SPECIFICATION OF A SPECIFICATION LANGUAGE

J. Ludewig

Brown Boveri Research Center, CH - 5405 Baden, Switzerland

Abstract. ESPRESO is a recently developed specification system for process control software. It consists of a specification language and a software system which serves as a tool to check, manage, and evaluate specifications. The language was precisely defined by an Extended Attribute Grammar and by a mapping into a programming language. The definition turned out to be most valuable not only for the implementation of the tool but also for the improvement of the language itself. As a by-product of this work, a better understanding of the nature of a specification language was achieved.

Keywords. Specification Language, System Specification, Language Definition, Software Tools.

INTRODUCTION

Based on experiences with PSL/PSA (Teichroew, Hershey, 1977), concepts for a specification language dedicated to process control software were developed. This language was called PCSL (Ludewig, 1980). It can be processed by the so called Generalized Analyzer (GA), an extended, table driven version of PSA. Thus, we did not have to develop a new tool for PCSL, but, on the other hand, we had to meet the requirements given by the GA.

Some of the disadvantages of PCSL/GA important for us were the following:

- The GA is very large (about 55 000 lines of code, mainly FORTRAN IV). Though it is well constructed, such a program is difficult to install and maintain, and a mainframe-computer is necessary.

- The GA is intended to process variations of PSL, rather than an arbitrary new specification language. E.g., the language definer can neither define a recursive syntax nor a handling of texts different from that in PSL. Thus, any language for the GA will unevitably remain syntactically similar to PSL.

- Though there is still a reasonable freedom for the language definer, he is not able to enforce the correct use of his language, because the GA cannot check any conditions but the simplest. Therefore, additional tools are necessary to perform those checks.

- The tables which describe the actual language are generated from a formal definition. This, however, does neither cover those properties that cannot be influenced by the language definer, nor are they defined elsewhere. Thus, there is in fact no complete formal definition of the language.

(Some of these properties of the GA may have been improved since 1979 when we observed them.)

As a consequence, a new system, which is no longer dependent from any existing software, was developed, based on the improved concepts from PCSL.

THE ESPRESO-SYSTEM

ESPRESO is a German acronym standing for "development of the specification of process control software". It was designed to provide an aid in the process of formalization. Its components are a formal language (ESPRESO-S) and a tool (ESPRESO-W) to check, store, accumulate, modify, and evaluate specifications. Both are built upon a basic set of concepts, which can be summarized as follows:

- All information, whether formal or not, should be documented as early as possible.

- The user should be supported in formalizing the specification.

- The user should be hindered from stating details too early.

- The clerical work to be done by the user should be minimized.

- There should be one central specification which can be easily accessed and updated by everybody.

- Tools are necessary for detecting errors as early as possible.

- Languages for specification should resemble other good languages, like PASCAL, e.g. To satisfy the needs of non-professional readers, various representations of the language may be defined (including graphics), but there has to be a sound basis. Pragmatic extension may ruin the concepts.

- The language should provide constructs which are simple, well known, easy to use, and translatable into well structured programs.

Since these requirements are partially contradictory, a compromise has to be looked for.

As a result, ESPRESO-S is a block-oriented, non-procedural specification language, stressing the static hierarchy of systems and subsystems ("modules"). Within modules, active and passive components are described, which represent executable programs and data. Much emphasis is on the data flow, which implies coordination of competing processes. Thus, no explicit synchronization is necessary.

To give some idea of the language, an example is shown below.

```
module data-collection:
text purpose @ system for receiving,
       filtering and storing data from
       a technical process @;

comprises
   buffer raw-values:
   produce restricted-to reader;
   consume restricted-to filter
   end raw-values

and
   module reader:
   comprises
      procedure get-values:
      started-by read-trigger
         where @ every 5 sec @;
      produces raw-values;
      reads periph-input
         where @ all sensors have to be
                     scanned within 100 msec @
      end
   end reader
end data-collection;


buffer raw-values:
capacity 10;
of-type value-record
end.
```

This small example exhibits but a few important features of ESPRESO-S: The user may exploit the recursive syntax to describe his system in a most natural way; he is allowed to reference objects which are not yet defined; he can repeat or extend definitions; he may use informal texts which can be managed by the tool ESPRESO-W, though they cannot be evaluated like the formalized information.

