SYSTEM DESCRIPTION METHODOLOGIES
D. Teichroew and G. David (Editors)
Elsevier Science Publishers B.V. (North-Holland)

A NOTE ON ABSTRACTION IN SOFTWARE DESCRIPTIONS

Jochen Ludewig

Computer Science Group
Brown Boveri Research Center
CH-5405 Baden, Switzerland

In the process of software development, abstraction is usually treated as a mere change of scale. Therefore, the only widely used principle for changing the level of abstraction is a change in quantity, for instance by stepwise refinement. This paper is based on the observation that there are in fact more differences between descriptions at different levels. Two consequences are suggested: first, an extended Entity-Relationship-Model, and second, a set of abstraction levels, each related to a specific "filter" through which the system is seen.
The paper is prefaced by a remark on terminology and by some information on my background.

A COMMENT ON TERMINOLOGY

Instead of an introduction, I would like to issue a comment on terminology. I think that some of the discussions of previous papers suffered from confusion about a few fundamental terms: language, tool, method. A scheme which I call the system triangle (figure 1) once helped me to make a clear distinction; maybe others find it useful too.

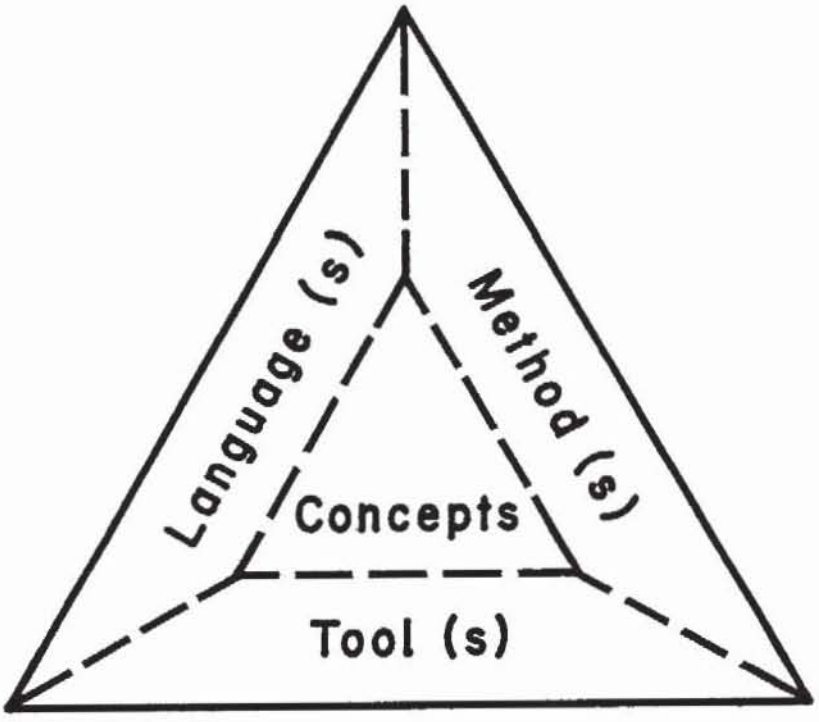

Figure 1: The System Triangle

Every description system, for instance a specification system or a programming system, consists of four components:

- Its heart is a set of abstract concepts, the semantic kernel. At the level of code, such concepts are e.g. loops and subprograms.

- For using and expressing the concepts, there is a language. Here, the language is just a syntax for the semantics conveyed by the concepts. Obviously, graphical representations like Nassi-Shneiderman-diagrams or Structure charts are languages just as good as conventional ones. For a certain set of concepts, several languages may exist.

- Writing a description (specification, code) requires some method, which again is related to the concepts. The strategy of stepwise refinement, e.g., is supported by the concept of subprograms.

- Tools are used to process descriptions, e.g. to check and store specifications, or to compile code.

The message of this figure is that language(s), method, and tool(s) are closely coupled. Their abstract kernel is the set of underlying concepts. These four terms can (and should) be well distinguished. (As Lehman stated, "methodology" in the place of "method" is simply wrong).

I believe that our inability to agree on a small, but sufficient set of concepts is at least one reason of current stagnation of research on software specification.

BACKGROUND

My personal background is six years of research in software specification systems, in particular for process control software. This work included experience with PSL/PSA, modification of PSL for process control [1], and design and implementation of a specification system named ESPRESO [2]. Most of this work was carried out at Nuclear Research Center, Karlsruhe, FRG.

Currently, our software engineering project at Brown Boveri Research Center, Baden, Switzerland, is working on the design of a Software Engineering System for process control software development and maintenance. This work, which is still in an early phase, is based on the idea that such an environment should provide all components of figure 1, i.e. method, languages, and a set of tools. The tools should not require the user to get aquainted with many different interfaces both for human computer interaction and for the communication between different tools. Therefore, two components should be used by all other tools: a universal man machine interface, and a general data base for program development.

