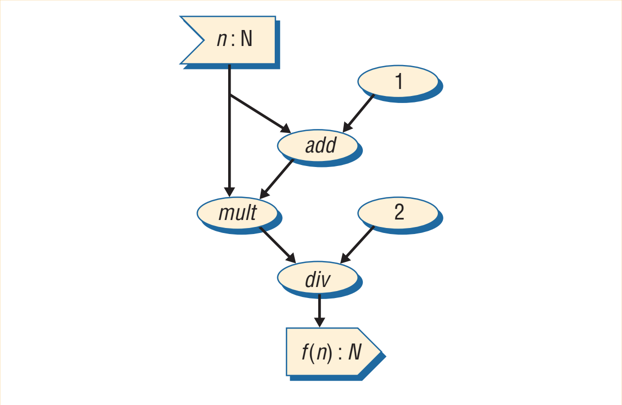


Figure A. A sample dataflow language expression. The dataflow calculates output f(n) = n*(n+1)/2 from its input n.

tactic rules is essential for precise communication.

Consider the algorithm

```
K = read();
X = 0;
for  (Y = 1 ; Y ≤ K ; Y ++) {
       X = X + Y;
}
print (X);
```

in which the authors want the computer to calculate and print the sum of all natural numbers up to the input K. The computer (as well as other people) must have this same semantic interpretation and must therefore somehow be told about the program's intended meaning. This is the responsibility of the semantics. When carefully devised, the semantics assigns an unambiguous meaning to each syntactically allowed phrase.

Of course, without semantics, the syntax is worthless because severe misinterpretations can occur, such as reading X = X + Y as "at this particular point, X must be equal to X + Y, and the program must check that fact." Worse, who says that the keywords `for`, `print`, and `read` have anything to do with their English meanings? Who says that "+" denotes addition? And what does "++" mean anyway?

It is possible to guess the meaning of most terms, since a good language designer probably chooses keywords and special symbols with a meaning similar to some accepted norm. But a computer cannot act on such assumptions. To be useful in the computing arena, any language—whether it is textual or visual or used for programming, requirements, specification, or design—must come complete with rigid rules that clearly state allowable syntactic expressions and give a rigid description of their meaning.

Semantic domain

Agreement on a language's meaning is partly a sociological process, without which the communicated data is worthless. A language's semantics

Dataflow diagrams consist of computational nodes equipped with input and output channels for communication. Directed dataflow links connect channels in a one-to-many style. Special nodes describe the diagram's interface to the environment.

We can define a dataflow diagram's semantics several ways, depending on what is being described. If the intent is to describe the structure only, the semantics should prescribe a white-box structural view for each enclosing component. This allows a hierarchical decomposition, but nothing is said or meant about whether, when, or why data will actually flow.

If the aim is to incorporate behavioral aspects, new questions arise. Does a computational component have memory? Can it be nondeterministic? Can the component react to partial input by emitting a partial result? Can several results be sent as a reaction to a single input? Is there a need to track the causality between input and output or is a message trace sufficient? Do the components need to be greedy, and can they emit messages spontaneously? Is there a buffer along the communication lines between components for storing unprocessed messages, or are messages lost if unprocessed? Is the fairness of processing input from different sources guaranteed? Is feedback (looping) in the diagram allowed? And on and on.

Different answers to such questions lead to a variety of different kinds of semantic domains for behavior: traces, input/output-relations, streams and stream-processing functions, and so on. In general, the less powerful components and channels are, the easier it is to define the semantic domain.

In the simplest case, the dataflow network is deterministic, reacts only to complete sets of inputs, and has no memory. It is then sufficient to adopt a function from inputs to outputs as the semantic domain: $IO_{\text{func}}: I \to O$. For this language, this would be $IO_{\text{func}}: Nat \to Nat$ defined by $IO_{\text{func}}(n) = n(n + 1)/2$.

Another semantic domain could be the set of traces, which includes observations of inputs and outputs in an interleaved manner: $IO_{\text{trace}} = \{x \mid x \in (I \cup O)^*\}$, where * denotes Kleene-iteration), but it is not possible to track a causal relationship for reactions or to describe composition properly. One way to alleviate this is to use an even richer semantic domain.[1]

Space permitting, we could describe more than 12 reasonable semantic domains for this language, each with its own special issues and complexities. Generally, however, for dataflow semantics, a subtle change in the semantic domain can improve the convenience of defining the semantic mapping for a given notation.

