Modeling Languages: Syntax, Semantics and All That Stuff

Part I: The Basic Stuff

David Harel and Bernhard Rumpe

October 2000

Abstract

The motivation for this paper, the first in a planned series of three parts, is the multitude of concepts surrounding the proper definition of complex modeling languages for systems and software, and the confusion that this often causes. Particularly relevant is the case of the recently standardized UML, which we refer to quite extensively as we proceed. Our intention is to discuss and clarify the notions involved in defining modeling languages. The main theme is the distinction between syntax and semantics, the nature and purpose of each, their usage and style, and the various means for defining and dealing with them. Underlying the exposition are the dichotomies of textual vs. visual languages, structural vs. behavioral specification, and requirements vs. system models. We hope that the paper will be useful to language designers, methodologists, tool vendors and educators.

1 Introduction

With the standardization of the Unified Modeling Language (UML) [UML98] as a large and complex collection of mostly diagrammatic notations for object-oriented modeling and analysis, there is currently an ongoing, vivid discussion about its semantics. Whereas the OMG is responsible for the standardization of the UML, the semantics of the language is still undergoing extensive investigation. There is a large amount of theoretical work available that discusses subsets and adaptations of the UML, see e.g., [BM99], with the goal of giving the UML a precise semantics and extracting results from it. However, this work, some of which is useful and important, needs to be carefully assessed. To start with, authors often have quite different things in mind when they use the term semantics. Second, implicit assumptions are often made in such

[HR00] D. Harel, B. Rumpe.
Modeling Languages: Syntax, Semantics and All That Stuff.
Technical Report MCS00-16, Weizmann Institute, Rehovot, Israel, 2000.
www.se-rwth.de/publications

work, which influence the definitions and results. It is very difficult to compare papers written on the semantics of the UML, since the comparison must take into account the subsets of the notation dealt with, the assumptions on the kind of systems it is intended for, the relationships between the constructs treated, the levels of detail used in defining the language, and the notations and representations used in the papers themselves.

This situation was a major motivation in our decision to write these papers. Undoubtedly there is a multitude of concepts surrounding the proper definition of complex modeling languages. We feel that, to a large extent, there is confusion as to what these concepts really mean, which of them are crucial and which marginal, how they are to be understood and used, who needs to know what and who needs to do what. This occurs in the UML, which is becoming very popular and has an ever-growing number of followers, but is true for many other approaches to modeling as well.

We have thus set out to try to clarify some of the notions involved in defining modeling languages. These notions come in many flavors; some are basic and some advanced, and many are really hard nuts to crack.

The main theme of this series of papers is the distinction between the syntax of a language (the notation) and its semantics (the meaning). These are of quite different nature and their definitions have different purposes, styles and usage. We discuss issues arising from the adoption of different variants of semantics, with their benefits and drawbacks. As the notions unfold, we will repeatedly relate to the duality between structure and behavior, and the differences between textual/symbolic languages and visual/diagrammatic ones. We will also clarify the difference between the requirements on the system and the system's implementation model; the latter, for example, can be specified in sufficient detail to enable model execution prior to the actual implementation.

The series is organized into three parts. This paper constitutes Part I, "The Basic Stuff". It deals with the basic components of language: syntax and semantics. Included in the latter are the semantic domain and the semantic function. We discuss the representation of these components, their possible levels of formality, and their intended audience. For illustration, we use simple examples from arithmetic expressions, basic programming languages and data-flow diagrams. One interesting question that arises here is where exactly to place the constraints that are often called "semantic conditions".

Part II, "The Advanced Stuff", will address a number of issues that arise in more complex circumstances, such as when dealing with composite languages that have different and separately defined sublanguages for different parts of the modeling, often having also multiple semantic domains and mappings.

Part III, "The Really Hard Stuff", will discuss the more difficult and complex cases, the UML being an illustrative example for these. In addition to different notations or sublanguages for different things, these also give rise to *overlapping*, which, put simply,

means that there are different notations for the *same* thing, and the syntax allows these different views to be present in the model at one and the same time.

Throughout the papers we make a special effort to address what we feel are the central and most important issues, and to present them in a clear and direct fashion. We do not claim to have the whole story under control or to have all the answers, but we have done our best. It is our hope that the papers might help clarify some of the knotty issues surrounding language definition, and that language designers, methodologists, authors, educators and tool vendors may perhaps find them useful in their work, if only by virtue of the care taken in our attempt to spell out the issues responsibly.

2 Syntax

Much has been said about the distinction between the puristic notion of *information* and its syntactic representation as *data*, which is the medium used to transport and store information. There is general agreement in the literature that data is used to communicate and needs an interpretation to extract the information behind it. An interpretation is always a mapping assigning 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 millenium"

denote the same day, although in very different ways. 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*, and we shall be discussing the two in some detail.

Data serves to communicate and store information. People use natural languages to communicate with each other, and machines use machine readable languages for this. There is a great variety of data in forms of spoken or written words in natural languages, or in artificial ones like Morse code, flag signs, or the great variety of machine-based communication media, such as programming languages or hardware description languages. Partners to communication must first agree on their communication language, which fixes the set of data that can be communicated.

A *language* consists of a syntactic notation (syntax), which is a possibly infinite set of elements that can be used in the communication, together with their meaning (semantics). We often denote a language generically by $\mathcal{L}$.

Various terms are used for the syntactic elements in different kinds of languages: words, sentences, statements, diagrams, terms, models, clauses, modules, etc. We will

use the rather general term *expression* for this. In many languages, complex expressions can be constructed from basic ones using special composition mechanisms.

One example of a language is that of arithmetic expressions with an additional function `foo`, given by a BNF-like grammar, the main composition rule of which is:

$$
\begin{array}{lcll}
\langle \mathit{Exp} \rangle & ::= & \langle \mathit{Number} \rangle & | \ \langle \mathit{Variable} \rangle \\
 & | & (\ \langle \mathit{Exp} \rangle \ ) & | \ - \langle \mathit{Exp} \rangle \\
 & | & \multicolumn{2}{l}{\langle \mathit{Exp} \rangle \ [\ + \mid - \mid * \mid / \ ] \ \langle \mathit{Exp} \rangle} \\
 & | & \multicolumn{2}{l}{\texttt{foo} \ (\ \langle \mathit{Exp} \rangle \ , \langle \mathit{Exp} \rangle \ )}
\end{array}
$$

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

To lead us into graphical/diagrammatic languages, which are one of the central concerns of our papers, here is another example: data-flow-diagrams. A sample expression of the language (i.e., a sample diagram) is depicted in Figure 1.