THE NEED FOR A WELL DEFINED SPECIFICATION LANGUAGE

Nobody can expect the user to deliver a complete and formal specification as the very first step of his work. So, if he is required to write down all his information as early as possible, the language must comprise of constructs for informal and imprecise information, which the tool must be able to handle. Some people conclude from this situation that there is no need for a precise definition of the specification language.

Experience proves that the opposite is true. The language for specification must be well defined, even more so because the specification itself tends to be incorrect (with respect to the intended meaning), incomplete, inconsistent, and vague. Natural language or an unclear specification language will blur those deficiencies.

A second reason is that the semantics of a non-operational language can not even be discovered by testing, as is frequently done in the use of ill defined programming languages. If a specification language is not clearly defined, it will be ambiguous forever.

The definition must cover three aspects of the language:

- Some specifications will be accepted by the tool, while others won't. The rules which distinguish between those two groups are called syntax. In the past, "syntax" was often used in the sense of "context free syntax". It should be noticed that context-sensitive elements like the consistent usage of names are included here.

- When a specification is processed by the tool, much redundant or meaningless information is discarded. E.g., if an object is specified twice, only one definition is stored, comprising of the union of both definitions. The user should know exactly what his input means to the content of the abstract specification stored by the tool. Then he will also know if two specifications are equivalent or not. This information is called semantics here.

- Finally, and most importantly for the user, every construct of ESPRESO-S has some meaning which, eventually, must be reflected by the ultimate implementation. This is called meaning, simply because the term "semantics" is already occupied. Semantics and meaning are only defined for specifications which are syntactically correct.

In the following paragraphs, each of those three aspects is discussed.

DEFINITION OF THE SYNTAX

After ALGOL 60 was defined by a set of production rules (Naur, 1963), BNF became widely used and accepted even by non-specialists, because it is easy to write, read and understand. The representation was sometimes modified, e.g. for PASCAL (Jensen, Wirth, 1974), or extended (SeegmÜller, 1974), but the principle remained unchanged.

BNF is limited to context free languages; therefore, many rules had to be defined informally (e.g. that every variable must be declared). Many attempts were made to extend the capabilities of formal grammars in order to include such information (Marcotty and co-workers, 1976). One of the new, more powerful type of grammars were the Attribute Grammars by Knuth (1968). A particularly concise notation of them was called "Extended Attribute Grammar" (EAG) (Watt, Madsen, 1977; Watt, 1979). This schema was applied for the definition of ESPRESO-S. Below, a short introduction is given to the principles of EAGs, as they are used for ESPRESO-S.

An EAG can be directly obtained from a BNF-grammar by adding certain information about the context. Since the context is usually arbitrarily complex, meta-rules containing meta-variables are used instead of actual rules; the latter can be generated from the former by substituting actual values for meta-variables, consistently throughout a meta-rule. Let a simple example serve as an explanation:

A section is a fundamental construct of ESPRESO-S for defining objects. For the sake of readability, the name of the object has to be repeated at the end of most sections. The context-free grammar in BNF is:

```
<section> ::= <object-sort> <object-name>
              colon <section-body>.

<section-body> ::= <statement> <section-body>
                 | end-symbol <object-name>.
```

In this simplified example, the production rules for <object-sort>, <object-name>, and <statement> are missing, but obviously the rules cannot enforce that the two occurrences of the object-name are to be consistent. That can be achieved by the use of attributes.

```
<section ↑ NAME> ::=    <object-sort>
                        <object-name ↑ NAME>
                  colon <section-body ↓ NAME>.

<section-body ↓ NAME1> ::=
          <statement> <section-body ↓ NAME1>
        | end-symbol <object-name  ↑ NAME2>
                     <test NAME1 = NAME2>.

<test TRUE> ::= .
```

If NAME, NAME1 and NAME2 are substituted consistently, the naming of sections is necessarily consistent, because the grammar does not provide a production rule for <test FALSE>. In a similar (however, often more complex) manner all context-sensitive properties of ESPRESO-S are defined.

