Modeling Languages: Syntax, Semantics and all that Stuff (or, What's the Semantics of "Semantics"?)

David Harel
Weizmann Institute of Science
dharel@weizmann.ac.il

Bernhard Rumpe
IRISA/Universite de Rennes 1 and
Technische Universität Braunschweig
www.sse.cs.tu-bs.de

July 18, 2004

Abstract

Motivated by the confusion surrounding the proper definition of complex modeling languages, especially the UML, we discuss the distinction between syntax and true semantics, and the nature and purpose of each.

Keywords: Semantics, Syntax, Language, Modeling, UML, Formality
ACM taxonomy: D.2, F.3.1

1 Introduction

The **unified modeling language** (UML) [OMG03] is a complex collection of mostly diagrammatic notations for software modeling. One upshot from its standardization by the OMG is an ongoing, vivid discussion about its semantics and how to represent it. There is a large amount of work available that discusses subsets and adaptations of the UML, with the goal of providing it with a precise semantics and extracting results from this efforts. However, this work needs to be carefully assessed. First, authors often have quite different things in mind when they use the term semantics. Second, implicit assumptions are often made in such work, which influence the definitions and results. It is thus extremely difficult to compare published research on the semantics of the UML, since the comparison must take into account the subsets dealt with, the kind of systems being assumed, the relationships between the constructs treated, the levels of detail used in defining the language, and the notations and representations used in the publications themselves.

This situation motivated us to write this paper. Undoubtedly, there is a multitude of concepts surrounding the proper definition of complex modeling languages. We feel that there is often confusion as to what these concepts really mean, which of them are crucial and which marginal, and how they are to be understood and used. This definitely occurs in the UML, a large and multi-faceted effort that has an ever-growing number of followers, but it is also true for other approaches to modeling.

We have thus set out to try to clarify the notions involved in defining modeling languages, with an eye towards the particular difficulties arising in the UML. Our main theme is the distinction between the syntax of a language (the notation), its semantics (the meaning) and their representation. These are of quite different nature and have different purposes, styles and usage. Towards the end we even make an attempt to list the multitude of diverse things people mean when they use the term "semantics".

We have made an effort to address what we feel are the central and most important issues, and present them in a clear and direct fashion. We hope that language users and designers, methodologists, authors, educators and tool vendors may find our exposition useful.

2 Syntax versus Semantics

Much has been said about the distinction between the puristic notion of **information** and its syntactic representation as **data**. There is general agreement in the literature that data is used to communicate, but that an interpretation is needed for extracting the information behind it. An interpretation will be a **mapping** that assigns a meaning to each (legal) piece of data.

The two notions are often mixed up, thus becoming a major source of confusion. On the one hand, the same piece of information may be encoded in a variety of pieces of data. For example,

"June 20th, 2000"

and

"The last day of the first spring in the second millennium"

denote the same information. On the other hand, the same piece of data may have several meanings and may therefore denote different information for different people or for different applications. We thus distinguish between **syntax** and **semantics**.

People use natural languages to communicate with each other, and machines use machine readable languages. There is a great variety of data in natural languages, or in artificial ones like Morse code, as well as in the great variety of machine-based communication media, such as programming or hardware description languages. Partners to communication must thus agree on their communication language, which fixes the set of data that can be communicated.

Accordingly, a **language $\mathcal{L}$** consists of a syntactic notation (syntax), which is a possibly infinite set of legal elements, together with their meaning (semantics). Various terms are used for the syntactic elements in different kinds of languages: words, sentences, statements, boxes, diagrams, terms, models, clauses, modules, etc. We will use the rather general term **expression** for these. In many languages, complex expressions can be constructed from basic ones using special composition mechanisms and different types of expressions exist.

As a very simple example of a language we shall use basic arithmetic expressions, as defined in Box A. To lead us into graphical/diagrammatic languages, we offer another simple example, used in many modeling frameworks — data-flow diagrams — discussed in Box B.

Textual languages are symbolic in spirit, and their basic syntactic expressions are put together in linear sequences of characters. In contrast, **iconic languages** are those whose basic expressions are small pictorial signs that refer to the elements they visually depict. An iconic language can be more intuitive than a textual one, but only if the icons are not abused. In useful diagrammatic languages, which have been termed **visual formalisms** [Har88], topological and geometric notions are employed: basic expressions include lines, arrows, closed curves and boxes, and composition mechanisms involve connectivity, partitioning and insideness. Despite some well-known critiques of diagrams [Dij93, FPB87], such languages are proving to be of great help in software and systems development. In a theoretical sense, there is no principal difference between textual languages and visual, diagrammatic ones. However, as discussed later, when it comes to the need to be rigorous and formal, diagrams appear to be much harder to define properly.

