Requirements have an inherent logical structure that must be detected and codified to obtain a good system design—one that reflects the process it controls.

Computer-Aided Specification of Process Control Systems

Jochen Ludewig, Brown Boveri Research Center

Whatever the approach, it has now been recognized that the analysis must be disciplined and structured . . .

M. M. Lehman[1]

After a decade of research and experimentation, the importance of requirements specification (whatever that means) is generally recognized, and the number of journals and conferences on this subject is growing accordingly.[2] But results are not yet consolidated, and practitioners looking for tools and methods available as turnkey systems will be frustrated. This article is an attempt to present some experiences, ideas, and suggestions to those who are interested but may be confused, rather than motivated, by the amount of available material.

Many definitions have been suggested for terms like "requirement," "design," "specification," "document," which are often used in this field. For this article, two simple conventions will be sufficient: A specification is a description of an object, giving its properties of interest; it usually implies that the description should be precise, testable, and formal.[3] A requirement is a property of a product regarded as necessary.[3] "Specification" stands for "requirement specification," where suggested by the context.

How do we grow programs?

A program is—usually—a product. That means it has to be planned (specified, designed, and developed) and manufactured. Manufacturing in the sense of hardware products like cars or pencils is almost negligible. Copying a tape or a deck of cards is no problem. But we still do not know how to *plan* a program.

I have sometimes used the pillars in a cavern as a metaphor for programs (see Figure 1). The ceiling corresponds to the wishes and intentions of the ultimate user, while the ground is the binary code finally loaded into the machine. The drops of water saturated with lime stand for our effort at building a solid connection between top and bottom. The pillar cannot be cast from a mold in one step, because programs are made from tiny drops of information that are not available like concrete.

Making a new pillar is no longer as difficult as it used to be. Assemblers and compilers have been provided for problem-oriented languages, and work is underway on transformational systems which will guarantee that operational and efficient programs can be derived from complete and formal, possibly nonoperational specifications.[4] These "bricks" are making production of the highest possible stalagmite an easier task.

The stalactite, on the other hand, is still made in the old way, and little is being done to shorten the gap from the top end. The reasons are obvious: Working on the stalactite means working without the formalisms available on solid ground. As long as a specification is admittedly neither formal nor complete, formal proofs and transformations are hard to use.

Though we cannot use bricks up there, we can at least try to control the drops to concentrate the effort. One way of guiding them is to restrict the language used to state the specification. We can also use a combination of formal and natural languages; such an approach will be sketched in this article.

Life-cycle models

Life-cycle models, though often cited or modified, did not contribute too much to the knowledge in the field of requirement specifications.[5] The reason may be that this is a management concept, so it is intended to answer questions about schedules and deadlines, not about activities of the staff and contents of documents. Originally, the life cycle was nothing but a scaled time axis. But activities cannot be nicely arranged on this axis, one after the other. So the model was tuned to allow for iteration of previous steps (see Figure 2).

0018-9162/82/0500-0012$00.75

Figure 1. The program as a pillar.

Maybe we had better keep that straight time axis, so it can at least serve its limited purpose, and add new pictures for other aspects, such as the hierarchy between top (most complex operations) and bottom (simplest operations), or the hierarchy between requirements specification (what is visible at the user's interface) and implementation (what is hidden). Though all three aspects are highly connected, they are not at all identical. That is what the overburdened mixed life cycle can teach us. To paraphrase Parnas,[7] it would be nice if the next person to introduce a new life-cycle model in a paper would advance arguments for it.

The unbiased requirements specification: A utopia

Many specifications describe the implementation envisaged by the analyst rather than the requirements. That is unfair, because he not only fails to do his job but also limits the designer's freedom. It was Parnas who first stated that we should define *what* the program is requested to do before we start to think about *how* these requirements can be fulfilled.[8] Meanwhile, that rule became generally accepted (at least by those who talk about specification), and most papers on this topic—for