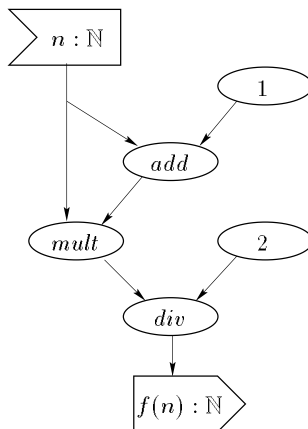

Figure 1: Sample data-flow diagram for $f(n) = \frac{n(n+1)}{2}$

Data flow diagrams have been made popular by [DeM79], have appeared in different versions and have been extended in many ways in various other approaches. These include such object-oriented proposals as the OMT [RBP+94], and ROOM [SGW94]. The basic constituents of a data flow diagram are the nodes that denote *computational components*, such as *add*. These nodes are equipped with *input channels* and *output channels*, where the communication of data with the node's environment (e.g., other components) takes place. Channels may be typed and can be connected through directed *data-flow links* in a one-to-many-style, i.e., one output channel can be 'fed' into more than one input channel. Components without inputs act as constants, and without output they act as sinks. To make data-flow diagrams composable, special nodes

are used to describe the input and output channels of the overall diagram, leading to and from the environment of the component that is described by the diagram.

Textual languages are symbolic in spirit, and their basic syntactic expressions are put together in linear sequences. *Iconic languages* are those whose basic expressions are small pictorial signs that refer to the elements they visually depict. These languages are also called diagrammatic or graphical. An iconic language can be more intuitive than a textual one, but it can also be confusing if the icons are used in abundance. The reason is that such languages often use composition mechanisms taken from textual languages, such as linear — horizontal or vertical — proximity, which somehow decreases the language's visual appeal. More useful are visual/diagrammatic languages, in which topological and geometric notions are used: basic expressions would be, for example, lines, arrows, closed curves and boxes, and composition mechanisms would involve connectivity, exclusivity, and insideness. See e.g., [Har88]. Despite arguments against diagrams [Dij93, FPB87], such languages can be of great help in software development.

From a theoretical point of view, there is no principal difference between textual and diagrammatic languages. Unfortunately, practice proves it is much harder to cope with diagrammatic languages, especially when it comes to the need to be rigorous and formal, something we are going to return to later on.

We use the term *syntax* whenever we refer to the notation of the language, and this includes diagrams too. Syntactic issues focus purely on the notational aspects of the language, completely disregarding any meaning. The meaning of a language is described by its semantics. It is interesting to note that, in general, computerized tools do not allow us to manipulate semantics directly. Instead, everything we see and work with on the 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 "K = **read**()" is written in a language whose input commands are of the form "K = **input**()", chances are that the result will be some kind of *syntax error*. And of course, we cannot hope to address the computer with the like of "*please read a value for K from the input*," or "*how about getting me a value for K*." These might result in a long string of obscure error messages. It is true that nice, talkative instructions, such as the ones we find in recipes, are more pleasant than their terse and impersonal equivalents. It is also true that we strive to make computers as user-friendly as possible. But since we are still far from computers that can understand free-flowing natural language like English, a formal, concise, and rigid set of syntactic rules is essential.

An algorithm written in a typical programming language is given in Figure 2. The intended meaning of this program is "obvious": It calculates and prints the sum of all natural numbers up to the input K. Of course, this is what *authors intend* this program to mean, which is not enough. The computer (as well as other people) must have

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

Figure 2: Program in an ordinary programming language

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*, that assigns an unambiguous meaning to each syntactically allowed phrase in the programming language. Without this, the syntax is worthless. Otherwise, severe misinterpretations become possible, 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*". Who says that the used keywords **for**, **print**, or **read** have anything at all to do with their meaning in English? Maybe the program segment of Figure 2 means "erase the computer's entire memory, change the values of all variables to zero, output 'TO HELL WITH PROGRAMMING LANGUAGES,' and stop!" Who says that "=" stands for "assign to", and that "+" denotes addition? And on and on. We might be able to *guess* what is meant, since the language designer probably chose 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 computer engineering discipline, any language, textual or diagrammatic, must come complete with rigid rules that prescribe the allowed *form* of a syntactically well-formed program, and also with rules, just as rigid, that prescribe its *semantics*.

3 Semantics: The semantic domain

Expressions are what we use to communicate information. It would be nice if any two communicating participants interpreted expressions of the language in exactly the same way. Agreement on the meaning of a language is an important and partly sociological process, without which the communicated data is worthless.

A semantic definition for a language $\mathcal{L}$, or simply a *semantics*, consists of two parts: a *semantic domain*, which we denote generically by $\mathcal{S}_{\mathcal{L}}$, or simply $\mathcal{S}$ when there is no confusion, and a *semantic mapping* from the syntax to the semantic domain, denoted by $\mathcal{M}_{\mathcal{L}}$, or simply $\mathcal{M}$. Thus a "*language*" is composed as described in Figure 3

Let us explain. The semantics of a language tells us about the meaning of each of its expressions. That meaning must be an element in some well-defined domain.

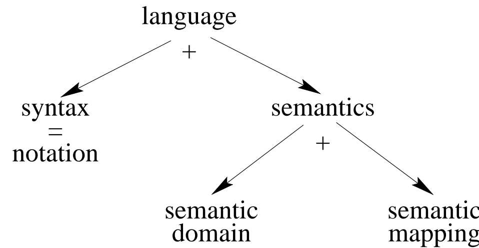

Figure 3: The structure of a language

For example, the meaning of an arithmetic expression in the language $\langle Exp \rangle$ would be a number. Thus we use $\mathcal{S} = \mathbb{N}$ as the semantic domain of $\langle Exp \rangle$. The semantic mapping would thus associate a number with each expression of the language: formally, $\mathcal{M} : \langle Exp \rangle \rightarrow \mathbb{N}$. Semantic mappings can often be defined in an inductive fashion, by providing the meaning of complex expressions of the language in terms of the meanings of simpler (already understood) expressions. This situation occurs not only when semantics is defined formally, but also when it is explained informally.

For the language of arithmetic expressions $\langle Exp \rangle$ defined earlier, it is quite natural to use standard mathematics as the basis of the mapping. For example, the obvious mapping of the symbol "+" is to the mathematical operation of addition. Indeed, while there is much sense in giving meanings that are consistent with accepted conventions, in principle we could give "+" any meaning we like. Thus, strange as it may seem, we might have specified that the expression "$x + y$" actually means the exponential x^y, or the binomial $\frac{x!}{y!(x-y)!}$.

For the case of data-flow diagrams it is less clear what the appropriate semantic domain should be. What properties are really described by a data-flow diagram? Are the structure and possible flow relationships between the computational components in the diagram the only important things, or are we trying to capture behavior too?

One common misconception in the world of system modeling languages is to take semantics as a synonym for behavior. Both the behavior and the structure of a system are important views thereof; both are represented by syntactic concepts and both need semantics. Although the behavioral aspects are usually less obvious and are much harder to define properly, languages focusing on structure only, such as ER-diagrams for databases or class diagrams in the UML, also need semantics, so that we know exactly what is being defined. And deciding upon the semantic domain amounts to deciding upon the kinds of things we want our language to express.