We use the term 'syntax' for the notation of the language. Syntactic issues focus purely on the notational aspects of the language, disregarding issues of meaning. The meaning of a language is described by its semantics. Interestingly, computerized tools do not allow us to manipulate the semantics directly. Instead, everything we see and work with on paper or on the screen is a syntactic representation. This also holds for the machine's internal representation, the so called **abstract syntax** or **meta-model**.

For example, a programming language must have a rigid syntax to be processable by a compiler. Any attempt to stretch this syntax might turn out to be disastrous. For example, if "**read(**data**)**" is written in a language whose input commands are of the form "**input(**data**)**", the result will be a **syntax error**. And of course, we cannot hope to address the computer with something like "*how about getting me a value for K*." Thus, a formal, concise and rigid set of syntactic rules is essential for precise communication.

Here is an algorithm written in a typical programming language:

```
$K = $ **read**();
$X = 0;$
**for** ($Y = 1$ ; $Y \leq K$ ; $Y$++) {
    $X = X + Y;$
}
**print** ($X$);
```

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

Without semantics the syntax is worthless, since 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 has to check that fact*". Worse, who says that the used keywords **for**, **print**, or **read** have anything at all to do with their meanings in English? Who says that "+" denotes addition? And what does "++" mean anyway? We might be able to *guess* what is meant by most of these, since a good language designer probably chooses keywords and special symbols intending their meaning to be similar to some accepted norm. But a computer cannot be made to act on such assumptions.

To be useful in the computing arena, any language — be it textual or visual, and used for programming, requirements, specification, or design — must come complete with rigid rules that prescribe which syntactic expressions are allowed, and also with a rigid description of their meaning.

3 The Semantic Domain

Agreement on the meaning of a language is an important and partly sociological process, without which the communicated data is worthless. Any reliable **semantics** for a language $\mathcal{L}$ must consist of two parts: a **semantic domain** $\mathcal{S}$ and a **semantic mapping** $\mathcal{M} : \mathcal{L} \to \mathcal{S}$ from the syntax to the semantic domain.

Let us explain. The semantics of a language must provide us with the meaning of each of its expressions. That meaning must be an element in some well-defined and well-understood domain. For example, natural numbers are chosen as a semantic domain for $\langle \mathit{Exp} \rangle$ in Box A, and the discussion in Box B shows that there is a variety of choices for an appropriate semantic domain for data-flow diagrams.

A common misconception in the world of system modeling languages is to confuse the term 'semantics' with the term 'behavior'. Both the behavior and the structure of a system are important views in modeling a system, both are represented by syntactic concepts and both need semantics. Therefore, languages such as ER-diagrams for databases, or class diagrams in the UML, also need semantics, so that we know exactly what is being defined.

The semantic domain is not to be taken lightly: it specifies the very concepts that exist in the universe of discourse. It thus serves as an abstraction of reality, capturing our decisions about the kinds of things we want our language to express. It is also a prerequisite to comparing different semantic definitions. Consequently, an explicit definition of the semantic domain is crucial, and although it is needed for defining the meaning of a language, the definition of the semantic domain itself is normally independent of the notation.

How does one describe the semantic domain and what does it look like? Well, the description can be given in varying degrees of formality, from plain English all the way to rigorous mathematics. Jumping for a moment to the full UML, we note that defining its semantic domain is far from being a simple matter: to be satisfactory, it must involve combinations of numerous kinds of elements, such as messages, states, events, data values, booleans, time elements, etc., and many kinds of combinations thereof. There seems to be no obvious way to define this complex semantic domain in a precise, clear and readable manner. Whereas UML descriptions in the literature are very detailed when it comes to syntax, defining even just the semantic domain is much more difficult. It is usually done very informally, if at all, with the relevant information being scattered throughout an entire (often extremely lengthy) description.

4 The Semantic Mapping

Given a syntax $\mathcal{L}$ and a semantic domain $\mathcal{S}$, the final and main step in defining a semantics is to relate the syntactic expression to the elements of the semantic domain, so that each syntactic creature is mapped to its meaning. Thus, it is important to say explicitly and clearly that each syntactic operator (even the obvious ones like "+") are mapped accordingly. One should not underestimate the importance of defining this mapping, although in this particular example it might seem trivial.