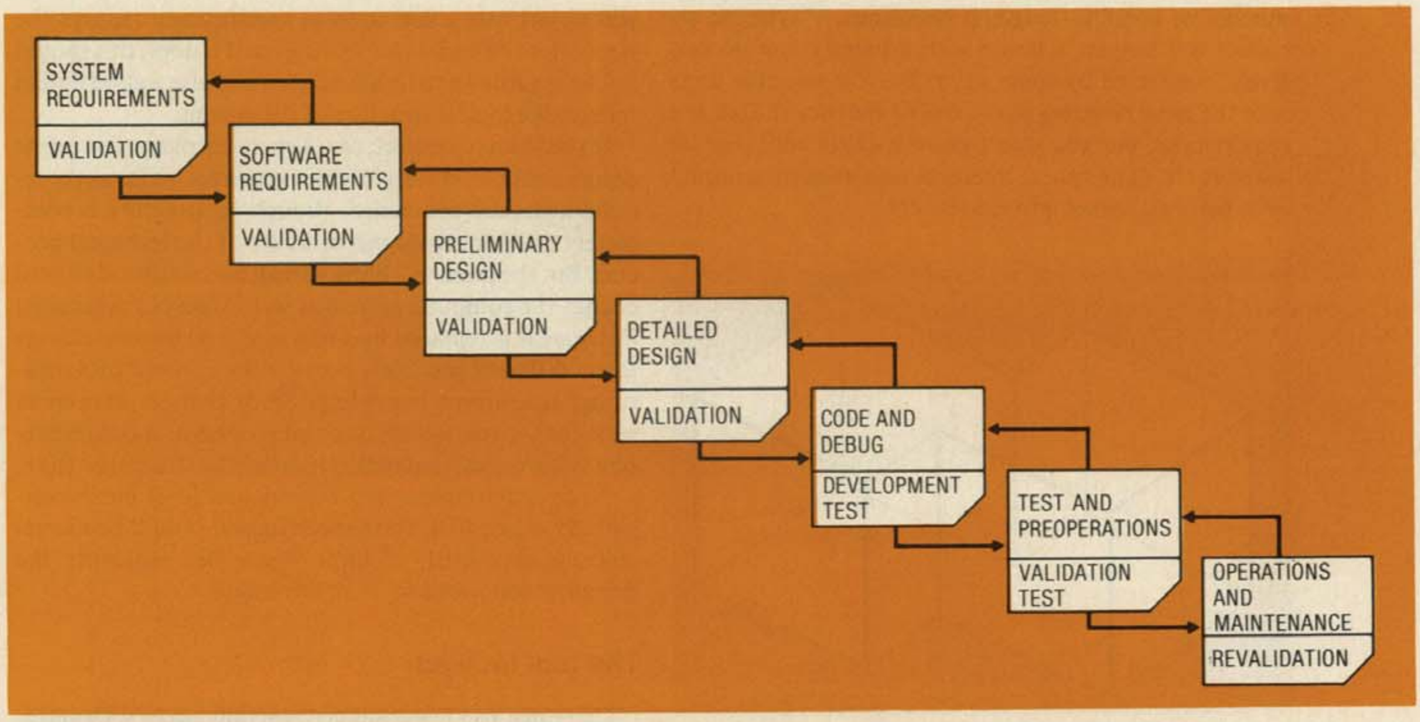

Figure 2. Boehm's life-cycle model.[6]

example, the one by Yeh and Zave[9]—repeat it as an introduction.

To describe a system as a black box, without any assumption about its inner structure, is a very nice idea, so one should keep it in mind as an ideal model. But sometimes I feel that the rule became a dogma, and I wouldn't support that. The effect is similar to the anti-GOTO movement: If 95 percent of all GOTOs should be avoided, there is still a reason to keep the GOTO for some applications which are very complicated without it.

As a matter of fact, there are several specifications of stacks (and sometimes even queues) which prove the feasibility of an "abstract specification." But I have never seen any life-size example consisting solely of genuine requirements. This is due to several reasons:

(1) The analyst has to write down information, but he does not know a language for specification. So he will either use natural language or a (maybe disguised) programming language. This argument can be eliminated by providing and teaching specification languages.

(2) The system exhibits an obvious decomposition, and the parts are easy to describe, while the whole is not. So, the analyst will preferably define the parts and their structure. If he is very clever, he will note that this decomposition is not necessarily the one to be implemented but only a "theoretical" implementation (see Parnas, [10] p. 862).

Figure 3 shows a simple example of such a system. The object can be easily described as a combination of two cubes. But that does not imply it is implemented that way.

(3) The system is very complex, and there is a mutual dependency between requirements and design. I do not feel that enough attention has been paid to this point, so I would like to discuss it more thoroughly.

Imagine you want to build a house for yourself and your family. You will pass your requirements on to the architect, but they are certainly not complete, not only because you have omitted something, but also because the number of possible designs is enormous. Maybe the architect will suggest a house with a living room on two levels, connected by some steps. Do you want the steps with the same covering as the rest of the room? That is a requirement, but you won't know it exists until you are asked for it. (The system interacts with its environment,[1] even before it comes into existence!)

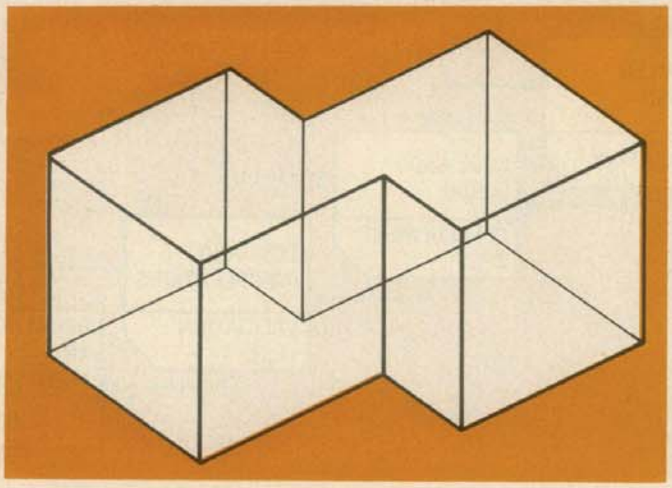

Figure 3. An object with an obvious decomposition.