Because we have used only deterministic components, the semantic mapping in our example is rather easy. A deterministic history function can represent a dataflow component's semantics. The `add` component adds its inputs pointwise and thus corresponds to the semantic function $F_{\text{add}}: Nat^* \times Nat^* \to Nat^*$ by stating that on any pair of input sequences $a = [a_1, a_2, \ldots, a_k]$ and $b = [b_1, b_2, \ldots, b_l]$, with $m = \min(k,l)$, we have $F_{\text{add}}(a,b) = [a_1 + b_1, a_2 + b_2, \ldots, a_m + b_m]$.

This definition allows inputs on channels to arrive at different times, so it implicitly models buffers on the dataflow links and at the same time specifies `add` as a greedy component. The 1-component models a continuous source of constants: $F_1 = 1^*$. The function composition then simply carries out composition. In our example, $F(n) = F_{\text{div}}(F_{\text{mult}}(n, F_{\text{add}}(n, F_1)), F_2)$.[2]

References

1. M. Broy et al., *The Design of Distributed Systems—An Introduction to Focus*, rev. version, SFB-Bericht 342/2-2/92A, Technische Universität München (Tech. Univ. of Munich), Jan. 1993.
2. D. Harel and B. Rumpe, *Modeling Languages: Syntax, Semantics and All That Stuff*, tech. report MCS00-16, Weizmann Institute of Science, 2000.

must provide the meaning of each expression, and that meaning must be an element in some well-defined and well-understood domain. In the first sample language in the sidebar, for example, we chose natural numbers as the semantic domain.

A common misconception in modeling languages is to confuse semantics with behavior. Both a system's behavior and its structure are important views in system modeling: Both are represented by syntactic concepts, and both need semantics. Thus, even entity-relationship diagrams for databases or UML class diagrams also need semantics so that users know exactly what the language is defining.

The semantic domain is not to be taken lightly: It specifies the very concepts that exist in the universe of discourse. As such, it serves as an abstraction of reality, capturing decisions about the kinds of things the language should express. The domain is also a prerequisite to comparing semantic definitions. Consequently, an explicit definition of the semantic domain is crucial, and although it is necessary to define a language's meaning, the semantic domain itself is normally independent of the notation.

So how can the semantic domain be described and what does it look like? The description can be in varying degrees of formality, from plain English to rigorous mathematics. For example, defining the semantic domain of the full UML, which contains many diagrammatic sublanguages, is far from a simple matter: A satisfactory definition must involve combinations of myriad elements, such as messages, states, events, data values, Boolean values, time elements, and many combinations thereof.

At present, we see no simple and obvious way to define this complex semantic domain precisely, clearly, and readably. While descriptions in the literature go into great detail about the syntax of the various UML sublanguages, the same authors define the semantic domain informally, if at all, scattering the relevant information throughout an often extremely long verbal description. Whereas language designers have provided satisfactory semantics for several widely known modeling languages, including one or two of UML's sublanguages, full UML, with its multitude of languages and its complex set of interconnections between them, still suffers severely from this deficiency.

Semantic mapping

A sound language definition must relate the syntactic expressions to the semantic domain elements so that each syntactic creature maps to its meaning. In particular, when defining a language's semantics, it is essential to explicitly and clearly associate each syntactic operator (even obvious ones like "+") with its meaning as an operator over the semantic domain. We cannot overestimate the importance of doing this, even though the task might seem trivial in this example.

The semantic domain captures decisions about the kinds of things the language should express.

Often, language definers explain the mapping informally through examples and plain English. Regardless of the exposition's degree of formality, the semantic mapping $M: L \rightarrow S$ must be a rigorously defined function from the language's syntax L to its semantic domain S. Needless to say, an adequate semantic mapping for the full UML does not exist.