For example, the semantics of data-flow diagrams can be defined in more ways than one. We can use them to describe structure only, in which case the semantics would prescribe a "white box" view of the structure of each enclosing component, together with the data-flow links that show the channels through which information flows. This

allows a hierarchical decomposition of the system functionality, as in [SGW94], but nothing is said or meant about whether, when, or why data will actually flow as the system actually "behaves"; that is, as it runs, or executes, or progresses. In this case, the semantic domain will not refer to the behavior at all.

Alternatively, we might also want to describe actual behavior using data-flow diagrams, and then new questions arise. Are we talking about possible behavior only, i.e., about what *might* happen, or maybe also about what *will* happen? Does a computational component have a memory? Can it be nondeterministic and thus react in different ways to the same input? 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 in the communication lines between components for storing unprocessed messages, or are messages lost if unprocessed?

Different answers to such questions lead to a variety of quite different kinds of semantic domains for behavior: traces [Hoa85], trace trees [Maz86], input/output-relations [LT89], streams and stream processing functions [BDD+93], and many more.

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 usually sufficient to adopt a function from inputs to outputs as the semantic domain:

$$IO_{func} : I \rightarrow O$$

In the data-flow example of Figure 1 this would be $IO_{func} : \mathbb{N} \rightarrow \mathbb{N}$, defined by $IO_{func}(n) = \frac{n(n+1)}{2}$. However, we could also use a relation of input/output pairs:

$$IO_{rel} = \{(i, o) \mid i \in I, o \in O\}$$

This would mean that since several I/O-pairs with the same input may exist, we are not insisting on determinism. Extending these pairs to finite sequences (denoted as I^* and O^*) allows us to talk about history:

$$IO_{hist} = \{(i, o) \mid i \in I^*, o \in O^*\}$$

Here, to determine the latest output of such an input/output pair might require the entire input history, and not only the latest input. Another semantic domain could be the set of traces itself, where inputs and outputs are observed in an interleaved manner:

$$IO_{trace} = \{x \mid x \in (I \cup O)^*\}$$

If we want our components to respond to each input with exactly one output, but still determine output based on the component's history, we could use the semantic domain:

$$IO_{hist2} = \{(i,o) \mid i \in I^*, o \in O^*, len(i) = len(o)\}$$

As explained in [BDD$^+$93], using traces or relations as the semantic domain makes it difficult to properly define the composition of data-flow diagrams. To alleviate this, the stream processing function $IO_{stf} : I^* \to O^*$ of [BDD$^+$93] can be used instead. Using a mapping of input traces to output traces even allows to specify components with history. This is an example of making a subtle change in the semantic domain in order to improve the convenience of defining the semantic mapping for a given notation.

It is worth emphasizing that the semantic domain as the target of a mathematical function must contain all possible meanings of all syntactic expressions. Thus, if we want the semantics of a data-flow diagram to be an I/O function, the semantic domain is *set of all* such functions; if we want it to be an I/O relation, the semantic domain must be the *set of all* such relations. Formally, in these cases we need to define the semantic domain by $\mathcal{S} = \mathbb{P}(IO_{func})$ or $\mathcal{S} = \mathbb{P}(IO_{hist})$, where $\mathbb{P}$ denotes the powerset construction. (We sometimes write these without the parentheses, e.g., $\mathbb{P}IO_{func}$.)

The semantic domain is not to be taken lightly: it specifies the very concepts that exist in the universe of discourse. It is an abstraction of reality, describing the important aspects of the systems that we are interested in developing. It is also a prerequisite for comparing different semantic definitions. An explicit definition of the semantic domain is thus crucial. Although, the semantic domain is defined for describing the meaning of a notation, the definition of the semantic domain is normally independent of the notation. This allows to "reuse" the semantic domain for other notations.

How does one describe the semantic domain? Well, it can be done in varying degrees of formality, as the brief examples from the world of arithmetic expressions and data-flow diagrams illustrate. Jumping for a moment to the most complex example we shall be using, the UML, we note that defining its semantic domain is far from being a simple matter. It must definitely involve combinations of numerous kinds of elements, such as messages, states, values for variables, boolean values for conditions, etc. But there seems to be no obvious way to define this complex semantic domain, so that the result is precise, clear and readable. Whereas UML descriptions in the literature are very detailed when it comes to syntax, defining even the semantic domain is much more difficult and is usually done very informally, if at all. Sometimes even the best-written documents on the UML scatter the information about the semantic domain throughout the entire description.

Unfortunately, the confusion that often exists between syntax and semantics is made worse by the fact that we need a syntactic representation for the semantics itself! To properly define a semantic domain we need some kind of language too. This will also be discussed later on.

4 Semantics: The semantic mapping

Given a syntax $\mathcal{L}$ and a semantic domain $\mathcal{S}$, the final step in defining a semantics is to relate the syntactic concepts to those of the semantic domain. Each syntactic creature is mapped to some semantic element. As explained earlier, it is important to say explicitly and clearly that the syntactic operator "+" in any arithmetic expression is mapped to the addition operator of arithmetic, so that the meaning of the expression "$12 + 30$" will end up being the number 42, which is the sum of the two numbers. The reader should not underestimate the importance of defining this mapping, although this particular example might seem trivial.

Often the mapping is explained informally, by examples and in plain English. But regardless of the degree of formality of its representation, the semantic mapping itself should be a function from $\mathcal{L}$ to $\mathcal{S}$, i.e.,

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

Here are some ways of assigning semantics to the language of arithmetic expressions $\langle Exp \rangle$ and the data-flow diagrams defined above.

We have already chosen the semantic domain $\mathcal{S}_{\langle Exp \rangle} = \mathbb{N}$, so we should now define the semantic mapping $\mathcal{M} : \langle Exp \rangle \rightarrow \mathbb{N}$. We will be giving expressions in $\langle Exp \rangle$ their standard interpretation from arithmetic. The basic cases are arithmetic constants and variables. Hence $\mathcal{M}(\text{"42"}) = 42$. If we have been given a variable assignment $\phi : \langle Variable \rangle \rightarrow \mathbb{N}$ by the environment (meaning that we have been told the current value of each variable), we will adopt it, simply setting

$$\mathcal{M}(v) = \phi(v)$$

for each variable v.

Having handled the basic cases, we now can define the inductive cases, in which expressions contain simpler expressions. If an expression has the form a"+"b, with subexpressions $a, b \in \langle Exp \rangle$, then we define:

$$\mathcal{M}(a\text{"}\mathsf{+}\text{"}b) = \mathcal{M}(a) + \mathcal{M}(b)$$

To clarify, this definition maps the symbol "+" to the operation plus. (The observant reader will notice that we have even used a different font for the syntactic "+" of the language and the mathematical symbol +.) Obviously, seeing this kind of definition can irritate, because it looks so obvious. However, it is extremely important, especially for functions that don't have an obvious interpretation at all. In our case, the function `foo` has been made part of the syntax of $\langle Exp \rangle$, and it definitely needs to be defined. We choose the following:

$$\mathcal{M}(\text{"}\texttt{foo(}\text{"}a\text{","}b\text{")"}) = \mathcal{M}(a)^{\mathcal{M}(b)}$$

which identifies `foo` with exponentiation. Had we defined

$$\mathcal{M}(\text{``}\texttt{foo(}\text{''}a\text{``,''}b\text{``)''}) = \mathcal{M}(a) + \mathcal{M}(b)$$

`foo` would have been redundant, as it would be identical, as a function, to "+".