The phenomenon described above becomes more important the less is known about the system when it is specified. Software to be developed in a scientific environment, where hardly anything is known in the beginning, is an extreme example. The contrary situation occurs when an existing system is specified to allow it to be modified or replaced in a controllable way (e.g., see Heninger[11]). The requirements are abstracted from the behavior of the system as it is, with some desired improvements. In this situation, there is of course no interaction between specification and design, but one should notice that this independence is not the one usually aimed at (*what* before *how*), since the design is legalized only when it is finished.

A popular example of the latter case is a telephone switching system. From a long history of development, standards have emerged which now seem to be independent of a particular technology. But they were not when they were introduced: The freedom of implementation resulted from the progress in technology, which now enables the engineer to replace one old module by many different new ones.

In the case of an existing system, you can write the specifications down by scanning the inputs and outputs in any order you like, because they are already static, and they won't be influenced by the sequence you choose. But if you start from scratch, you need a logical ordering which reduces complexity. This means that you need a hierarchical model.[12] Requirements have an inherent hierarchical structure which shouldn't be waived aside; it should be detected and codified. The requirement to print a message "invalid value, enter again," for instance, is logically subordinate to the requirement to get a number from the operator, which in turn is part of a dialogue, which is necessary for some control task, etc. And that hierarchy should be present in the specification, because it will ease the difficulties of understanding and modification.

Purists, fighting for aseptic requirement specifications, will object that a hierarchical specification reduces the freedom of design. I think that even if it does, this should not be regarded as a disadvantage, because a good design reflects the logical structure of the system.

If you want to control, say, a chemical plant, you might design a computer system which is correct, as far as the requirements are concerned, though its structure is completely different from the structure of the technical process. But the plant will be modified, and somebody has to change the computer system as well. If an old measuring instrument is replaced by a new one, you have to change many different modules, because the concept of a measuring instrument is missing. Since changes happen in terms of the real world, a computer system should represent those terms; that means it should be structured alike.

Since requirements are not on one level but hierarchically dependent, their specification should be hierarchically structured. If those hierarchies influence the design, it will prove to be an advantage.

The tool instead

Certainly, too few pains are generally taken with specifications and in keeping up-to-date documentation.

Thus, we have to spend far more than we saved, during implementation and, particularly, in maintance.

This message has reached the offices of software managers and evoked some activity. They find themselves in a situation in which they have to prove that they know the problem and a solution for it. That means they should get acquainted with *methods* and *techniques* that are in use, carefully choose the one which is applicable in their particular environment, and establish it. Since that is hard work, many managers prefer to buy a *tool* instead. (For scientific environments, use "build" rather than "buy.") This resembles the tendency of many parents to buy expensive toys for their children when they feel they should spend more time playing with them.

This trend is, of course, supported by the suppliers, who do not stop telling us how beautiful their tool is, preferably at scientific congresses. Selling methods is not that easy, especially when you don't have one to sell.

But a tool is only useful under two conditions: First, it has to be the tool that you actually need (if you want to cut glass, you cannot use a saw). Secondly, you need to know *how* to use it. Thus, you cannot choose a tool without analyzing your particular problems, and you cannot establish it without providing the method.

To summarize the above: A tool can be very useful if you buy (or build) it as a result of your analysis and as a complement of the techniques to be applied; but its value is questionable if it is purchased as a substitute. Adding the method to the tool is a rope trick which will fail.

A specification language

After some investigations[13] and practical experiences with available specification systems, a new one was developed, which is named Espreso.[14] Espreso is an acronym standing for "System zur Erstellung der Spezifikation von Prozessrechner-Software" (system for the development of process control software). Based on some fundamental assumptions about the way software is (or should be) developed, concepts were selected which can be expressed in a language (Sprache) called Espreso-S. A tool (Werkzeug) was designed to check, store, and evaluate specifications; its name is Espreso-W.

This section provides an overview of Espreso to show how the ideas given above can be applied to a real specification system. The most important principles, which were used as a general guideline, are listed below. For a more elaborated and complete list of such criteria, see Balzer and Goldman.[15]

The first principle of Espreso is to restrict the specification to aspects which are important for "programming in the large."[16] All arithmetical and logical expressions and assignments are abandoned to reduce the number of concepts and to prevent the users from dealing with details too early. The sequence in which different parts of the system are active (programs are executed) is not relevant on this level, except where a certain sequence is explicitly required (e.g., every line of operator input must be separately prompted). In most situations, the sequence can remain undefined, because it is either implicitly enforced by data flow or it really does not matter. If the language does not discriminate against parallelism (as most programming languages do by offering a very simple grammar for sequential execution but a very complicated one for parallelism), the specification stays on a high level of abstraction, and the implementation can make full use of the parallelism which is conceptually possible (e.g., by a dedicated configuration of microprocessors).

Secondly, Espreso is intended to guide its user. So it has to move towards him. His first ideas are certainly neither formal nor complete, so Espreso-S allows informal descriptions (texts) to be entered, and Espreso-W accepts fragments of specifications as long as they are syntactically correct. Though we are not able to analyze a text for its meaning, there is at least a way to recognize the occurence of names referring to objects in the specification. So you can fix logical connections on a very informal level.