In the example, the attributes are marked by a vertical arrow, pointing either up or down. The direction is given only to improve readability; so called inherited attributes (down) are fully defined by the context, while the synthesized attributes (up) are at least influenced from the productions of the syntactical variable under discussion. NAME1, for instance, is defined by the name in the sectionheader; there it is a synthesized attribute. The section body, on the other hand, uses NAME1 as an inherited attribute.

Here, only one attribute was introduced; real productions will contain several, typically from two up to four or five. In EAGs, the attributes of one particular meta-variable are distinguished simply by their position, rather than by a name, just like parameters of procedures in most programming languages are.

For a typical syntactical variable, there is an inherited attribute whose value is a set, which contains most of the relevant information about the actual context at the very point of analysis. In the definition of a programming language, e.g., that attribute would at any particular point of the program hold all valid (declared) names and their related types. If the subtree of that syntactical variable may contribute to the context of other variables, a second set is defined for a synthesized attribute. The general construction is:

```
< variable-name ↓ INHER-CONTEXT ∨ ...
      ↑ ... ↑ (INHER-CONTEXT PHI NEW-INFO) >.
```

The last parameter is the new context, which consists of the inherited context plus the information derived from the subtree of "variable-name". PHI is an operator, especially defined for this grammar. If two sets are "added" by PHI, the result is undefined if they are inconsistent (e.g. "X is a procedure" PHI "X is a data"). An undefined result means an error-message during conversion. Otherwise, the informations are superimposed, and only the consistent, non redundant subset of the result is kept. Thus, the context attribute can never become redundant or contradictory.

In the grammar of ESPRESO-S, PHI is formally defined by set-operations.

DEFINITION OF THE SEMANTICS

The context-attribute has to contain all information relevant for checking at any point during analysis. Since most of the semantics

is relevant in this sense, it passes through the context attribute. In the grammar of ESPRESO-S, two steps were taken to have the whole semantics accumulated within one attribute: Firstly, no information is ever discarded, i.e. the set is never reduced on input. Secondly, some information is added to the attribute though it is never used for any check.

Thus, the syntax-definition of ESPRESO-S provides an excellent description of the "abstract specification" which has to be built up in the so called ESPRESO-file (a fairly complex data structure) when a specification is entered. The effect of adding information to a specification which is already stored fits very well into this concept because ESPRESO-W reacts exactly as though the concatenated specifications would have been fed into an empty ESPRESO-file.

The semantics of ESPRESO-S is evaluated in the so called conversion, which is the initialisation or extension of the stored specification by processing ESPRESO-S-input. The reverse operation is accordingly called deconversion; parts of the specification, as selected by the user, are retransformed into ESPRESO-S, and deleted in the abstract specification. The deconversion could be defined in a similar way as the conversion; another special operation like PHI is necessary to describe the effect of removing information from the ESPRESO-file.

When a report is generated by ESPRESO-W, a subset of the information kept in the ESPRESO-file is selected and printed in some convenient syntax. Obviously, this process could also be defined formally. Again, the specification of the report would exhibit a clear distinction between content and form, which is very desirable for designing reports.

THE IMPLEMENTATION OF ESPRESO-W

Theoretically, the program for conversion could have been automatically gene- rated from the EAG. Even though we might have had access to such a generating system, we did not consider its use, because ESPRESO-W was required to run on a minicomputer, and even to perform reasonably. Nevertheless, the formal defini- tion was very useful for implementation (Eckert, Ludewig, 1981). No questions ever arose due to ambiguities, and implementation was easily separated as a task for a master-thesis. The structure of the grammar was used as a guide for structuring the programs; while the few most basic constructs are handled by special code, the large number of similar statements is treated by a table driven system. As the syntax is recursive, the programs for conversion and deconversion are recursive as well.