Turning to the data-flow example, since we have only used deterministic components, we may choose a deterministic history function to represent the behavior of a data-flow component. Let M^∞ be used to describe the finite and infinite sequences over a message set M.

We first define the basic components of a diagram, *add* and the 1-component. We want *add* to depict pointwise addition on sequences. Thus, we define the function F_{add} : $\mathbb{N}^\infty \times \mathbb{N}^\infty \rightarrow \mathbb{N}^\infty$, 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 is extended to infinite histories in the obvious way.

Note that this definition means that we have chosen to allow inputs on channels to arrive at different times, thus implicitly modeling buffers on the data-flow links. Furthermore, we do not talk about time explicitly, thus allowing the computation component to take time to process input. However, we have modeled *add* as a greedy component, since its semantics prescribes that sooner or later it must process its input without further stimulation.

We now define the semantics of the 1-component of a data-flow diagram as a constant, continuously emitting its number. Thus, we have $F_1 \in \mathbb{N}^\infty$, with

$$F_1 \;=\; 1^\infty$$

Having defined the basic semantics in such a way, we must now define the way the meaning of a composite diagram is derived from the meanings of its constituent parts. In our example, we compose the semantics according to the structure depicted in Figure 1:

$$F(n) \;=\; F_{div}(F_{mult}(n, F_{add}(n, F_1)), F_2).$$

Using elementary algebra, this can be seen to satisfy

$$F[n_1, n_2, \ldots] \;=\; \left[\frac{n_1(n_1+1)}{2},\; \frac{n_2(n_2+1)}{2},\; \ldots\right]$$

Thus, we have again seen a semantic definition for a syntactic construct, that is built from semantic definitions of the construct's constituents. When this is done in a general way, as we did for the "+" operation in an arithmetic expression, we call it a *compositional* semantics, as it allows us to compose the semantics; that is, the meaning of a composite creature based on the meanings of its parts, see [Old86].

Compositionality is highly desirable, and should be used even on an informal level, and even for actual code. Unfortunately, there can be subtle problems with pure black-box compositionality. The recent notion of components [Szy97] focuses on composition.

Let us discuss another graphical/diagrammatic language which is of widespread use — class diagrams, which we shall call $\langle CD \rangle$. This is the central language for specifying structure in object-oriented methods, and is, in particular, the main structure language in the UML. Class diagrams have been quite widely studied in an attempt to provide them with a precise semantics (e.g., [FBS+97, BHH+97, HSB99]). Despite these attempts, there are some subtle issues around $\langle CD \rangle$, upon which there is still no general agreement (for example, the precise differences between aggregation and composition). Without getting into the unresolved issues, we would like to illustrate how such a structural description language can receive its semantics.[1]

A sample expression of the language is depicted in Figure 4. This simple diagram contains three classes as boxes and three associations, which we regard as undirected even though the name indicates that there is a direction.

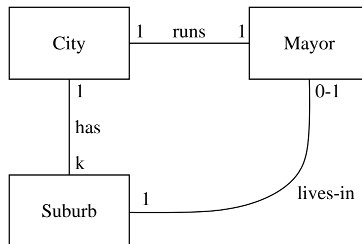

Figure 4: Sample class diagram

A proper semantic domain for class diagrams should contain snapshots of the running system, so that a diagram can be interpreted as the set of all possibilities of its object structures. Such a structure is a "frozen" situation at any given time during the system's execution. We thus use an object store containing three sets of objects, one for each of the classes in the example. As we would like class diagrams to be flexible and extensible, we do not use a fixed triple $(Obj_{City}, Obj_{Suburb}, Obj_{Mayor})$, or anything similar, but introduce a set of object stores OS, together with the following three retrieval functions for the objects of the respective classes:

$$city : OS \to \mathbb{P}Obj_{City}$$

$$suburb : OS \to \mathbb{P}Obj_{Suburb}$$

[1]Class diagrams do not talk explicitly about behavior; they are intended to focus on structural issues. However, composition (strong aggregation) and some other features do impose behavioral restrictions. Instead of dealing with these here, we will concentrate on the structural aspects and discuss the behavioral issues of $\langle CD \rangle$ in Part III of our series, the "Really Hard Stuff" part.

$$mayor : OS \to \mathbb{P}Obj_{Mayor}$$

This approach is rather abstract, as we need not say explicitly what the objects look like or whether they have a true identity. Of course, object identity can easily be added, obtaining a more detailed, implementation oriented semantics for class diagrams.

We use the same kind of abstract approach for the associations between objects, and introduce the following functions:

$$runs : OS \to \mathbb{P}(Obj_{Mayor} \times Obj_{City})$$

$$has : OS \to \mathbb{P}(Obj_{City} \times Obj_{Suburb})$$

$$lives\text{-}in : OS \to \mathbb{P}(Obj_{Major} \times Obj_{Suburb})$$

The abstractness here means, for example, that we need not decide whether the information about connections can be derived from the objects themselves (through stored identifiers) or from other elements in the store representing the associations.

The constraints on the associations now impose a number of restrictions on the allowed object stores. For example, if os is an arbitrary object store, we must require that each city has exactly one mayor running it:

$$\pi_2(runs(os)) = mayor(os)$$

(Here π_2 denotes projection on the second argument.) Formally, the language $\langle CD \rangle$ of class diagrams gets its meaning according to the function:

$$\mathcal{M}_{CD} : \langle CD \rangle \to \mathbb{P}(OS)$$

And we will have $os \in \mathcal{M}_{CD}(cd)$ if os satisfies all the restrictions imposed by the class diagram *cd*. Note that the functions *city*, *runs*, etc. play a dual role in this context: on the one hand, they are the representations of syntactic concepts in the context of the semantic domain. On the other hand, they are used as auxiliary functions to define the actual semantic mapping $\mathcal{M}_{CD}$.

5 Representation

All elements of a language definition: the syntax, the semantic domain and the semantic mapping, need a representation. Rigorous and readable mechanisms are necessary in order to define and represent these elements appropriately. In a later subsection we discuss the intended audience of these definitions, but for now we wish to concentrate on the issue of a satisfactory rigorous representation of them.

For conventional textual languages, the syntax is described by an employed set of characters — the *alphabet* — and the sequences of characters that are legal, i.e., those

we are allowed to use. We will typically first group characters into *words*, and then arrange words into *sentences* according to precise grammatical rules. The language then consists of the set of all these legal sentences. As stated earlier, in some ways there is no principal difference between textual and visual/diagrammatic languages. However, in the latter case it is far less easy to make out the words and sentences.