The concepts are simple, easy to understand, and most of them are well known. They are tailored to the needs rather than to the implementation. The support for parallelism mentioned above is one example, another one is the set of mechanisms for communication. Processes may read and write variables, and this implies what you would expect: During writing, no other access is allowed. Processes may also communicate using messages. Sending implies waiting until there is space for the message, and receiving implies waiting if no message is in the buffer. A resource can be used temporarily, which implies that no other process may access it until it is released (mutual exclusion). Those three mechanisms for communication form a problem-oriented model, which nevertheless lends itself to a (maybe inefficient) design, because all elements are defined in terms of programs (see below).

On the other hand, not all concepts currently in use elsewhere have been incorporated. The concept of events, for example, was intentionally left out because events seem to cause difficulties. Events are simple, but too simple to be precise on the level of Espreso-S. If a requirement says "event *E* must evoke action *A*," the meaning seems to be obvious. But what does it require if *A* is just happening or is blocked? And how many actions are necessary after a burst of, say, 10 event *E*'s? Certainly, this can all be specified, with or without events, but I feel events hide the problem. So they are replaced by messages, which I feel are safer. Guidance also means keeping the user away from mined areas.

Finally, the guide should not leave his charge in the middle of nowhere, but show him the way to get home safely. In Espreso, for every construct, a corresponding piece of code is given in a Pascal-like language. The user can easily write down a skeleton of his program, translating the refined specification by hand or automatically if such an additional tool is implemented.

Third, the language has to be well defined. The only way I know to achieve this is to define it formally. Many people argue that a formal description is of no value because the user does not read it, and even if he did, he wouldn't understand it. But a formal description is not made primarily for the user. Even if he does not read it (and maybe the user is not as ignorant as many pretend), it will be his guarantee against unforeseen inconsistencies. When I use a bridge, I feel better if I know that the calculations were done properly, even though I do not understand them. Therefore, Espreso-S was precisely defined (see below).

Besides the principles listed above, Espreso has to meet the requirements which hold for most software systems: Manpower, time, and money are limited, and the programs should be portable, correctable, and modifiable. It should be possible to run Espreso-W on a minicomputer since that is the type of machine used in most process control systems.

An example

Instead of a language description which would inevitably be either too lengthy or too simplified, a small example is given below. Though it does not exhibit all elements and properties of the language, it will show its style and the way it can be used. For a more complete example, see Reference 14.

The problem used here was first introduced[17] with a language called PCSL, which preceded Espreso. It can be stated informally as follows: Every three seconds, a set of some 100 values has to be fetched from a technical process. After conversion to physical units, the data are filtered and recorded. The state of the process is displayed, and the information on the screen will be updated whenever the operator presses a button. In Figure 4, the problem is partially restated in a graphical representation.

The symbols used in Figure 4 correspond to the following sorts (i.e., types of objects) and their relations:

- A rectangle is a procedure (executable unit).
- The other parallelograms are variables (entities which can be written and read).
- The circle represents a concept called a "buffer."
- The ovals are triggers.

Buffers contain messages, i.e., data of a certain type, which are produced and consumed. This differs from reading and writing in that not only the content but also the quantity is affected. A variable can be read hundreds of times without any loss, but a buffer contains items each of which has been produced, and will be consumed, only once. Thus, consuming from an empty buffer implies waiting. Produce operations may also be retarded, because the space in a buffer is limited to a certain capacity. Alternatively, a buffer may be specified to discard all messages when occupied. The triggers are distinguished from buffers because they do not contain messages, just tokens, so they do not have a type like buffers.

In Figure 4, reading and writing is indicated by arrows; double lines are used for producing and consuming. The zigzag arrow means a special kind of consumption (start an activity when a token is available, and consume that token).

In Espreso-S, the system can be specified as follows:

```
procedure Data-collection:          (*beginning of a section*)
text function
@ checks data from the technical process,
  puts reduced data into the archives. Updates
  the display when the operator pushes
  a button on the display. @;
while       System-is-up;
started-by  Timer-1;
reads       Process-data, Scaling-factors, Date-and-time;
produces    Archives;
writes      Screen
end Data-collection                       (*end of section*).
```

In this first sketch, you can see both comments (which are ignored) and a text which is stored. The "function" is an arbitrary key; every key (including the empty string) may be used once for every object. The example is well formatted just to improve readability. The sequence of statements inside the section is meaningless.

The data-collection system has to be refined in order to state the requirements more precisely. We can distinguish three activities running in parallel:

```
procedure Data-collection:
parallel
  block Input-processing:
  while       System-is-up;
  started-by Timer-1;
```