MAPPING OF ESPRESO-S INTO A PROGRAMMING LANGUAGE

Though the grammar defines precisely the mapping from a user's specification to the abstract specification stored within ESPRESO-W, it does not say anything about the meaning of that specification to the user. This problem was attacked separately, not within the grammar. This definition is only semi-formal, and no complete algorithms are given.

Defining the meaning of a non-operational language turned out to be very hard. The basic approach was to map ESPRESO-S onto another machine, the operations of which are well defined. A virtual machine was chosen, which was called E-PASCAL ("E" stands for ESPRESO). E-PASCAL differs from standard-PASCAL by some exten- sions which are very useful or even necessary to implement programs specified in ESPRESO-S. A more powerful language like ADA could have been used without extensions, but such a mapping would not have shown which particular requirements on the implementation language and the run time system are imposed by the specification language.

Most units of the specification like variables, buffers, etc. can be transformed into complete declarations, which the programmer need not access any more. But, obviously, a specification written in a language that does not allow for a complete software-description cannot be mapped onto a program ready for compilation. Therefore, those procedures and blocks which are specified to perform some so called actions (e.g. reading) must be finished by the programmer. For every such unit, a "hole", i.e. an empty frame, is generated, permitting only those procedures and operations to be accessed which are specified. Inside the holes, the programmer may declare and define whatever he wants, but the interface is fixed.

Let us assume that the specification contains the following definition:

```
procedure check-input:
consumes measured-value
   where @ between 0.0 and 30.0 @;
reads upper-limit, lower-limit;
produces checked-value
end check-input.
```

This procedure is mapped onto E-PASCAL as follows:

```
procedure check-input;
interface
   consumes measured-value
      (* assertion: between 0.0 and 30.0 *);
   reads    upper-limit, lower-limit;
   produces checked-value
interfend;
begin
   (* the hole to be filled *)
end (* check-input *).
```

The interface-declaration provides all access-paths to the environment which are available for the code to be added in the hole.

In general, a medium may be accessed by more than one process at the same time. Therefore, a full definition of actions must also cover the mechanisms applied for coordination. In the description of ESPRESO-S, monitor-like sequences of operations are defined for all actions. These monitors are based on the INC and DEC operations as defined by EWICS TC 8 (Lalive d'Epinay, 1979).

THE FEEDBACK FROM A FORMAL DEFINITION

Some people who attach little value to formal definitions regard it to be nothing but a supplement to the language. But if the definition is developed just as the language evolves, it will be much more. If the language is not as simple as it should be, the formal definition won't be either. If the semantics are puffy, it will be very difficult to find an appropriate set of attributes.

On the other hand, if the language is required to have certain formal properties, a formal definition can be used to prove them. ESPRESO-S was kept as simple as possible, in order to ease the implementation of the tool. Two major contributions to simplicity were made by choosing a simple grammar: The context-free syntax is of type LL(1), i.e. the syntax-tree of any correct specification can be constructed without any look-ahead or backsetting. In the context-sensitive syntax, all attributes can be evaluated within one pass from left to right (Bochmann, 1976). Both properties could be easily checked by the formal definition (Eckert, 1980). (The LL(1)-property is not obvious, because there are some left-recursive productions in the grammar, which can be removed by some simple transformations.)

Last but not least, the investigations about the mapping to a programming language provided much deeper insight into what is really expressed in a specification language of this type.

MANAGEMENT OF A LARGE GRAMMAR

All the advantages of a formal grammar may be useless when it is full of errors. Therefore, some care must be taken to make sure that the errors can be controlled in some way. Our experience confirmed the statement by Marcotty and co-workers (1976) that a text-management-system is necessary. The grammar of ESPRESO-S was stored on computer for about one year, and again and again output on an inkjet-printer, which provides a large set of special characters including arrows. It should be noted that the availability of such tools may be crucial for the success.