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

1. A set of characters forms an alphabet.

2. The characters are grouped into words, denoting keywords, numbers, delimiters, etc. This lexical layer is typically defined using regular expressions.

3. A third layer groups these words into sentences, usually by using a context free grammar.

4. A fourth and final layer constrains the sentences by imposing context conditions. (For example, requiring that variables are defined before they are used, or that their usage is consistent with their types.)

In compiler theory, the context conditions are often called "semantic conditions" as they are triggered by semantic considerations. However, they really just constrain the syntax and do not contribute to the definition of semantics. In modern languages a number of conditions are expressed as context conditions for convenience, even so they could be expressed in the context free grammar as well. A typical example is the priority scheme of infix operators.

For our language of arithmetic expressions, $\langle Exp \rangle$, we take the usual character set as the alphabet. Words include variable names, numbers, and delimiters such as "(". A complete sentence then is an expression such as "$(3 + a) - y$", definable using context free grammars. Using context conditions, we further restrict the set of well-formed sentences by disallowing the use of the special name "`foo`" as a variable or with other than two parameters. This makes expressions such as (3+`foo`) and `foo`(3) syntactically incorrect. It is important that the context conditions are decidable, as they have to be checkable by the parser.

As to diagrams, here there is a different way of viewing their syntax. We need not think of drawing a diagram as starting with lines or line-segments and then making boxes and arcs out of them. Rather, we have layers of topological notions, that are then 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 (the latter are just closed Jordan curves).

2. These are specialized geometrically to several kinds of lines (e.g., arrows, straight lines and splines, all with various line styles and colors, etc.), and closed shapes (boxes, circles, also 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 and intersection, and then arranging these geometrically into an actual 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. This textual part can be defined using a conventional textual grammar.

The general conclusion of this discussion is that in defining the syntax of a language $\mathcal{L}$ we need to use a notation already known, which for now we shall call $\mathcal{N}_{\mathcal{L}}$. For textual languages, $\mathcal{N}_{\mathcal{L}}$ will typically contain a combination of the Backus-Naur Form (BNF) and Chomsky-2 context free grammars (CH-2) (see the $\langle Exp \rangle$ language example above).

The use of a notation $\mathcal{N}_{\mathcal{L}}$ to define the syntax of $\mathcal{L}$ often results in more than just the definition of the syntax. As a side benefit, it can also provide an abstract version of $\mathcal{L}$, called the *abstract syntax tree*, and an algorithm for parsing the concrete into the abstract version. Let us now identify the language with its abstract version and discuss the representation of the semantics.

To define the semantic domain, we need again an underlying notation, $\mathcal{N}_{\mathcal{S}}$. The variety of notations used for this purpose is much larger than in the case of the syntax. Besides natural languages such as English, many general purpose formal languages can be used, such as Z [Spi88], algebraic specification languages [BFG+93, Wir90], or pure mathematics (see the semantic domain for the class diagrams in Example 4). Later we will discuss these possibilities and the implications of making a choice between them.

Above, we gave several mathematically defined versions of semantic domains for data-flow diagrams. We can easily rewrite these domains using a specification language like Z or an algebraic specification language, yielding quite different-looking representations of the same semantic domain. Thus, the semantic domain $\mathcal{S}$ and the notation $\mathcal{N}_{\mathcal{S}}$ used to describe it are rather independent.

As to the semantic mapping, the different kinds of $\mathcal{N}_{\mathcal{L}}$ notations used to describe syntax and the $\mathcal{N}_{\mathcal{S}}$ notations used for the semantic domain give rise to a great variety of ways to define the semantic mapping between the two. In many attempts to define semantics, the semantic mapping is given informally, e.g., by showing specific examples of a mapping from $\mathcal{L}$ to $\mathcal{S}$, without explicitly giving the mapping $\mathcal{M}$ itself. However, when $\mathcal{M}$ is to be given explicitly (and this is clearly the preferred way to do things), a notation is required for it too, call it $\mathcal{N}_{\mathcal{M}}$. While there is a variety of rigorous notations for syntax and semantic domain, there are not that many that are appropriate for the mapping. On the one hand, we can use pure mathematical notation [WB98, Rum96,

RK96, DF98], and on the other hand there is an approach called *graph transformations* [SW97, BCMR98]. Interestingly enough, there seems to be no approach that uses Z or some algebraic specification language to explicitly define the semantic mapping. This might be partly due to the fact that a notation for the mapping must somehow include the notations for the syntax and the semantic domains too. This works nicely for graph transformations if both domains are graph structures, so that $\mathcal{N_L}, \mathcal{N_S} \subseteq \mathcal{N_M}$, and it works well if standard mathematics is used, since all relevant elements can be dealt with within the generic mathematical framework. However, using Z or a similar language as $\mathcal{N_M}$ would require major additional work to model the syntax of the language $\mathcal{L}$ (which is essentially a context free language or a graph) within Z. Furthermore, the use of Z as the semantic domain $\mathcal{S}$ makes an explicit definition of the mapping $\mathcal{M}$ almost unthinkable.

6 Defining the UML

Even today, with the increasing popularity of graphical/diagrammatic languages, such as those that constitute the UML, it is unclear what is the best notation for describing them. Whereas for textual languages the use of grammars for the syntax is widely accepted, for diagrammatic languages there exist two competing approaches. On the one hand, we have graph grammars [Ehr79] that extend the grammatical ideas from textual languages to diagrams, and they have indeed been applied to significant parts of the UML already [SW97, BCMR98]. On the other hand, the very class diagrams of the UML can be used to model the abstract syntax of a diagrammatic language.

In fact, in the UML standardization documents [UML98] the latter technique, called *meta-modeling*, is applied in a "bootstrapping" fashion. The result is a meta-model that is essentially a class diagram. Although class diagrams are more intuitive than graph grammars, they are less expressive, and many properties of the syntax of the language have to be defined outside the class diagrams, as *context conditions*. In the UML documents many of these context conditions are defined using the Object Constraint Language (OCL) [WK98], which is also part of the UML. Other context conditions are stated in English. In textual languages, these constraints can be stated more precisely, e.g., by an appropriate attribution of the abstract syntax tree, which results from the parsing process via the context free grammar.

It is important to emphasize that whatever parts of the UML are defined using the meta-model, they describe only the syntax; there still remains the problem of defining the semantics. As mentioned earlier, context conditions are not "semantic conditions", as some people refer to them — they merely constrain the syntax. Context conditions are well-formedness rules (like the one that requires each variable to be defined before it is used, without telling us what a variable is and what its usages mean). Semantics is a synonym to "meaning". Just as C^{++} cannot be understood from its context free

grammar and its context conditions (without deep knowledge of similarly structured languages), so is the case for the UML. By constraining the syntax in a way that rules out problematic syntactic constructs, context conditions are a great aid in 'preparing' a language for a sound semantics. But they are not the semantics, and therefore terming them "semantic conditions" is misleading. In our framework, UML is the language $\mathcal{L}$ to be defined, and it must be defined in full: syntax, semantics and all. The part of the UML containing class diagrams and the OCL can be viewed as the notation $\mathcal{N}_\mathcal{L}$ used for defining the syntax, but that's all it is.