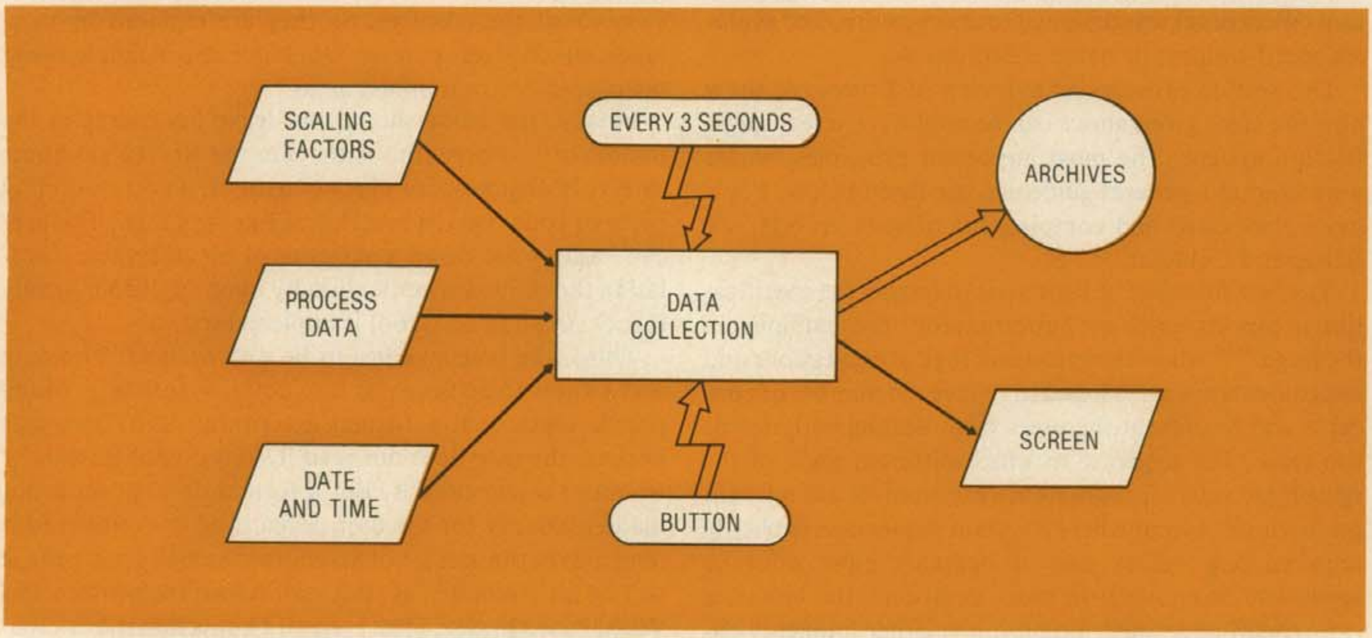

Figure 4. Coarse specification of a simple system.

reads Process-data, Scaling-factors, Date-and-time;
produces Normalized-data;
writes Actual-state
end Input-processing
parallel
block Data-reduction-and-storage:
while System-is-up
(*execution is controlled by buffers only*)
consumes Normalized-data;
produces Archives
end Data-reduction-and-storage
parallel
block Update-screen:
while System-is-up
started-by Button;
reads Actual-state;
writes Screen
end Update-screen
end Data-collection.

Variables and buffers have not yet been defined. Two examples are given below:

buffer Normalized-data:
of-type nd-type
end Normalized-data;

variable Actual-state:
consists-of
variable Sampling-time
and
variable Process-state:
@ this state is written every time
!Input-processing is executed @;
(*the text above contains a reference, indicated by '!'*)
consists-of . . . . .
end Process-state
end Actual-state.

In a real specification, this would probably be only a subsystem among many others. In that case, it is advantageous to hide all the objects within a module to prevent undesired references to data and procedures from the outside.

module Data-collection-system:
comprises procedure Data-collection
and buffer Normalized-data
and variable Process-state
and . . . .
end Data-collection-system.

Procedure data-collection should be accessible from the outside, so we extend the specification by

procedure Data-collection:
available-in . . . (*list of modules from which this procedure may be called*)
end Data-collection.

The example (illustrated in Figure 5) shows that

- objects and their relations are described in sections;
- every object may be described in several (nonconflicting) sections, allowing a stepwise specification; and
- objects linked in a tree structure can be described by recursive constructions of the language.

The information in the specification concentrates on

- data-flow,
- (implicit) coordination of possibly concurrent processes, and
- access-limitations for data and procedures.

Nothing is said about computation of arithmetical or logical values.

Espreso-S requires some redundancy for clarity, e.g., the name of the section has to be repeated at the end. In some situations, however, enforced redundancy does not really contribute to the quality. There, redundant words such as "block" after "parallel" may be omitted.