The sidebar shows how to build a semantic definition for an arithmetic expression from the semantic definitions of the expression's constituents. The resulting semantics is *compositional*, composed analogous to the way the syntax is structured, with the meaning of a composite creature being fully based on the meanings of its parts.[4] A compositional semantics is highly desirable, although often difficult (and sometimes impossible) to achieve.

REPRESENTATION

All elements of a language definition—the syntax, semantic domain, and semantic mapping—need a representation. This could entail using a fourth language or using the same language (such as basic mathematics) to describe both the semantics itself and its representation.

Either case gives rise to additional sources of confusion. Most languages have several definition layers. Many textual languages have not only clearly defined and separated layers, but also standard techniques for defining them:

- A set of characters forms an alphabet.
- Groups of characters form words, denoting keywords, numbers, delimiters, and so on. Regular expressions typically define this lexical layer.
- A third layer groups these words into sentences or expressions, usually with a context-free grammar.
- A fourth and final layer constrains the sentences by imposing context conditions, for example, that variable use be consistent with variable types.

In compiler theory, the constraints on the fourth layer are often called *semantic conditions* because semantic considerations trigger them. However, the constraints affect only the syntax; they do not contribute to the actual definition of semantics. For example, some conditions are expressed as context conditions for convenience, although they could have been expressed as part of the context-free grammar with the same effects. An example of this is the well-known priority scheme for infix operators.

A typical constraint for the arithmetic language in the sidebar restricts the set of well-formed sentences by disallowing use of the special name "foo" as a variable or as a function with more than one parameter. Language designers must define context conditions in a decidable form, since normally the parser must be able to check them.

Although there is no principal difference between textual and visual languages, it is harder to make words and sentences in a visual language. Language designers often start out with a set of topological notions that they first specialize using geometry, then put together topologically, and finally specialize once again using geometry.

The process might follow these steps:

- create the first layer with two kinds of basic topological elements: open and closed line segments (closed segments being closed Jordan curves);
- specialize these elements geometrically into several kinds of lines and closed shapes—arrows, straight lines, splines, boxes and circles, and so on—with various line styles and colors;
- arrange the geometric shapes into diagrams by first making topologically meaningful combinations using connectivity, insideness, partitioning and intersection, and the like and then laying these out geometrically in a 2D or 3D diagram; and
- create the fourth layer by imposing context conditions for the set of legal diagrams.

To characterize the elements in each syntactic layer, a notation N_L represents the syntax of language L. For textual languages, N_L typically consists of a combination of the Backus-Naur Form (BNF) and Chomsky-2 context-free grammars. As a side benefit, N_L also provides an abstract version of L, sometimes called the abstract syntax, together with an algorithm for parsing the concrete into the abstract version. In this way, the language and its abstract version become identifiable.

Defining the semantic domain S requires an underlying notation N_S. In practice, the N_S's used for this purpose are more numerous than N_L's: natural languages such as English, general-purpose formal languages such as logic and algebraic specification languages, standard mathematics, and so on.

The various N_L's and the N_S's give rise to many ways of defining the mapping between the two notations. Often authors describe the semantic mapping informally, showing specific examples of the mapping but without giving the mapping itself. An explicit definition of the mapping M, which is clearly preferable, requires a notation as well, N_M.

Candidates for N_M are pure mathematical notation[5] as well as graph transformations. The mapping notation must somehow include both N_L and N_S such that "N_L, $N_S \subseteq N_M$." Graph transformations work nicely if both domains are graph structures, and mathematics works well too, since the language designer can deal with all relevant elements within the generic mathematical framework. However, using Z or an algebraic language similar to the mapping notation would require major additional work to model the language's syntax N_L within Z. Further, using Z as the semantic domain would render an explicit mapping definition extremely difficult.

METAMODELING IN UML

Although visual formalisms are becoming increasingly popular, it is not that clear what notations would be best for describing them. For textual languages, using grammars for the syntax is widely accepted, but visual languages have two major competing approaches. One involves graph grammars,[6] which extend grammar concepts from textual languages to diagrams. The other approach calls for using a kind of entity-relationship diagram—specifically UML class diagrams—to model a diagrammatic language's abstract syntax. While class diagrams appear to be more intuitive than graph grammars, they are also less expressive.