Another important part of my background are EWICS TC 11 (European Workshop on Industrial Computers, Technical Committee on Application Oriented Specifications) and the Working Group on Ada for Specification. Both activities are supported partially by the European Community.

GENERALITY IN SPECIFICATION LANGUAGES: AN EXTENDED ENTITY RELATIONSHIP MODEL

A few specification languages, like HOS, are fully defined, i.e. their definition covers both syntax and semantics. Most others, however, are missing such a definition, but are more popular, apparently because they are less difficult to use.

Striving for specifications which are both clear and usable, we must try to discover the reasons for this dichotomy. Conclusions like "People just don't like formality" are not satisfactory, because programming (i.e. coding) languages actually are (or could be) fully defined without any disadvantage to the user. Therefore, it might be useful to look for other reasons.

A key to the problem may be that a specification should represent our ideas about a system as they are, not a transformation of these ideas into some formal world which is fine for the computer, but not for the human writer and reader. Our ideas are vague; hence, the specification must allow for vagueness.

In programs, there is one special way to express vagueness: if a procedure call refers to a procedure which is not yet designed, this detail of the program remains undefined. This is the basic idea of stepwise refinement; at first, only the most general part of the program is implemented, details are added as the implementation proceeds. In the following, this property of incomplete programs is called generality; a general description is incomplete, it may be refined in different ways.

The generality offered by undefined procedures is, however, very limited; there is information which can be given only at the level of full detail. If you know that two processes communicate via some data-object whose type is not yet defined, you cannot express that in a programming language, because you can neither express "communicate" nor "data-object". Note that this does not mean that the meaning of those two terms is undefined. "data-object" could e.g. be an object of one type from a given set of types. "communicate" could mean that either of the processes receives data from the other one. In real life problems, such situations occur again and again.

Languages like PROLOG [3] allow for such statements. See the following description:

```
data_object (X) :- integer (X).
data_object (X) :- real    (X).
data_object (X) :- boolean (X).
communicate (X, Y) :- writes (X, Z), reads (Y, Z).
communicate (X, Y) :- writes (Y, Z), reads (X, Z).
```

The predicates "data_object" and "communicate" are true if and only if their arguments meet any of the alternative conditions behind the hyphen.

Many approaches for system description are based on the Entity Relationship Model [4]. The basic principle, as used e.g. in PSL/PSA, is as follows: The system is described by a bipartite graph. Everyone of its nodes belongs either to the set of entities (or objects), or to the set of relationships (or links). Both objects and links are further classified by their types (object-types, link-types).

Every edge connects a link to an object. Edges are labelled; the number and labelling of the edges connected to a certain link are defined by its type. Objects are identified by a name; links are identified by their type and by the objects they are connected to.

Let there be a link-type "contains", and three object-types "module", "data", and "procedure". Every link of type "contains" is connected by two edges; the first one, labelled "owner", goes to an object which must be of type "module", the other one, labelled "property", goes to an object whose type may be either "data" or "procedure" (see figure 2). The objects are identified by their names ABC and XYZ, while the link is identified only by its type and by the ordered set of connected objects (owner = ABC, property = XYZ).

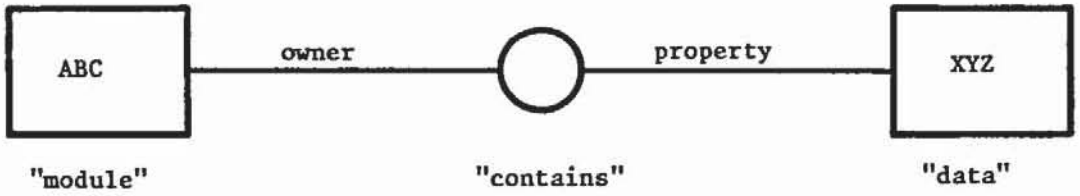

Figure 2: Two objects connected to a link

The Entity Relationship Model could gain much of the flexibility of PROLOG if the principle of generalization could be applied to object-types and link- types. The user would then be able to express his current view of the target system at various levels of generality. He might for instance introduce an object as being a data_object, thus postponing the decision about its type, or he could express the intent to have a communication link between two instances without choosing its direction. When the information is refined later on, the general information becomes obsolete, and is discarded.

Consistency rules for the system description are much more concise if they may refer to generalized types. In the example given above, object-types "module", "data" and "procedure" can be condensed in a type "mod-dat-proc". Then, the property-component of the contains-link has to be of this type. If the link refers to an undefined object, its type is set to "mod-dat-proc", until it is further defined, either explicitely or implicitely.

In the framework of our software engineering project, we are currently working on an adequate way to define types, generalization, and consistency rules. This definition is to be used by the data base generator. Consistency checking and control of generalized types can then be shifted to the data base management system, which should do it very efficiently.

SOME COMMENTS ON THE PRINCIPLE OF ZOOMING

Most of the work related to software description is based on the assumption that the descriptions at different levels of abstraction differ only by scale, much in the same way as photos differ which have been taken from a single position, but using a zoom lens. Good examples of this zooming principle can be found in papers on SADT (see figure 3).