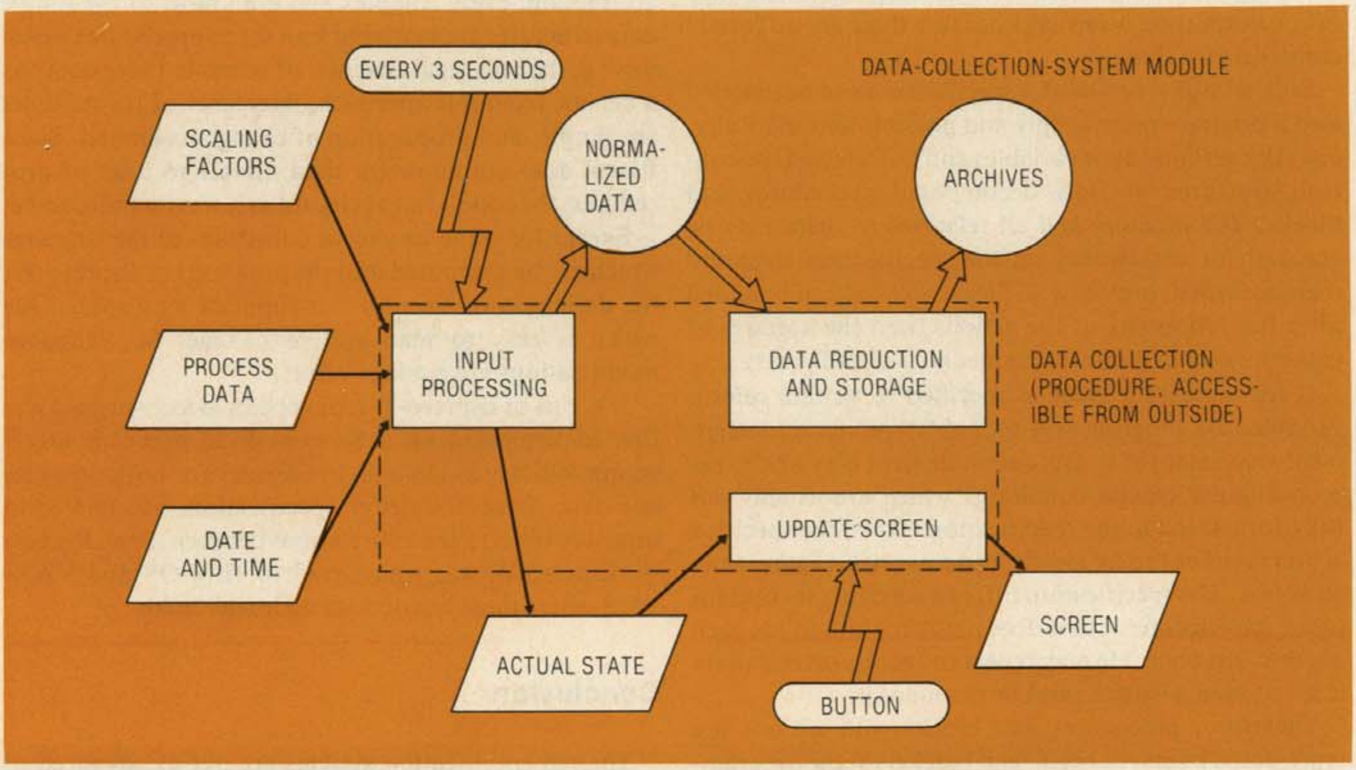

Figure 5. Refined specification.

Definition of the specification language

Algol 60 was described in Backus-Naur form, [18] and Pascal and many other programming languages followed the same line. But BNF is limited to the context-free part of the syntax, leaving the context-sensitive part undefined. A simple example is the declaration of names in Algol 60, which cannot be enforced by the syntax, because that is beyond the power of BNF. Thus, the declaration-rules must be in the informal part of the grammar. This part is euphemistically called "semantics." There are, however, subtle problems hidden in such rules, which lead to difficulties and ambiguities, not only for the reader but also for the implementer. Therefore, Espreso-S was defined by an extended attribute grammar[19] which describes the full, i.e., context-sensitive, syntax. "Syntax" is used here for any rule that will be checked by the processor when a specification is entered or extended. That includes, for example, the consistent usage of identifiers and structural correctness.

When a specification is entered into Espreso-W, the specification stored internally is modified. Since any redundancy, including repetition, is allowed in Espreso specifications, this effect is not trivial; thus, it has to be defined as well. The extended attribute grammar accomplishes that almost as a by-product. When the tool for conversion was implemented, no confusion ever arose about its effect.

The effect, as discussed above, is somehow related to the semantics of specifications (e.g., two semantical equivalent specifications should have the same effect), but the *meaning* intended by the specifier is not defined by the extended attribute grammar.

Defining the meaning of a semiformal specification is difficult because the informal parts cannot be taken into account, though they contain some information. (Otherwise they would be redundant.) In Espreso-S, all arithmetical and logical expressions and assignments may only be expressed informally because there are no formal constructs for that.

Still, a large fraction of a specification can be mapped into a program quite simply and possibly automatically, e.g., the definitions of variables and buffers and the control structures of fully decomposed procedures and blocks. The modules and all relations to them may be checked for consistency on the specification level and then discarded, provided no illegal accesses can be added after that. Mapping of the actions (i.e., the accesses of procedures and blocks to the media) is not that easy.

If for instance a block is specified to write a certain variable, the program will probably contain an assignment statement for it. But that statement may not be executed under certain conditions which are usually not fully formalized in the specification. Thus, the specified action turns out to be merely an access right, rather than an access. The specification offers a set of access options to the implementer, but it does not force him to use each of them just once. He may repeat the access several times (e.g., to read a value again) or renounce it.

Therefore, procedures and blocks with actions are represented by empty begin/end brackets at the programming level. The programmer may choose any code within those brackets, including internal variables and substructures, but the links to the environment are strictly limited to those which had been specified.

Many people tend to look upon a specification as a vague outline of a program. The mapping of Espreso-S into a Pascal dialect[14] proves that it can also be regarded as a skeleton, incomplete but precise.