Official UML definitions use the class diagram approach in a recursive, bootstrapping fashion. The technique, *metamodeling*, also uses context conditions written in the Object Constraint Language that help overcome the weaker expressive power. Metamodeling's advantage is that UML users, who probably have basic UML knowledge, don't need to learn a new external notation to be able to see a good syntax definition.

But however intuitive and appropriate the technique is, using it to define UML is still limited to describing syntax; the problem of defining semantics remains. Just as C++ semantics cannot be adequately understood merely from its context-free grammar and its additional context conditions, so UML is not understandable from its syntax alone. And even for the simpler issue of syntax definition, this bootstrapping requires a solid base, which means that UML language designers must first define class diagrams and the Object Constraint Language expressions in full—including semantics. Obviously, this requires techniques beyond metamodeling.

The UML standard concentrates on syntax and does not offer a rigorous definition of UML's true semantics.

In its current form, the Object Management Group's documents do not offer a rigorous definition of UML's true semantics, not even of the semantic domain. Rather, they concentrate on the abstract syntax, intermixed with informal natural-language discussions of what the semantics should be. These discussions certainly contain much interesting information on the semantics, but they are a far cry from what developers, as well as tool vendors, really need. As recent research shows, they still lack many clarifying details and contain many inconsistencies.

Actually, rigorously defining semantics for the full UML is a daunting task that would require a far more detailed mathematical analysis of UML's many loosely connected kinds of diagrams than has been done to date.

Nevertheless, defining the semantics of the language of statecharts, which is UML's pure behavioral core and can be used to drive the execution of UML models, has indeed been carried out, both for its non-object-oriented and object-oriented versions.[7,8] In such a definition, a detailed algorithm is given for computing the legal executions (runs) of any system defined using the language. The semantic domain consists of these appropriately represented system runs, and the algorithm constitutes the semantic mapping.

DEGREE OF FORMALITY

One misconception about formality is that textual and symbolic languages are inherently formal and visual, and diagrammatic languages are not. Many people believe that if a language is formal

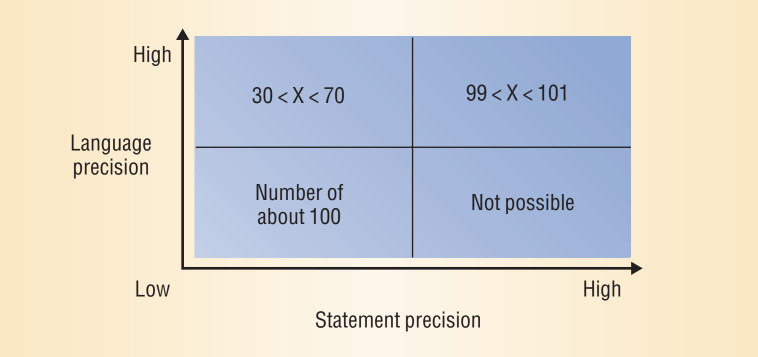


Figure 1. Language precision versus statement fuzziness. It's possible to make imprecise statements with a precise, rigorously defined language, but it's hard to be precise with an imprecise language.

looking it must be formal, as if they were measuring formality by how many Greek letters and mathematical symbols the language contains. This is a myth. Some languages, such as versions of Petri nets and statecharts, don't look formal at all, but are in fact very formal.

Admittedly, there is a correlation between a language's mathematical appearance and its degree of formality simply because people who communicate using mathematical terminology and notation tend to define things mathematically and therefore precisely as well. Still, we emphasize that "visual" and "informal" are by no means synonymous, and that "formal" is a label for any language endowed with precise and unambiguously defined syntax and semantics.

Another kind of precision is also relevant—the degree to which language expressions make precise statements. This is not the same as how precisely a language is defined. As Figure 1 shows, a language can be very rigorous, yet can still be used to make imprecise statements, but the reverse is not generally true.

Even in the complex situation of modeling systems, the degree of formality of the notation used to describe a system is orthogonal to the degree of the model's precision—the model's "detailedness." It is possible to describe a system using a rather abstract model—with many details not described—even with a fully satisfactory formal definition of UML syntax and semantics.