The recursive, bootstrapping nature of 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 meta-language to be able to see a good definition of the syntax. Using bootstrapping to describe the UML syntax can also be practical for newcomers, as they can identify a core part of UML to be adopted at first. In other realms there are similar situations of such definition, but it is very important to have a solid basis for this. In particular, class diagrams and the OCL must have also definitions using other techniques.

How is the semantics of the UML defined in the standardization documents? Well, the documents [UML98] do contain a part called the "Semantics of UML". However, it really does not focus on semantics but mainly describes the abstract syntax of the UML. The documents do contain many informal descriptions and insights into the semantics of the UML, and these might be sufficient for experienced users to gain more knowledge about the purpose of the constructs of UML. But they are far from being a satisfactory semantics, and many ambiguities remain. This has been illustrated in numerous papers dealing with packages [SW97], class diagrams [BLM97, FEL97], UML statecharts [BCMR98, WB98], and the integration of several kinds of diagrams [BHH+97, PR97].

7 The degree of formality

One misconception about formality is the belief that textual languages are *a priori* formal and diagrammatic ones are not. The myth that some people come away believing, when exposed to the notion of a "formal language", is that a formal language is a formal-*looking* language; that any language that contains lots of Greek letters and mathematical symbols is formal. This equality is false in both directions: there have been highly formal-looking languages that lack severely in true formality, and there are languages that don't look very formal at all, but are in fact as formal as anything. The degree of formality of a language is independent of its appearance. Natural languages are textual, but informal, and some visual languages are fully formal. Petri Nets and statecharts, for example, have a precisely defined syntax and semantics, even though

you don't see many strange-looking symbols therein. Obviously, due to the background of the kinds of people who deal with textual/algebraic languages and the fact that there is a less accepted theory for the definition of diagrammatic languages, there is a correlation between the mathematical appearance of a language and its degree of formality. Still, it is important to realize that "diagrammatic" and "informal" are by no means synonymous. We use the adjectives "formal" and "rigorous" to emphasize that the language has precise and unambiguous syntax and semantics definitions.

There is also another kind of precision relevant here, which is the degree to which expressions in the language make precise statements. Precision of the language, as opposed to its 'fuzzyness', depends on the degree of formality, of course; a language that is not defined in a sufficiently formal way cannot be precise. However, a language can be rigorous, yet make imprecise statements. For example, the term "*x is a number of about 100*" is fuzzy and imprecise. In fact, one can claim that it doesn't really exclude any numbers at all! We can make this statement precise by using a mathematical expression like $30 < x < 170$. This expression is given in a formal and rigorous language (mathematics), but the statement itself makes a fuzzy statement about x. A less fuzzy, but equally precise statement would be $99 < x < 101$ (see Figure 5).

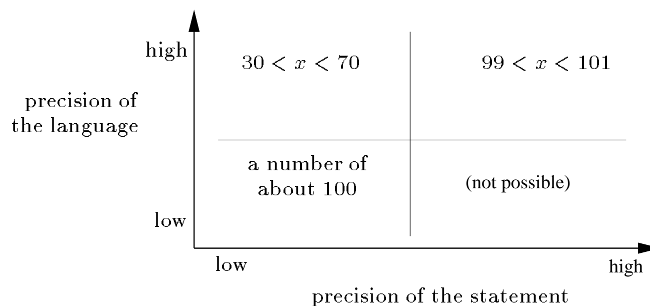

Figure 5: Precision of the language vs. fuzzyness of its statements

Carrying this example over to 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 (detailedness) of the model. In particular, we could describe systems with rather abstract and liberal UML models, even if the UML were to have a precisely defined syntax and semantics. Similarly, we can develop very detailed UML models even if we possess only a non-rigorous understanding of UML's actual meaning.

One of the main arguments against a formal foundation for diagrammatic languages arises from the confusion between these two concepts, i.e., equating abstraction of the models with the fuzziness of the language. One 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 problem, called *overspecification*, does not arise from

the formality of the language used, but from the failure on part of the developers to use the right abstractions. This is sometimes a consequence of the inability of the language or the tools implementing it to provide appropriate abstraction mechanisms, while incomplete models are allowed during system development. The definition of a single number, as in "*the number 100*", is precise, but it leaves us no freedom in implementation. Thus, having the right abstraction mechanisms at hand (in our example above, the use of intervals such as $99 < x < 101$) prevents overspecification.

Let's take a closer look at the degree of formality of semantic definitions. An explicit definition of the semantic mapping $\mathcal{M}$ allows us to reason about it. Some current approaches define $\mathcal{M}$ for a diagrammatic language in an effective algorithmic fashion. The idea behind this is to cater for the software engineer. It enables the software engineer to translate expressions of the language $\mathcal{L}$ into expressions of the semantic domain $\mathcal{S}$, e.g., in order to be able to use proof and analysis techniques on $\mathcal{S}$. To illustrate this, let us assume there exists an automatic checker for the predicate

$$consistent : \mathcal{S} \to Bool$$

that states that an element of $\mathcal{S}$ is consistent, so that a proper implementation of it can be found. The software engineer can apply the checker to expressions in the language, after applying the mapping from $\mathcal{L}$ to $\mathcal{S}$. A drawback of this approach is the need for engineers to be willing and able to understand not only the syntax, $\mathcal{L}$, but also the semantic notation $\mathcal{S}$. Typically, the engineer is not really interested in an explicit definition of the semantic domain $\mathcal{S}$. One of the reasons for this is that the semantic domain needs to be explained in yet another unfamiliar notation $\mathcal{N}_{\mathcal{S}}$. Instead, the engineer only wants to deal with $\mathcal{L}$. It would be better if the definer of the semantics would prove once and for all that for all expressions $e \in \mathcal{L}$ the resulting semantics is consistent and therefore implementable. In other words, that inconsistent expressions have already been eliminated by a careful definition of the language itself.

For example, when compiling a higher level programming language $\mathcal{L}$, the consistency checker (as part of the compiler) typically ensures that the result is correct and consistent. Consistency means that the result is either directly executable opcode or it can be translated to opcode without further possible errors. The well-formedness conditions of $\mathcal{L}$ should be defined in such a way that the software engineer using the language $\mathcal{L}$ can be sure to have consistent expressions without being explicitly exposed to the formally given semantic domain. In order to successfully deal with the provided techniques and tools, one would then need only an informal understanding of the semantic domain.