The tool

Espreso-W was designed and partially implemented to show that the concepts of Espreso lead to a system that can be easily translated into reality. It does not require a huge machine but only a minicomputer with a reliable Pascal compiler.

The most important features of Espreso-W are

- creation of a new file (i.e., a set of lists for a specification) and deletion of such a file,
- conversion of a specification into an internal representation,
- generation of documentation and reports from the stored specification, and
- deconversion of a stored specification, i.e., reconstruction of an Espreso specification.

Conversion and deconversion are possible not only for a complete specification but also for parts of it. That allows stepwise development, refinement, and correction under the guidance of a tool which checks automatically for syntactical correctness.

To minimize the implementation effort, a fairly simple syntax was defined. Pascal was used for implementation, and the error handling is rather plain. Instead of a real data base, a set of lists declared in the Pascal code contains all information. Those lists are restored before and saved after every run.

The design of Espreso-W relies on the concept of data abstraction; every complex and even most of the simple data structures are managed by a set of procedures which serve as the only legal channels of access to those data. As a benefit from this approach, interfaces of the modules are simple, and propagation of changes is limited. Since Pascal does not allow for data private to a set of procedures, the code is not as elegant as it was intended to be.

Except for some very basic constructs of the language which are incorporated into the programs of Espreso-W, the definition of Espreso-S is supplied by a special file which is easy to read and to change. So, language modifications are a minor effort.

A kernel of Espreso-W (conversion, deconversion) was first implemented on a Siemens R-20 minicomputer[20] within 40K words (16 bits) of memory for both program and data. Then, the size of specifications was limited to some 100 objects and 200 relations between them. Recently, Espreso-W was transferred to IBM-OS and VAX-VMS where these restrictions no longer hold.

Conclusion

Though specification systems are not as advanced as some of the suppliers claim, users who have recognized the need for better specifications can expect to find some kind of support. They should first try to state their needs and then make a choice from the various methods, representation schemes, and tools. One benefit of almost *any* formalized approach will be that users will become more aware of this problem area and increase their efforts in this field. Communication and documentation will be improved.

Users' experiences should be published since, for various reasons, developers of a specification system often will not apply it to real projects and thus appreciate any feedback.

The last illustration, Figure 6, is similar to the first one, but slightly more formal. The degree of program formalization, which can only be estimated, is plotted against the time, which is related to the phases of program development. This kind of diagram was first introduced in Ludewig and Streng.[13] The activity of program development is described by a link between the first idea (0 percent formalized) and full implementation (100 percent formalized). The traditional approach is represented by a line that remains very low for most of the time (until programming languages can be used). Recent systems like Espreso can be compared to stepping stones across a river, which allow the user to choose a way off the boggy bank, i.e., to formalize his information early. Current work is aiming at systems which provide a chain of such steps,[21] but these systems are not yet available. ■

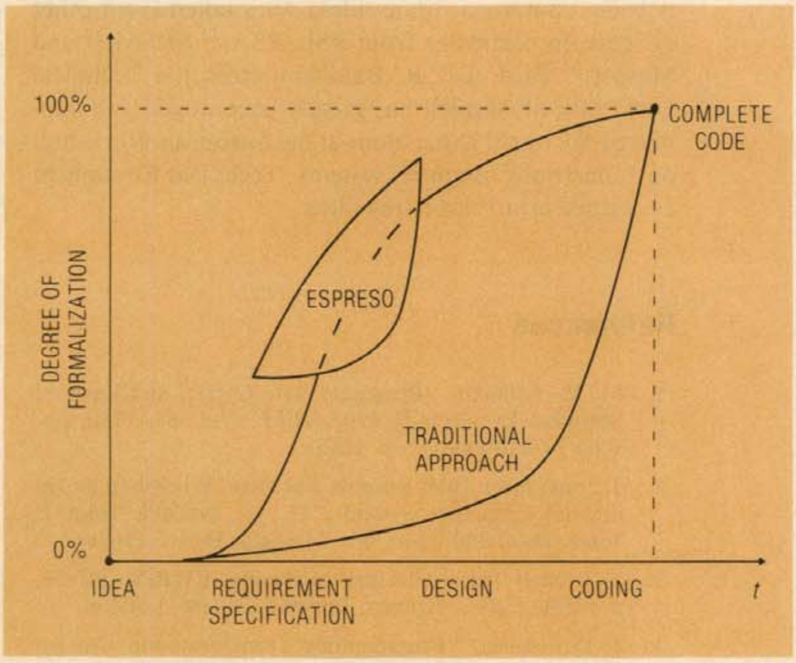

Figure 6. Programming as the process of formalization.

Acknowledgments

Most of the reported results were achieved during my employment at the Nuclear Research Center, Karlsruhe, Federal Republic of Germany, where the implementation

is being continued. Many ideas were taken from other systems, in particular from PSL/PSA,[22] SREM,[23] and Mascot.[24] Prof. Dr. R. Baumann from the Technical University of Munich has greatly encouraged and supported my work. Discussions at the European Workshop on Industrial Computer Systems, Technical Committee 11, helped me to clarify my ideas.