One of the main arguments against a formal foundation for visual languages stems from incorrectly equating model abstraction with language fuzziness. Consequently, people incorrectly conclude that a precisely defined language forces developers to fill out details they don't want to. This overspecification problem does not arise from the language's formality, but from the developers' failure to use the right abstractions. Sometimes the language's inability to provide those abstractions is at fault, but mostly the culpability lies with the tools that implement the language.

THE DOODLING PHENOMENON

In general, people tend to take diagrams too lightly, finding it difficult to consider a collection of graphics serious enough to be a language and profound enough to be the real thing. Perhaps the blame lies with the early failure of visual programming techniques to replace conventional programming languages. As a result, we often see the doodling phenomenon—a mind-set that says diagrams are what an engineer scribbles on the back of a napkin, but the real work is done with textual languages.

Sadly, too many language designers and methodologists share this view. Some find it difficult to understand why we can't simply add more graphical notations to a visual formalism without spoiling an easy to understand semantics by introducing special cases or concept combinations that contradict each other. For example, in private communication, people have proposed all kinds of extensions to statecharts, such as (actual quotes) a new kind of arrow that "means synchronization" and a new kind of box that "means separate-thread concurrency."

These well-meaning individuals seem to think that it is enough simply to add concepts and explain their intended meaning in a few words. In reality, such additions can be extremely challenging. Defining a consistent syntax and semantics for a full-fledged extension of statecharts[9] that would allow states to overlap took considerable work,[10] and the results turned out to be too complex to implement. Nevertheless, people still ask why statecharts don't support overlapping. One person, certainly a die-hard doodler, kept asking, "Why don't you just tell your system not to give me an error message when I draw these overlapping boxes?" as if that were all there was to it.

ACCOMMODATING THE INTENDED AUDIENCE

Any decision about how to represent a language definition must consider the intended audience. Are potential readers notation developers, language definers, methodologists, tool vendors, or users?

If the target audience is users, formulas won't be suitable. Typical users won't try to understand even the semantic domain definition, especially if they have to first understand the notation for it (N_S), which itself is probably another formal language. Because no semantic formalism is understandable to a broad range of users, it's probably best to use natural language with many examples to explain the notation and carefully describe the semantics.

Language developers and methodologists, on the other hand, would be willing to cope with the notations for semantics. Developers would gather insights into what would be the best form for language concepts. Methodologists would be motivated to use the notation in the interest of discovering how to advise language users.

Tool vendors should also be exposed to a rigorous semantics, but they are probably better off with precise descriptions of "how to deal with" instead of the "what" and the "why." They typically will be less interested in a definition of what the notation means mathematically, preferring to learn how to generate code and tests from it that are faithful to the original semantics. Some vendors are also interested in rules for adding, removing, and adapting notation elements, as in refinement, refactoring, and transformation calculi.[5,11]

PERTINENT QUESTIONS

Usually, deep insights result from the rigorous process of defining a language's semantics—insights that are foundational to improving the language itself. This process should involve seeking answers to at least these four questions:

- Does the given formalization capture the intended users' intuition?
- Are the context conditions sufficient to ensure that language expressions are consistent and meaningful?
- Does the notation permit the specification of important semantic domain properties?
- If analysis techniques or transformations for the language exist, are they sound with respect to the semantics?

A tremendous amount of work is necessary to address such questions, but it is work that must be part of any serious language definition. A prerequisite for determining sound analysis techniques and transformations (fourth question) is an explicit definition of the semantic mapping. Other questions address issues of user consensus and acceptance, which are based on a broadly accepted, clear, and precise standardization of both syntax and semantics.

Developers use programming languages to introduce new classes, attributes, or operations, and multinotation modeling languages are often extendable as well. UML has numerous mechanisms for introducing new elements. Besides classes, methods, and other so-called first-class citizens, UML allows users to specialize the meaning of certain elements through stereotypes and tagged values. Unfortunately, however, UML does not offer a mechanism to precisely describe the meaning of these additions within the language itself. Instead, users resort to informal descriptions, which places UML at an even greater distance from the ultimate goal of a full, well-defined language.