Often the mapping $\mathcal{M}$ is explained informally, by examples and in plain English. But regardless of the degree of formality of its exposition, the semantic mapping itself must be a rigorously defined function from $\mathcal{L}$ to $\mathcal{S}$, written

$$\mathcal{M} : \mathcal{L} \to \mathcal{S}$$

The semantic definition for an arithmetic expression is built up, as shown in Box A, from the semantic definitions of the expression's constituents. When this is done in a general way, we say that we have a **compositional**

semantics, since it allows us to compose the semantics analogously to the way the syntax is structured, with the meaning of a composite creature being based on the meanings of its parts; see [Old86]. Compositionality is highly desirable — though often difficult to achieve (and in some cases provably impossible).

5 Representation

All elements of a language definition — the syntax, the semantic domain, and the semantic mapping — need a representation. This could entail using yet another language, a fourth one, or using the same language (such as basic mathematics) to describe both the semantics itself and its representation. In either case we have additional sources for confusion.

Most languages have several layers of definition. For many textual languages we find not only clearly defined and separated layers, but also standard techniques for defining most of them:

1. A set of characters forms an alphabet.
2. Characters are grouped into words, denoting keywords, numbers, delimiters, etc. This lexical layer is typically defined by regular expressions.
3. A third layer groups these words into sentences or expressions, usually by a context free grammar.
4. A fourth and final layer constrains the sentences by imposing context conditions. (For example that variables are used consistent with their types.)

In compiler theory, the constraints on Clause 4 are often called **semantic conditions**, as they are triggered by semantic considerations. However, it is important to realize that they really just constrain the syntax and do not contribute to the definition of semantics. Some kinds of conditions are expressed as context conditions for convenience, although they could have been expressed as part of the context free grammar. A typical example is the priority scheme for infix operators.

A typical constraint for the language of arithmetic expressions from Box A restricts the set of well-formed sentences by disallowing the use of the special name “`foo`” as a variable or as a function with more than one parameter. It is important that the context conditions are decidable, since they normally have to be checkable by the parser.

As stated earlier, in some ways there is no principal difference between textual and visual languages. However, in the latter case it is far less easy to make out the words and sentences. Often we would have a set of topological notions, that would then be specialized using geometry, then put together topologically, and then specialized once again using geometry. Here's how this might go:

1. The first layer consists of two kinds of basic topological elements — open line segments and closed ones (mathematically, the latter are just closed Jordan curves).
2. These elements are specialized geometrically into several kinds of lines and closed shapes (e.g., arrows, straight lines and splines, boxes and circles, etc., all with various line styles and colors).
3. The geometric shapes are arranged into diagrams by first making topologically meaningful combinations of them, using, e.g., connectivity, insideness, partitioning and intersection, and then laying these out geometrically in a two- (or three-) dimensional diagram.
4. The fourth layer yields the set of legal diagrams, by imposing context conditions.

The textual attributes are used in the second layer, e.g., as class names or expressions, and can be defined using a conventional textual grammar.

So generally we need a notation to characterize the elements in each layer of the syntax of a language $\mathcal{L}$. We call this notation $\mathcal{N}_{\mathcal{L}}$. For textual languages, $\mathcal{N}_{\mathcal{L}}$ will typically consist of a combination of the Backus-Naur Form (BNF) and Chomsky-2 context free grammars. As a side benefit, a notation $\mathcal{N}_{\mathcal{L}}$ also provides an abstract version of $\mathcal{L}$, sometimes called the **abstract syntax**, together with an algorithm for parsing the concrete into the abstract version. We can thus identify the language with its abstract version.

To define the semantic domain, we again need an underlying notation, call it $\mathcal{N}_{\mathcal{S}}$. Here, the variety of possibilities is much larger than for the syntax. Besides natural languages such as English, general purpose formal languages can be used, such as logics and algebraic specification languages, or standard mathematics.