Thinking about the photographic analogy, one can see that no good photographer will deliberately work in such a way, simply because a macro photo, a portrait, a landscape and all the other types of pictures differ not only in scale, but also in the viewpoint, filtering, lighting, and photographic material. The lesson we might learn from them is that we, too, need different concepts for desribing software at different levels of abstraction.

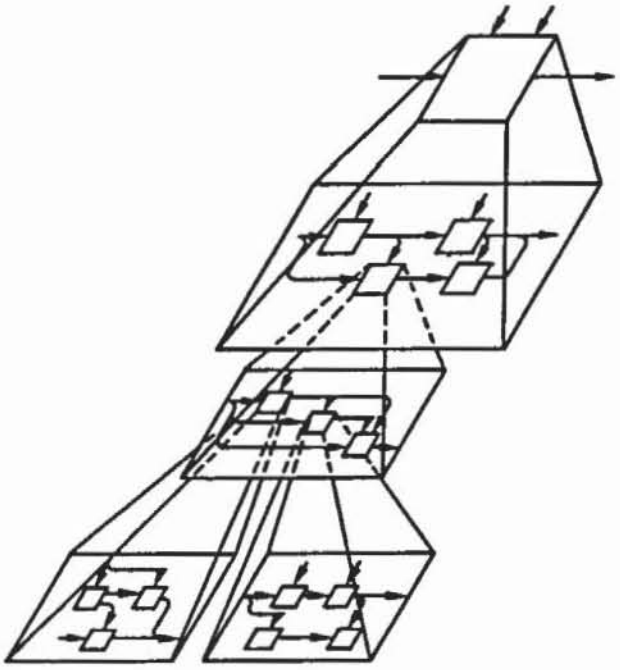

Figure 3: Zooming concept of SADT (from [5])

Our idea of a large and complex system, say a manufacturing plant, is static. We usually think of it as a object which is in a stable state. If we know a bit more about it, we think of its purpose, i.e. about its essential inputs and outputs. This, again, is a static view; we say "This factory produces generators," even though the products may be finished only at intervals of several weeks. Note that we do not mention side effects like consumption of energy, production of sewage, etc.

If we are interested to learn more about the system, we decompose it into subsystems and look at those, one by one. (Obviously, there are many ways to decompose the system, but this problem is not addressed here.) The less complex the subsystems we deal with, the more we tend to regard them as dynamic objects, which perform some functions. Simple tools in the factory are described by their function: "This machine smooths the surfaces of turbine blades." At the lowest level, we relate the function to time instances: "After every cycle, this scraper removes the waste."

The same basic principles apply to descriptions of other systems in general and of software systems in particular. We are unable to deal with a complex functionality; thus, it is either ignored (as in the case of the manufacturing plant), or it is greatly simplified. Time does not play any role in such descriptions. The functionality of less complex systems is more easily understood. Time (or sequence of execution) is introduced only at the lowest levels.

It should be possible to define a set of such "views", or "lightings", which could guide us through the process of software development. Table 1 shows a very tentative set of such levels. Maybe that we need different constructs for describing the target system at various levels, or the very same language is used at all levels, but a clever tool filters from the full description just the information related to the level at which we want to work.

At the Level of	the description deals with
SYSTEM	a large, complex system
UNITS	subsystems, similar to a system
TASKS	units, including their basic purpose
ACTIONS	tasks, including the conditions under which they are performed
CONTOURS	the precise interface between actions
CODE	the implementation

Table 1: Very tentative set of abstraction levels

This idea was first presented at EWICS TC 11, where it is currently being worked out [6].

CONCLUSION

Abstraction is an important principle in system description, but current tools and methods are based on a too limited understanding of abstraction. The two ideas presented above indicate possible extensions. They should not be taken as results, but as a starting point for further work and discussion.

REFERENCES

[1] Ludewig, J., Process control software specification in PCSL, in: Haase, V. (ed.), IFAC/IFIP workshop on real-time programming (Pergamon Press, Oxford etc., 1980) pp.103-108.

[2] Ludewig, J., ESPRESO - A system for process control software specification. 7th Conf. on operating systems, Visegrad, Hungary, January 1982; IEEE Transactions on Software Engineering, SE-9 (1983), July-issue.

[3] Clocksin, W.F. and Mellish, C.S., Programming in PROLOG (Springer Verlag, Berlin, Heidelberg, New York, 1981).

[4] Chen, P.P.-S., The Entity-Relationship Model - toward a unified view of data. ACM Transactions on Data Base Systems, 1 (1976), 1, 9-36.

[5] Ross, D.T., Structured analysis (SA): A language for communicating ideas, IEEE Transactions on Software Engineering, SE-3 (1977) 16-34.

[6] Jones, J., Kramer, J., and Ludewig, J., Abstraction levels in system descriptions - an intuitive approach, Contribution to the status report of EWICS TC 11, to appear ca. November 1983.