The "careful definition" alluded to above is necessary for such a consistency proof to be feasible, and typically context conditions constraining the language are required. If the language is defined in this way, the syntax $\mathcal{L}$ would then be restricted to the set

of consistent documents, e.g. by a predicate

$$consistent_{\mathcal{L}} : \mathcal{L} \to Bool$$

with the condition that for all syntactically well-formed expressions $e \in \mathcal{L}$:

$$consistent_{\mathcal{L}}(e) \implies consistent(\mathcal{M}(e))$$

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. Ideally, the insights are obtained by the people who provided the semantic definition and not the users of language. Some questions that are relevant when defining the semantics are:

1. Does the given formalization capture the intuition of the users?
2. Are the context conditions sufficient to ensure consistency?
3. Does the notation allow the specification of important properties of the semantic domain?
4. If analysis techniques or transformation rules for the language exist, are they sound with respect to the given semantics?

A tremendous amount of work is necessary so that a semantics can properly address these and related questions, but it surely must be done for any serious language definition effort. A necessary prerequisite for success with respect to the forth question is an explicit definition of the semantic mapping $\mathcal{M}$. Other questions, like the first, also aim at a consensus between users. This can only be achieved by a broadly accepted standardization, based on a clear and complete formalization of both syntax and semantics.

8 The doodling phenomenon

There is another point worth making here, regarding visual/diagrammatic languages. Unfortunately, many people take diagrams lightly — much too lightly. They find it hard to relate to a bunch of graphics as something serious enough to be called a language, and profound enough to be the 'real thing'. Perhaps this is a result of the failure of flowcharts to replace conventional high-level programs (and we will not get into the reasons for this failure).

In any case, all too often one encounters what we may call the 'doodling phenomenon', whereby diagrams are considered by people to be what an engineer scribbles down on the side, as a kind of visual aid. The real work, the rationalization goes, has

to be done in textual or symbolic languages. Sadly, for a long time this view was held by many language designers and methodologists too. Many years ago, Martin and McClure published a book with the title *Diagramming Techniques for Analysis and Programmers* [MM85]. It was full of scores of diagrammatic notations, but made little attempt to convince the reader that the diagrams needed rigorous definitions. At best, readers came away with a slew of graphical notations for making little side pictures of things they were working on using a 'real' language. This might be helpful, but it works against the aim of exploiting the virtues of true visual formalisms.

Some people find it hard to understand why you can't simply add more and more graphical notations to a visual language. For example, there have been many cases of people proposing (in private communication) all kinds of extensions and additions to the language of statecharts. These people could not understand why you can't just add a new kind of arrow that "means synchronization", or a new kind of box that "means separate-thread concurrency" (these are actual quotes from such proposals). It seemed to them that if you have boxes and lines and they mean things, you can add more 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 having overlapping states in statecharts. This was proposed, in a very preliminary way, as a possible area for further work in the original statecharts paper [Har87], but it took a lot of hard work to figure out a consistent syntax and semantics for such an extension [HK92]. In fact, the result turned out to be too complex to justify implementation. Nevertheless, people often ask why we don't allow overlapping in systems implementing statecharts, such as Statemate or Rhapsody. It is very hard to convince them that it is not at all simple. One person 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 though the only thing that needs to be done is to remove the error message and you are in business! This person definitely had doodling in mind.

There are other serious difficulties people have when relating to the issue of designing visual languages. One is non-compositionality, which we have already discussed somewhat, and the other — perhaps the most severe — is the temptation to provide a variety of views. We shall discuss views in the second part of our series.

9 The intended audience

The intended audience of the semantic definition of a language is an important consideration when selecting the right representation. 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. This is definitely true if the user does not wish to have a formal definition but only an intuitive, appealing description of the language's purposes. As stated earlier, typical

users will not be willing to make an effort to understand the semantic domain $\mathcal{S}$ given in a notation $\mathcal{N}_{\mathcal{S}}$ in which they are not trained. To begin with, they need to understand the notation $\mathcal{N}_{\mathcal{S}}$, which itself is defined using yet another formal language. Even if the user has skills in formal methods or mathematics, certain resistance to learning $\mathcal{N}_{\mathcal{S}}$ usually remains.

Since there is no semantic formalism that is commonly understood by a broad range of users, it is probably best to use a natural language for explaining the notation and carefully describing the semantics. To learn Japanese, many people need a translation into English, which is the language they know. A given relationship between English and Japanese helps them understand Japanese, but a translation from Japanese to German is of no use, as it would be necessary to learn German before understanding Japanese. In the same way, the audience of a semantic definition needs to understand the semantics in order to benefit from it.

In contrast to a user, a language developer (who ideally also features as the definer of the semantics) would be willing to cope with the notation $\mathcal{N}_{\mathcal{S}}$ to gather insights for defining the modeling language $\mathcal{L}$. The same goes for a methodologist, who has to possess perfect knowledge of the languages about which he/she makes detailed recommendations to users.

Tool vendors should also be exposed to a precise semantics, but it is probably better to give them detailed descriptions of the "how to deal with" instead of the "why". This would include rules for adding, removing and adapting elements of the notation, as, e.g., promoted in [BCMR98], or refinement and transformation calculi as given, e.g., in [Rum96, OJ93]. Tool vendors are often less interested in "what" a notation means, but in "how" its symbols can be modified, and "how" they can generate code out of it that is faithful to the original semantics. However, tool vendors should not forget that these issues depend on the basic availability of a rigorous, formal and commonly agreed semantics.

An example of how a complicated semantics can be described in a way that is not too frightening, yet conveys the rigor of a full definition, appears in [HN96]. This paper contains an operational-style semantic definition of the full statecharts language in a SA/SD (functional decomposition) framework, as implemented in the Statemate system. It can be comprehended by tool vendors and methodologists, and also by various kinds of users.

10 Extensibility

Until now, we have implicitly assumed that our languages are fixed in advance. This is seldom the case in practice. Most existing languages have a built in extension mechanism that allows the extension of the used vocabulary. The most prominent examples are the natural languages, which through the mechanism of explanation or

natural growth of the subject matter conveyed with the language, constantly allow extensions. An example of natural language extensibility can be found is this very paper, where we introduce terms like *syntax* and *semantics* by explaining what we mean by them.

Other examples are programming languages, where the main concern is to introduce new procedures, methods, types or classes. These are named, and are then used as new abstractions.

A language extension must always be explained in terms of the language itself. This allows the communication partner to understand the new concept. Thus, an extension gets its semantics indirectly through the semantics of the explanation's semantics. One common mechanism of extension relies on binding a subexpression of the language to a name, where the subexpression has already been given a semantics. If the new name is already used within the subexpression that defines the same name, the result is *recursion*. Recursion is rather common in defining functions and procedures, as well as types and class structures (whole-part, lists, etc.). Although recursion by its very nature is circular-looking, the classical theory of fixpoints [Kle52, Tar55] has solved its definition problem, allowing us to give a precise semantics for recursive definitions that fits the intuitive understanding we have of them.