As to the semantic mapping, the various $\mathcal{N}_{\mathcal{L}}$ notations used for the syntax and the $\mathcal{N}_{\mathcal{S}}$ notations used for the semantic domain give rise to a great variety of ways to define the mapping between the two. In many attempts to define semantics, the semantic mapping is given informally, by showing specific examples of how the mapping goes, without giving the mapping $\mathcal{M}$ itself. However, when $\mathcal{M}$ is to be given explicitly (and this is clearly the preferred way of doing things), a notation is required for it too, call it $\mathcal{N}_{\mathcal{M}}$. We can use pure mathematical notation for $\mathcal{N}_{\mathcal{M}}$ [Rum96], or, alternatively, **graph transformations**.

It is rather straightforward to understand that a notation for the mapping must somehow include the notations for the syntax and the semantic domains, so that $\mathcal{N}_\mathcal{L} \subseteq \mathcal{N}_\mathcal{M}$ and $\mathcal{N}_\mathcal{L}, \mathcal{N}_\mathcal{S} \subseteq \mathcal{N}_\mathcal{M}$. Graph transformations work nicely if both domains are graph structures, and mathematics works well too, since all relevant elements can be dealt with within the generic mathematical framework. However, using Z or a similar algebraic language as $\mathcal{N}_\mathcal{M}$ would require major additional work to model the syntax of the language within Z. Furthermore, the use of Z as the semantic domain $\mathcal{S}$ would render an explicit definition of the mapping $\mathcal{M}$ extremely difficult.

6 Meta-Modeling in the UML

Although visual formalisms are becoming increasingly popular, it is not that clear what notations would be best for describing them. For textual languages the use of grammars for the syntax is widely accepted, but for visual languages there are two major competing approaches. One involves **graph grammars** [Ehr79], which extend grammatical ideas from textual languages to diagrams. The other calls for using a kind of entity relationship diagram — specifically, UML class diagrams — to model the abstract syntax of a diagrammatic language. While class diagrams appear to be more intuitive than graph grammars, they are also less expressive.

The official definitions of the UML use the class diagram approach in a recursive, "bootstrapping" fashion [OMG03]. The technique is called **meta-modeling** and also uses context conditions written in OCL that help overcome the weaker expressive power. The meta-model approach to the definition of the syntax of the UML is elegant. From a pragmatic point of view, it is very useful, since UML users will probably have a basic knowledge of the UML when they get to look at its detailed definition, and they don't have to learn a new external notation to be able to see a good definition of the syntax.

However, it is important to emphasize that however intuitive and appropriate, using the meta-model to define the UML is mostly limited to describing the syntax only; there still remains the problem of defining semantics. Just as the semantics of `C++` cannot be understood from its context free grammar and its additional context conditions, so is the case for the UML. Thus, UML is the language $\mathcal{L}$ under discourse, but is is also (actually, a subset thereof) the notation $\mathcal{N}_\mathcal{L}$ used to describe the syntax. Moreover, even for this simpler issue of defining the syntax, the core of the technique needs a solid basis: the class diagrams and OCL expressions themselves must be defined in full — including semantics! — which requires additional techniques.

Turning to the semantics of the UML, the official document titled "Semantics of UML" [OMG03] does not really offer a rigorous definition of the true semantics of the language, not even the semantic domain, but concentrates rather 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 is really needed. One possibly good way to define the dynamic semantics of the UML would be to provide a detailed algorithm for computing the legal executions (i.e., runs) of a system defined using the language. This mean that the semantic domain could consist of appropriately represented runs of the system, and the algorithm would constitute the semantic mapping. An example of how this can be done for the non-OO version of the language of statecharts can be found in [HN96], and a version dealing with the OO version, which is at the heart of the UML in [HK03].

7 The Degree of Formality

One misconception about formality is the belief that textual/symbolic languages are *a priori* formal and visual/diagrammatic ones are not. It is a myth that a "formal-*looking* language" is a formal language; that formality of a language can be equated with its containing lots of Greek letters and mathematical symbols. There are, in fact, languages that don't look very formal at all, but are in fact as formal as anything (e.g., versions of Petri Nets and statecharts). Admittedly, there is probably a high correlation between the mathematical appearance of a language 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, it is important to realize that "visual" and "informal" are by no means synonymous. We use the term "formal" to emphasize that the language has been endowed with precise and unambiguous definitions of syntax and semantics.

There is also another kind of precision relevant here — the degree to which expressions in the language make precise statements, as opposed to the precision of the language's definition. A language can be very rigorous, yet make imprecise statements (see Figure 1). Using a precise language we still can make imprecise statements, but not the other way round.