References

1. M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," *Proc. IEEE,* Vol. 68, 1980, pp. 1060-1076 (quotation, p. 1066).
2. J. Jones (ed.), Bibliography, European Workshop on Industrial Computer Systems, TC 11, available from J. Jones, Hatfield Polytechnic, Hatfield, Herts., England.
3. J. Kramer (ed.), Glossary of Terms, EWICS, TC 11, available from J. Kramer, Imperial College, London.
4. J. Darlington, "Programming Transformation: An Introduction and Survey," *Computing Bulletin,* Ser. 2, No. 22, Dec. 1979, pp. 22-24.
5. R. G. Babb and L. L. Tripp, "An Approach to Defining Areas within the Field of Software Engineering," ACM Sigsoft, *Software Engineering Notes,* Vol. 4, No. 4, Oct. 1979, pp. 9-17.
6. B. W. Boehm, "Software Engineering," *IEEE Trans. Computers,* Vol. C-25, Dec. 1976, pp. 1226-1241.
7. D. Parnas, "On a 'Buzzword': Hierarchical Structure," J. L. Rosenfeld (ed.), *Information Processing 74,* North Holland Publishing Company, Amsterdam, 1974, pp. 336-339.
8. D. Parnas, "A Technique for Software Module Specification with Examples," *Comm. ACM,* Vol. 15, No. 5, May 1972, pp. 330-336.
9. R. T. Yeh, and P. Zave, "Specifying Software Requirements," *Proc. IEEE,* Vol. 68, 1980, pp. 1077-1085.
10. D. L. Parnas, "The Use of Precise Specifications in the Development of Software," B. Gilchrist (ed.), *Information Processing 77,* North Holland Publishing Company, Amsterdam, 1977, pp. 861-867.
11. K. L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," *IEEE Trans. Software Eng.,* Vol. SE-6, No. 1, Jan. 1980, pp. 2-13.
12. H. A. Simon, "The Architecture of Complexity," *Proc. American Philosophical Society,* Vol. 106, No. 6, Dec. 1962, pp. 467-482.
13. J. Ludewig and W. Streng, "Methods and Tools for Software Specification and Design—A Survey," EWICS, TC 7, Paper No. 149, Zurich, April 1978.
14. J. Ludewig, "Zur Erstellung der Spezifikation von Prozessrechner-Software," doctoral dissertation, Technical University Munich, 1981. Reprinted as KfK Report No. 3060, Kernforschungszentrum Karlsruhe, FRG.
15. R. Balzer and N. Goldman, "Principles of Good Software Specification and Their Implications for Specification Languages," *Proc. Specification of Reliable Software,* 1979, IEEE Cat. No. 79 CH 1402-9C, pp. 58-67.
16. F. DeRemer and H. H. Kron, "Programming-in-the-Large versus Programming-in-the-Small," *IEEE Trans. Software Engineering,* Vol. SE-2, No. 2, June 1976, pp. 80-86.
17. J. Ludewig, "PCSL—A Process Control Software Specification Language," KfK Report No. 2874, Kernforschungszentrum Karlsruhe, FRG, 1980.
18. P. Naur (ed.), "Revised Report on the Algorithmic Language ALGOL 60," *Comm. ACM,* Vol. 6, No. 1, Jan. 1963, and *Numerische Mathematik,* Vol. 4, 1963, pp. 420-453.
19. D. A. Watt and O. L. Madsen, "Extended Attribute Grammar for Pascal," *Sigplan Notices,* Vol. 14, No. 2, Feb. 1977, pp. 60-74.
20. K. Eckert and J. Ludewig, "ESPRESO-W, ein Werkzeug für die Spezifikation von Prozessrechner-Software," G. Goos (ed.), *Werkzeuge der Programmiertechnik,* Springer-Verlag, Berlin, Heidelberg, New York, 1981, pp. 101-112.
21. W. Howden, "Contemporary Software Development Environments," *ACM Software Engineering Notes,* Vol. 6, No. 4, Aug. 1981, pp. 6-15.
22. D. Teichroew and E. A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," *IEEE Trans. Software Eng.,* Vol. SE-3, No. 1, Jan. 1977, pp. 41-48.
23. M. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Trans. Software Eng.,* Vol. SE-3, No. 1, Jan. 1977, pp. 60-69.
24. K. Jackson and H. F. Harte, "The Achievement of Well Structured Software in Real-Time Applications," *Proc. IFAC/IFIP Workshop on Real-Time Programming,* Rocquencourt, June 1976, pp. 229-238.

Jochen Ludewig is a member of the Computer Science Group of Brown, Boveri & Co., Ltd., Research Center at Baden near Zurich, Switzerland. He is interested in methods and tools for the early stages of software development. From 1975 until 1980, he worked at the Institute for Industrial Data Processing, which is part of the Nuclear Research Center, Karlsruhe, FRG. There, his main activity was in software documentation and specification, including the installation and extension of PSL/PSA.

Ludewig is a member of the German Society for Informatics and of the Technical Committee on Application Oriented Specification of the European Workshop on Industrial Computer Systems, the European Purdue Workshop. He received his master's degree in electrical engineering from the Technical University of Hannover and a postgraduate certificate in computer science from the Technical University of Munich, where he also obtained a PhD.