Back to our main example, the UML, which has quite a number of mechanisms for introducing new elements. Besides the introduction of classes, methods and other so-called *first class citizens*, UML allows us to specialize the meaning of certain elements through stereotypes and tagged values. Unfortunately, UML does not offer any mechanism to describe the meaning of a newly introduced stereotype or tagged value within the language itself. Instead, informal English definitions are frequently used, or even definitions that appeal to specific tools. UML in its current version thus has a serious drawback with regards to extensions of this kind.

11 Summary

Syntax is the 'front window' of the language. It contains everything the user of a notation deals with. *Context conditions* are syntactic constraints that further restrict the syntax.

Beyond a precisely defined syntax, each language requires an unambiguously defined meaning, the *semantics*. There is no principal difference between the use of textual or visual/diagrammatic languages. Both kinds require a precisely defined syntax, describing the set of possible expressions in the language, and a similarly precise semantics, determining the meaning of these expressions. Semantics can be described in a variety of ways, depending on the purpose and the intended audience. The semantics of a language is usually described in two parts: the *semantic domain*, that provides the information on what we are talking about, and the *semantic mapping*, that maps the

context condition constrains the syntax; it describes the set of well-formed expressions of a language.

expression is a meaningful, well-formed element of a language; the following are synonyms: word, statement, sentence, document, diagram, model, term, piece of data, clause, module.

interpretation extracts the information from a piece of data; it is a mapping of data to a semantic domain.

language is a possibly infinite set of expressions used to communicate; it is a synonym to notation; a language allows us to syntactically represent information.

modeling language is used for specifying and documenting properties of a system in different abstractions and from different points of view.

notation is a syntactic representation of information; a synonym to language; what the user deals with.

programming language is used for programming software systems.

semantics defines the meaning of a notation: what information do the expressions in the notation describe.

semantic domain is a well understood domain of elements; elements of the semantic domain describe the important properties of what we are trying to define using a language (in our context, this means software and hardware systems and components of such systems).

semantic mapping is a mapping that relates each syntactic construct to a construct of the semantic domain; it usually explains new constructs in terms of known constructs.

sub-language is a subset of the syntactic elements, together with an appropriate adaptation/projection.

textual language is a language consisting of linear strings of characters and symbols (words, sentences, etc.).

visual/diagrammatic language is a language based mainly on graphic (topological/geometric) elements; it can employ textual elements too.

visual formalism is a diagrammatic language that has formal syntax and semantics.

Figure 6: Glossary

syntax into the semantic domain. The semantic domain can be defined independently of the syntax: in fact, we can use completely different languages for describing the same kinds of systems, so that these languages might all use the same semantic domain.

All three elements, the syntax $\mathcal{L}$, the semantic domain $\mathcal{S}$, and the semantic mapping $\mathcal{M} : \mathcal{L} \to \mathcal{S}$, need to be denoted by appropriate notations. The careful choice of these notations is crucial for gaining useful results from a semantic definition. Formality of a semantic definition depends on the way the syntax, the semantic domain and the semantic mapping are represented. A fully formal semantic definition includes an explicit formal definition of the semantic mapping. Defining the semantic mapping by examples only does not make a satisfactory semantics. For example, it does not allow analysis of the mapping itself to gain a better understanding thereof and better implementations in tools.

Figure 6 contains a short glossary.

References

[BCMR98] Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors. *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universitaet Muenchen, TUM-I9803, April 1998.

[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.

[BFG+93] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hußmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirements and Design Specification Language SPECTRUM, An Informal Introduction, Version 1.0, Part 1. Technical Report TUM-I9312, Technische Universität München, 1993.

[BHH+97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a Formalization of the Unified Modeling Language. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 Proceedings*. Springer-Verlag, LNCS 1241, 1997.

[BLM97] J. Bicarregui, K. Lano, and T. Maibaum. Objects, associations and subsystems: A heirarchical approach to encapsulation. In *Proceedings of ECOOP 97, LNCS 1489*. Springer-Verlag, 1997.

[BM99] J. Bézivin and P.-A. Muller. *Proceedings of the Unified Modeling Language Conference. ⟨⟨UML'98⟩⟩: Beyond the Notation*. Lecture Notes in Computer Science 1618. Springer Verlag, 1999. Mulhouse, France.

[DeM79] Tom DeMarco. *Structured Analysis and System Specification*. Yourdon Press, Englewood Cliffs, 1979.

[DF98] R. Diaconescu and K. Futatsugi. Logical Semantics of CafeOBJ. In: PSMT - Workshop on Precise Semantics for Software Modeling Techniques. Technical Report TUM-I9803, Technische Universität München, 1998.

[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.

[FBS+97] R. France, J.-M. Bruel, M. Saksena, E. Grant, and M. Larrondo-Petrie. Towards a rigorous object-oriented analysis and design method. In IEEE Computer Society Press, editor, *Proceedings of the 1st ICFEM International Conference on Formal Engineering Methods*, 1997.

[FEL97] R. France, A. Evans, and K. Lano. OOPSLA'97 Workshop on OO Behavioral Semantics. Technical Report TUM-I9737, Technische Univerität München, 1997.

[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.

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

[Hoa85] A. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[HSB99] Brian Henderson-Sellers and Frank Barbier. *Black and White Diamonds*. LNCS 1723. Springer Verlag, 1999.

[Kle52] S. Kleene. *Introduction to Metamathematics*. Van Nostrand, 1952.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Maz86] Antoni Mazurkiewicz. Trace Theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Application and Relationship to Other Models of Conucrrency*, pages 279–304. Springer, 1986. LNCS 255.

[MM85] J. Martin and C. McClure. *Diagramming Techniques for Analysis and Programmers*. Prentice Hall, 1985.

[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.

[PR97] J. Philipps and B. Rumpe. Refinement of information flow architectures. In M. Hinchey, editor, *ICFEM'97*. IEEE CS Press, 1997.

[RBP+94] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1994.

[RK96] B. Rumpe and C. Klein. *Automata with Output as Description of Object Behavior*, pages 265–286. Kluwer Academic Publishers, Norwell, Massachusetts, 1996.

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

[SGW94] B. Selic, G. Gulkeson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[Spi88] J. Spivey. *Understanding Z*. Cambridge University Press, 1988.

[SW97] A. Schürr and A.J. Winter. Formal Definition and Refinement of UML's Module/Package Concept. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology — ECOOP'97 Workshop Reader*, volume 1357 of

Lecture Notes in Computer Science, pages 211–215, Berlin, 1997. Springer Verlag.

[Szy97] Clemens Szypersky. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1997.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[UML98] Taskforce UML. Unified modeling language. Version 1.3, OMG, 1998.

[WB98] R. Wieringa and J. Broersen. Minimal Transition System Semantics for Lightweight Class and Behavior Diagrams. Technical Report TUM-I9803, Technische Universität München, 1998.

[Wir90] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 675–788. Elsevier Science Publishers B. V., 1990.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison Wesley, Reading, Mass., 1998.