Even in the more complex situation of modeling systems, we find that the degree of formality of the notation used to describe a system is orthogonal to the degree of precision (the 'detailedness') of the model. In particular, we could describe a system using a rather abstract model — that is, many details are not described — even if we had a fully satisfactory definition of the syntax and semantics of the UML.

One of the main arguments against a formal foundation for visual languages arises from the confusion between these two concepts, i.e., equating abstraction of the models with the fuzziness of the language. A result of this confusion is the incorrect statement that a precisely defined language forces developers to fill out details they don't want to. Indeed, the latter

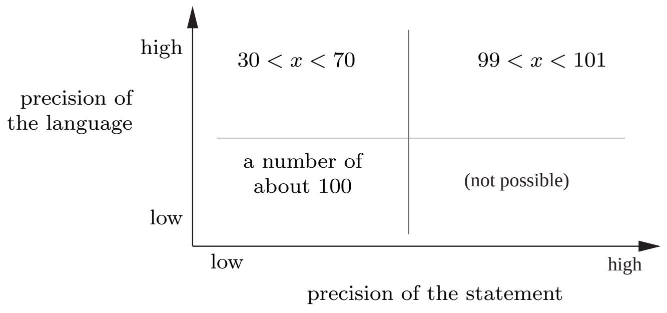


Figure 1: Precision of the language vs. fuzziness of its statements

problem, called **over-specification**, does not arise from the formality of the language used, but from failure on the part of the developers to use the right abstractions. This is sometimes a consequence of the inability of the language, but mostly of the tools implementing it, to provide appropriate abstraction mechanisms.

8 The Doodling Phenomenon

Unfortunately, many people take diagrams too lightly. They find it hard to consider a collection of graphics to be serious enough to be called a language and profound enough to be the 'real thing'. Perhaps this is a result of the early failure of visual programming techniques to replace conventional programming languages.

Often, one encounters the 'doodling phenomenon', whereby diagrams are considered to be what an engineer scribbles down on the back of a napkin, as a kind of visual aid. The "real" work, the rationalization goes, has to be done with textual languages. Sadly, many language designers and methodologists have this view too.

Some people find it hard to understand why you can't simply add more and more graphical notations to a visual formalism without spoiling an easy to understand semantics, introducing various special cases, or concept combinations that contradict each other. For example, people have tried to propose (in private communication) all kinds of extensions to the language of statecharts, like a new kind of arrow that "means synchronization", or a new kind of box that "means separate-thread concurrency" (these are actual quotes). It seems that you can simply add more concepts and just say in a few words what they are intended to mean.

A good example of how difficult such additions can really be is the idea of allowing states to overlap in a statechart. This possibility was proposed in a preliminary way in [Har87], but it took a lot of hard work to figure out a consistent syntax and semantics for such an extension [HK92]. Interestingly, the result turned out to be complex enough that it was not recommended for implementation. Nevertheless, people still ask why overlapping is not supported in statecharts. For example one person, who certainly had doodling in mind, kept asking this: “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 that was needed ...

9 The Intended Audience

When selecting the representation $\mathcal{N}$ of a language definition the intended audience must be taken into consideration. Potential reader-groups include notation developers, language definers, methodologists, tool vendors and users.

If the definition is intended for the user, we can forget about using formulas. Typical users will not make the effort to understand even the precise definition of the semantic domain $\mathcal{S}$, especially if to do so they would need to understand the notation $\mathcal{N}_{\mathcal{S}}$, which itself is probably another formal language. Since there is no semantic formalism that is commonly understood by a broad range of users, it is probably best to use natural language and many examples for explaining the notation and carefully describing the semantics.

In contrast to users, language developers and methodologists would be willing to cope with the notation $\mathcal{N}_{\mathcal{S}}$; the former in order to gather insights into the best form of concepts for the language, and the latter to make recommendations to users about using it.

Tool vendors should also be exposed to a precise semantics, but they are probably better off with precise descriptions of “how to deal with” instead of the “what” and the “why”. They will typically be less interested in a definition of what the notation means mathematically, and more in how to generate code from it that is faithful to the original semantics. Some vendors are also interested in rules for adding, removing and adapting elements of the notation, as in refinement, refactoring and transformation calculi [Rum96, OJ93].