Another disturbing issue with broad modeling frameworks like UML is the possibility of constructing conflicting descriptions. We won't attempt a detailed discussion of this, but suffice to say that when multiple language views offer different ways to capture the same aspects of the modeled system, users will get into trouble. Thus, many UML users discover that the behavioral aspects of their specifications overlap, causing redundancy at best and inconsistency at worst. Therefore, providing a formal semantics for all such sublanguages and all their interconnections, together with tools for analyzing and executing their behavior and for consistency checking, is crucial.

As the complexity of modeling languages such as UML increases, so does the confusion about how to properly define them. Rather than being merely a definitional issue, this is fundamental to using such languages for specifying and designing critical large-scale systems, especially systems with intricate dynamic behavior.

The crux of proper modeling language definition is in the clear distinction between syntax and true semantics. Semantics is not merely a term that theoreticians use to prove theorems. Rather, it denotes an extremely important issue in language definition. Language designers and users must take the time to understand what semantics is about, how to describe it, and how to recognize its unique purpose. Only then will we be able to provide better tools and methods for the complex modeling languages now being used and help ease the introduction of future ones. ■

References

1. D. Harel, "On Visual Formalisms," *Comm. ACM*, vol. 31, no. 5, 1988, pp. 514-530.
2. E. Dijkstra, "On the Economy of Doing Mathematics," *The Mathematics of Program Construction*, J. Woodcook, C. Morgan, and R. Bird, eds., Springer-Verlag, 1993.
3. F.P. Brooks Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, May 1987, pp. 10-19.
4. E-R. Olderog, "Semantics of Concurrent Processes: The Search for Structure and Abstraction," Parts I and II, *Bull. EATCS*, 1986, no. 28, pp. 73-97, and no. 29, pp. 96-117.

5. B. Rumpe, *Formal Method for Development of Distributed Object-Oriented Systems*, Herbert Utz Verlag Wissenschaft, Munich Univ. of Technology (in German), 1996.
6. H. Ehrig, "Introduction to the Algebraic Theory of Graph Grammars," *Proc. Int'l Workshop Graph-Grammars and Their Application to Computer Science and Biology*, V. Claus, H. Ehrig, and G. Rozenberg, eds., LNCS 73, Springer-Verlag, 1979.
7. D. Harel and A. Naamad, "The STATEMATE Semantics of Statecharts," *ACM Trans. Software Eng. Methodologies*, vol. 5, no. 4, 1996, pp. 293-333.
8. D. Harel and H. Kugler, "The Rhapsody Semantics of Statecharts (or On the Executable Core of the UML)," *Proc. 3rd Int'l Workshop Integration of Software Specification Techniques for Applications in Engineering, H. Ehrig et al., eds.*, LNCS 3147, Springer-Verlag, 2004, pp. 325-354.
9. D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Scientific Computer Programming*, vol. 8, 1987, pp. 231-274.
10. D. Harel and H.A. Kahana, "On Statecharts with Overlapping," *ACM Trans. Software Eng. Methodologies*, vol. 1, no. 4, 1992, pp. 399-421.
11. W.F. Opdyke and R.E. Johnson, *Creating Abstract Superclasses by Refactoring*, tech. report, Computer Science Dept., Univ. of Illinois and AT&T Bell Laboratories, 1993.

***David Harel** is dean of the Faculty of Mathematics and Computer Science at the Weizmann Institute of Science. He has worked in many computer science areas, including automata and computability theory, logics of programs, database theory, software and systems engineering, visual languages, diagram layout, modeling and analysis of biological systems, and the synthesis and communication of smell. Harel was also a cofounder of I-Logix Inc. He invented statecharts, coinvented live-sequence charts, and was a member of the team that designed Statemate and Rhapsody. Recently, he coinvented the play-in/out approach to scenario-based programming and the Play-Engine. He is a Fellow of the ACM and the IEEE. Contact him at dharel@weizmann.ac.il.*

***Bernhard Rumpe** is head of the Software Systems Engineering Institute at the Technische Universität Braunschweig. He advocates an approach to software development that includes rigorous model analysis, test modeling, and model refactoring, as well as application of the methodologies underlying these new technologies for industrial projects. He has contributed to the definition of UML, to OMG's model-driven architecture, and to the development and enhancement of software engineering processes. Contact him at b.rumpe@tu-bs.de.*