CONCLUSION: GENERALITY AND PRECISENESS IN PROGRAM SPECIFICATIONS

The grammar of ESPRESO-S and its mapping to a programming language show that a specification language can be formally defined, and that such a definition is desirable. It is feasible in spite of the fact that the ultimate code cannot be generated from the specification. The trick here is to allow for "holes" or "white stains" in the specification, which do not bother the analyst. When the specification is mapped onto code, the limit of the hole is well defined while its contents are not.

For some applications, the holes provided by ESPRESO-S may still be too narrow. For instance, the analyst might like to state that there will be some communication between two modules without saying anything about, let's say, the direction. In such a case, it would be useful to provide additional terms which are more general. Such terms can be defined to be the union of other ones. Then, they are still as precise as the old ones.

Thus, a clear distinction is made between fuzziness and generality: The terms of ESPRESO-S have a general meaning, and additional levels of generality could be added. But the language is still precise, and any implementation can clearly be said to be correct or incorrect, with respect to the specification.

ACKNOWLEDGEMENTS

ESPRESO was developed at Nuclear Research Center, Karlsruhe, and supported by Prof. Dr. R. Baumann at Munich Technical University.

Many ideas were taken from other systems, in particular from PSL/PSA (Teichroew and Hershey, 1977), SREM (Alford, 1977) and MASCOT (Jackson, Harte, 1976).

The current work gains by the excellent conditions in our group at Brown Boveri Research Center, in particular from the discussions with Michael Vitins.

REFERENCES

Alford, M. (1977): A requirements engineering methodology for real time processing requirements. IEEE Trans. Software Eng., SE-3, 60-69.

Bochmann, G.V. (1976): Semantic evaluation from left to right. Commun. ACM, 19, 55-62.

Eckert, K., J. Ludewig (1981): ESPRESO-W, ein Werkzeug für die Spezifikation von Prozessrechner-Software. In G. Goos (Ed.), Werkzeuge der Programmiertechnik. Springer-Verlag, Berlin, Heidelberg, New York, pp. 101-112. (in German)

Jackson, K., H.F. Harte (1976): The achievement of well structured software in real-time applications. Proc. of the IFAC/IFIP Workshop on Real-Time Programming, Rocquencourt, June 1976, pp.229-238.

Jensen, K., N. Wirth (1974): PASCAL: user manual and report. Lecture Notes in Computer Science, Vol.18, Springer-Verlag, Berlin, Heidelberg, New York.

Knuth, D.E. (1968): Semantics of context-free languages. Math. Syst. Theory, 2, 2, 127-145 (see also 5, 1, 95-96, for corrections; same author and title, 1971).

Lalive d'Epinay (Ed.) (1979): TC 8 up to date report. European Workshop on Industrial Computer Systems (EWICS), Technical Committee on Real-Time Operating Systems, paper no. I-1-8.

Ludewig, J. (1980): Process control software specification in PCSL. in V. Haase (Ed.): IFAC/IFIP Workshop on Real-Time Programming, Graz, April 1980, Pergamon Press, pp. 103-108.

Ludewig, J. (1981): Zur Erstellung der Spezifikation von Prozessrechner-Software. Doctoral dissertation, Technical University München. Reprinted as KfK-Report No. 3060, Kernforschungszentrum Karlsruhe, FRG. (in German)

Marcotty, M., H.F. Ledgard, G.V. Bochmann (1976): A sampler of formal definitions. Computing Surveys, 8, 191-276.

Naur, P. (Ed.) (1963): Revised report on the algorithmic language ALGOL 60. Numerische Mathematik, 4, 420-453.

Seegmüller, G. (1974): Einführung in die Systemprogrammierung. Bibliographisches Institut AG, Zürich, Reihe Informatik/11. (in German)

Teichroew, D., E.A. Hershey III (1977): PSL/PSA: A computer aided technique for structured documentation and analysis of information processing systems. IEEE Trans. Software Eng., SE-3, 41-48.

Watt, D.A., O.L. Madsen (1977): Extended attribute grammars. Report no.10, University of Glasgow, Computing Department.

Watt, D.A. (1979): An extended attribute grammar for PASCAL. SIGPLAN-Notices, 14, No.2, 60-74.