10 More on the Importance of Semantics

Very often, deep insights are gained from carrying out a rigorous semantic definition of a language, and these can then be used to improve the language itself. Some relevant questions to ask when defining semantics are:

1. Does the given formalization capture the intuition of the intended users?

2. Are the context conditions sufficient to ensure consistency and meaningfulness of the language expressions?

3. Does the notation allow the specification of the important properties of the semantic domain?

4. If analysis techniques or transformations for the language exist, are they sound with respect to the semantics?

A tremendous amount of work is necessary for a semantics to properly address such questions, but it surely must be done for any serious language definition effort. A necessary prerequisite for success with respect to Question 4 is an explicit definition of the semantic mapping $\mathcal{M}$. Other questions address issues of consensus and acceptance among users, which are based on a broadly accepted, clear and precise, standardization of both syntax and semantics.

Programming languages allow to introduce new classes, attributes or operations. Multi-notation modeling languages are also often extendable. The UML has quite a number of 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, the UML does not offer a mechanism to precisely describe the meaning of these additional kinds of elements within the language itself. Instead, informal descriptions are frequently used. This places the UML at an even greater distance from the ultimate goal of a full, well-defined language.

Another disturbing issue with broad modeling frameworks like the UML is the possibility of the user constructing conflicting descriptions. A detailed discussion of this is beyond the scope of our paper, but suffices to say that when multiple views of the language offer different ways to capture the same aspects of the modeled system, users will easily get into trouble. Thus, many UML users discover that their specifications overlap, causing redundancy (in the best case) and inconsistency (in the worst case). The existence of formal semantics for all such sub-languages, together with tools for executing their behavior and for consistency-checking, is therefore a must.

11 What else do People Mean by "Semantics"?

It is quite illuminating to consider the many other ways the term "semantics" has been used by people in software and systems engineering. Listening to presentations and reading papers, we have been able to identify numerous different usages. Some are specializations of the general concept, but others are downright misuses of the term. Here are some of the most commonly found ones, many in the context of the UML:

1. **Semantics is the meta-model**: Some people refer to the meta-model as semantics. As we have said repeatedly, the meta-model is but a way to describe the syntax of the language; it is a necessary precursor, but it is not the semantics itself. Just knowing what a language looks like does not mean that you understand what it means.

2. **Semantics is the semantic domain**: The word semantics is sometimes used as shorthand for the semantic domain of the language. This is often just meant as shorthand for the statement that the semantics of the language is given in terms of the particular semantic domain, or that it maps the syntax into that domain. Still, it causes confusion.

3. **Semantics is the context conditions**: This use of the term has its roots in compiler theory, where everything beyond the basic context free grammar was regarded as semantics. It seems to have had a great influence on the way OCL constraints are used on top of UML's meta-model. This usage of the term semantics does not entail a semantic mapping at all, and hence does not constitute true semantics. It simply further constrains the syntax. The term "static semantics" is sometimes also used for it.

4. **Semantics is dealing with behavior**: The most intricate languages deal with behavior, especially reactive behavior. The semantics of such a language must prescribe the system's behavior for each allowed program/model/expression, so that for such languages behavior and semantics are closely related. However, semantics for structure description languages, for example, don't talk about behavior, so that semantics and behavior are not to be confused.

5. **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, not all those that do so are (or need to be) executable. Also, even if the language is meant to be executable, we can give it a non-executable, denotational semantics. Thus, in general, a language having semantics has little to do with its ability to be executed.

6. **System semantics vs. language semantics**: Sometimes people talk about the "semantics" of a particular system; e.g., the way it behaves, its reaction time, etc. However, that is quite different from the semantics of the language used to describe that system.

7. **Semantics is meanings for individual constructs**: People often refer to the semantics, not of the entire language, but merely of some

part of it; even just a single construct. Clearly, that's only a part of the story.

8. **Semantics is looking mathematical**: The very fact that parts of a language are defined using mathematical symbols can result in some people thinking that if it looks formal it must be well-defined. A myth, as we said earlier.

9. **Semantics on empty**: Finally, 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 very often causes other people to think that the language has been properly endowed with semantics. Sadly, the worst of these cases is when it causes the very person making the statement to think so him/herself.

Maybe "semantics" is an overloaded term, but it denotes an extremely important issue in language definition. We hope that this paper will help people understand what semantics is about, how to describe it and what it is useful for.

References

[BDD+93] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems — An Introduction to FOCUS – revised version –. SFB-Bericht 342/2-2/92 A, Technische Universität München, January 1993.

[Dij93] E. Dijkstra. On the Economy of doing Mathematics. In J. Woodcook, C. Morgan, and R. Bird, editors, *The Mathematics of Program Construction*. Springer Verlag, 1993.

[Ehr79] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proc. Int. Workshop Graph-Grammars and Their Application to Computer Science and Biology*, LNCS 73. Springer Verlag, 1979.

[FPB87] Jr F. P. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *Computer*, 31:5:10–19, 1987.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Programming*, 8:231–274, 1987.

[Har88] D. Harel. On Visual Formalisms. *Comm. Assoc. Comput. Mach.*, 31:5:514–530, 1988.

[HK92] D. Harel and H.-A. Kahana. On Statecharts with Overlapping. *ACM Trans. on Software Engineering Method.*, 1:4:399–421, 1992.

[HK03] D. Harel and H. Kugler. The Rhapsody Semantics of Statecharts (or, On The Executable Core of the UML). to appear, 2003.

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. on Software Engineering Method.*, 5:4:293–333, 1996.

[HR00] D. Harel and B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff. Technical Report MCS00-16, The Weizmann Institute of Science, Rehovot, Israel, 2000.

[OJ93] W. F. Opdyke and R. E. Johnson. Creating Abstract Superclasses by Refactoring. Technical report, Department of Computer Science, University of Illinois and AT&T Bell Laboratories, 1993.

[Old86] E.-R. Olderog. Semantics of concurrent processes: the search for structure and abstraction, Part I and II. *Bulletin of the EATCS*, 28 and 29:73–97, 96–117, 1986.

[OMG03] OMG. Unified Modeling Language. Version 2.0, OMG, 2003.

[Rum96] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996. PhD thesis, Technische Universität München.

Remark:

The following sections are to be put into two boxes.

A Sample Language: Arithmetic Expressions

The **syntax** for (simplified) arithmetic expressions is given by a BNF-like grammar, the main composition rule of which is:

$$\begin{array}{lll}\langle \mathit{Exp}\rangle & ::= & \langle \mathit{Number}\rangle \;\mid\; \langle \mathit{Variable}\rangle \\ & \mid & (\,\langle \mathit{Exp}\rangle\,) \;\mid\; -\langle \mathit{Exp}\rangle \\ & \mid & \langle \mathit{Exp}\rangle + \langle \mathit{Exp}\rangle \;\mid \langle \mathit{Exp}\rangle * \langle \mathit{Exp}\rangle \\ & \mid & \texttt{foo}\,(\,\langle \mathit{Exp}\rangle\,)\end{array}$$

The basic expressions of this language are the arithmetic operations, function symbol `foo`, and the symbols used in defining numbers and variables.

As the **semantic domain**, we may choose the natural numbers. Thus we have $\mathcal{S}_{\langle Exp\rangle} = \mathbb{N}$.

The **semantic mapping** must associate a number with each expression of the language; thus, $\mathcal{M}_{\langle Exp\rangle} : \langle Exp\rangle \to \mathbb{N}$. It is quite natural to use standard mathematics as the **notation for describing the mapping**.

As the expressions are built in form of syntax trees, the semantic mapping is defined inductively. The basic cases are arithmetic constants with $\mathcal{M}(\text{“42”}) = 42$. Accordingly variables need an variable assignment given by the environment.

In the inductive cases expressions contain simpler expressions combined by operators. The obvious mapping of the symbol “+” is to the mathematical operation of addition. It is worth emphasizing that we could have given “+” any other meaning like being the modulo operation, but that would be strange for our perception. This we handle “+”, mapping the symbol “+” to the operation plus. If an expression has the form a“+”b, with subexpressions $a, b \in \langle Exp\rangle$, we define the semantics of the combination as follows:

$$\mathcal{M}(a\text{“+”}b) = \mathcal{M}(a) + \mathcal{M}(b)$$

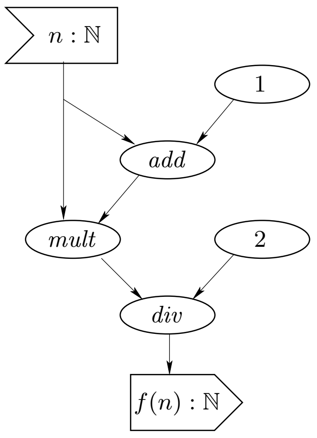
This kind of definition is perhaps annoyingly obvious, but it is extremely important, especially for functions that do not have an obvious agreed-upon interpretation like the function `foo`. We choose:

$$\mathcal{M}(\ \text{“}\texttt{foo(}\text{”}\ a\ \text{“}\texttt{)}\text{”}\ ) = \mathcal{M}(a) * \mathcal{M}(a)$$

B Sample Language: Data-Flow Diagrams

Syntax

Data flow diagrams consist of (computational) nodes that are equipped with input and output channels for communication. Channels are connected through directed data-flow links in a one-to-many style. Special nodes are used to describe the interface of the overall diagram. A sample expression of the data-flow language is:



The **semantics of data-flow diagrams** can be defined in several ways. If intended to describe structure only, the semantics would prescribe a "white box" view of the structure of each enclosing component. This allows a hierarchical decomposition, but nothing is said or meant about whether, when, or why data will actually flow.

If we want to incorporate behavioral aspects, new questions arise. Does a computational component have memory? Can it be nondeterministic? Is the component allowed to react on partial input by emitting a (partial) result? Can several results be sent as reaction to a single input? Are we interested in tracking the causality between input and output or is the trace of messages sufficient? Need the components 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 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 quite different kinds of semantic domains for behavior: traces, input/output-relations, streams and stream processing functions, and many more. In general, the less powerful components (and channels) are, the easier the semantic domain can be.

In the most simple case, the data-flow 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: $IO_{func} : I \rightarrow O$. Here this would be $IO_{func} : \mathbb{N} \rightarrow \mathbb{N}$, defined by $IO_{func}(n) = n(n+1)/2$.

Another semantic domain could be the set of traces, where inputs and outputs are observed in an interleaved manner (* means Kleene-Iteration): $IO_{trace} = \{x \mid x \in (I \cup O)^*\}$, but no causal relationship for reactions can be tracked, and composition cannot be described properly. To alleviate this, [BDD$^+$93] uses an even semantically richer domain.

In total, we know of more than twelve semantic domains, each with its own problems and complexities. Data-flow semantics is a nice example of a case where a subtle change in the semantic domain improves the convenience of defining the semantic mapping for a given notation.

The **semantic mapping** in our example is rather easy, since we have only used deterministic components. Here, we choose a deterministic history function to represent the semantics of a data-flow component.

Component *add* adds its inputs pointwise. Thus, `add` corresponds to semantic function $F_{add} : \mathbb{N}^* \times \mathbb{N}^* \to \mathbb{N}^*$, 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 $F_{add}(a, b) = [a_1 + b_1, a_2 + b_2, \ldots, a_m + b_m]$.

This definitions allows inputs on channels to arrive at different times, and thus implicitly models buffers on the data-flow links. However, *add* is a greedy component.

The 1-component models a continuous source of constants: $F_1 = 1^*$ Composition is then simply carried out by function composition. In our example we have: $F(n) = F_{div}(F_{mult}(n, F_{add}(n, F_1)), F_2)$.

([HR00] contains a more detailed discussion.)

C Bios

Prof. David Harel (dharel@weizmann.ac.il) has been Dean of the Faculty of Mathematics and Computer Science at the Weizmann Institute of Science since 1998. He is also co-founder of I-Logix, Inc., Andover, MA. He has worked in many areas of computer science, including automata and computability theory, logics of programs, database theory, software and systems engineering, visual languages, layout of diagrams, modeling and analysis of biological systems, and the synthesis and communication of smell. He is the inventor of statecharts, and co-inventor of live sequence charts (LSCs), and was part of the team that designed Statemate and Rhapsody. Recently, he has worked on the play-in/out approach to scenario-based programming. Some of his writing is intended for a general audience (see, for example, *Computers Ltd.: What They Really Can't Do* , Oxford University Press, 2000). He has received a number of awards, including ACM's Karlstrom Outstanding Educator Award in 1992. He is a Fellow of the ACM and of the IEEE.

Prof. Bernhard Rumpe is head of the Software Systems Engineering Institute at the Technische Universität Braunschweig. His is advocating a "'model engineering"' approach, that benefits from rigorous and practical approaches and including its embedding in high-quality yet efficient software development methods. This also includes the impact of new technologies to industry. In various publications he contributed to the definition of the UML, MDA, formal methods and to the development and enhancement of software engineering processes. He is author and editor of eleven books and

Editor-in-Chief of the new Springer International Journal on Software and Systems Modeling (www.sosym.org).