ESPRESO—A System for Process Control Software Specification

JOCHEN LUDEWIG

Abstract—This paper outlines a specification system for process control software, named ESPRESO, which was developed at the Nuclear Research Center, Karlsruhe, West Germany. ESPRESO is based on some new ideas, which are combined with elements taken from other systems. ESPRESO consists of a set of concepts, a specification language, a tool for the management, evaluation and validation of specifications, and the method how to use the system. Language, tool, and method are carefully adapted to the concepts. The primary aim was to demonstrate some features of a specification system which are currently not available, rather than to provide a new tool for the software market.

The concepts of ESPRESO were abstracted from the most common tasks in real time programming, focusing on precisely defined operations for process communication. The concept of modules allows for the definition of static units which hide their internal structure from the outside. An attribute grammar was used, which describes not only the context sensitive syntax of the language, but also the effect of the tool for storing specifications. The tool-set was coded in Pascal, and installed on IBM-OS and VAX-VMS.

While ESPRESO is already being used for the specification of a nuclear reactor safety system, more tools are being implemented in order to provide better means for validation and documentation.

Manuscript received December 12, 1981; revised August 17, 1982. An earlier version of this paper was presented at the Seventh Conference on Operating Systems, Visegrad, Hungary, January 1982.

The author is with the Brown Boveri Research Center, CH-5405 Baden, Switzerland.

Index Terms—Attribute grammar, language definition, process communication, process control software, software specification.

I. Experiences with PSL/PSA and the Generalized Analyzer

IN 1976, a group at the Nuclear Research Center, Karlsruhe, West Germany, began to investigate the current state of systems for the specification and design of software. PSL/PSA [1] by ISDOS was soon identified as the best candidate for the institute, and installation was started early in 1977.

When the system was operational, we tried to apply it to some test cases which were taken from the current work of our colleagues and also from textbooks. The results did not support any simple answer as to whether PSL/PSA would improve our software development. We found that the basic ideas (e.g., the concept of a central database and the approach to modeling systems by objects and their relations) are certainly useful, and the implementation was, under the given constraints, successfully achieved, but, on the other hand, we were not completely happy with the PSL/PSA due to some significant difficulties. These were

- the large size of the system, which makes it hard to manage,

0098-5589/83/0700-0427$01.00

- the lack of good training material, without which meant we did not know how to use it,
- the vague explanations of syntax and semantics, which were obviously caused by a very pragmatic style,
- and, for our particular environment (process control software), the absence of constructs for describing the dynamics of the target system.

By 1978, ISDOS had extended their tools, so that languages other than PSL could be defined, and processed by the so-called Generalized Analyzer (GA) which enabled us to develop a language named PCSL (Process Control software Specification Language) [2]. Thanks to an improved basic model, PCSL is better suited to our applications than PSL was. Still, PCSL suffered from some of the disadvantages of the GA. We were not able to define a recursive syntax (for nested structures), and we could not compel the GA to refuse specifications which are inconsistent with our structuring rules. Finally, the GA was even larger than the PSA-system, and offered (at least at that time) very little support for the evaluation and documentation of specifications.

II. Aims and Concepts of ESPRESO

ESPRESO is a German acronym standing for "development of the specification of process control software" ("System zur Erstellung der Spezifikation von Prozessrechner-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.

The goals listed below are valid for any software specification system.

- 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. There should be one central specification which can be easily accessed and updated by everybody.
- The language should be based on concepts which are simple, well known, easy to use, and translatable into well structured programs. Syntactically, languages for specification should resemble other good languages, e.g., Pascal. To satisfy the needs of nonprofessional readers, various representations of the language may be defined (including graphics).
- A set of tools should be provided to support all activities related to the specification, in particular, creation and maintenance, for error detection, communication, and documentation. The clerical work to be done by the user should be minimized. The tools should be easy to implement and modify. Installation on other computers, including mini-computers, should require only minimal changes.

In the case of ESPRESO, the special properties of process control software must be taken into account. These properties, which are discussed in [3], can be summarized as follows.

Process control systems are part of larger systems which also consist of asynchronous technical processes and human operators. The processes, whose optimization is the overall goal, dictate the interface to the control system. This implies that a system for process control software specification must be capable of representing:

- the environment, and its interface with the process control system,
- parallelism and communication,
- timing constraints.

Since the goals are partially contradictory, ESPRESO had to be a compromise.

III. The Specification Language ESPRESO-S

According to the previous goals, and to our experience with PSL and PCSL, the basic concepts for describing the process control system were chosen as follows.

- If there is a choice among different structuring principles, tree structures should take preference as they allow simple description and easy understanding. The language must support nested representation of tree structures.
- The language must allow for mixing formal and informal parts ("texts") in the specification. The management of the texts must be supported by the language.
- There must be a construct for describing units which consist of data and the operations up on it ("abstract data structures").
- Constructs must be provided for the simple description of data flow and the implicit definition of the coordination of concurrent processes. Passing *messages* is a particularly important way of communication, so the model should provide special constructs to support this. Messages can also be used instead of simple *events* which are lost once they have occurred. According to our experience, specifications with such events tend to be ambiguous and incomplete.
- Arithmetic is banned from the model in order to keep it simple, and also to support high level specification.
- ESPRESO should not enforce a particular sequence of specification.
- The grammar of the language must be simple, not only in order to limit the size of the tool but also to make it easy to learn.
- The language must be well defined, in every respect.

Different from the approach reported elsewhere [4], the object-relationship-model [5] was not questioned when ESPRESO-S was designed.

As a result, ESPRESO-S offers to the user 14 kinds of objects (entities) and 45 relations. The entities are as follows.

- *Text-objects* for any kind of informal texts. Several texts may be distinguished by *selectors*. Texts can also be attached to any other object.
- *Modules* for describing abstract data structures, whose content is accessed only in a strictly controlled way.
- *Procedures* and *blocks*, which are the active components of the systems. Procedures may be called at several points, while blocks are used only at one point, forming a tree structure under the relations for refinement (see below).
- *Parameters*, which are split up into input-, transient-, and output-parameters. They are used to communicate with procedures.
- *Variables* (in the usual meaning).

- *Buffers* for messages.
- *Triggers* for trivial messages (tokens).
- *Resources* for concrete or abstract, nonconsumable objects.

Triggers are used to represent events, including those defined by time. Variables, buffers, triggers, and resources are called *media*. They can only be accessed by "actions" (see below).

- *Types* for describing variables, buffers and more complex types.
- *Durations* for defining real time requirements and performance.
- *Constants* for predefined values.

The most important *relations* in ESPRESO-S can be summarized as follows.

- A module *comprises* other objects, including modules.
- A procedure or a block may *call* a procedure, or be refined by several blocks. The execution of subordinate blocks is defined to be either *sequential*, *parallel*, or *alternative*. If specified, execution of a procedure or a block is *repeated* as long as a condition holds.
- A parameter of a procedure is *assigned* to a constant, variable or buffer by a calling procedure.
- A procedure or block accesses a medium (e.g., *reads* a variable, *produces* an item for a buffer or *occupies* a resource). Every such access is called an *action*. Actions imply the necessary coordination of parallel processes (reader/writer, producer/consumer, mutual exclusion). Media and actions form a central concept of ESPRESO. They provide the means to describe the communication of asynchronous processes without diving into the details of synchronization.
- Procedures may be accessible (*available*) from certain modules only. Similar restrictions exist for all media.
- Dynamic relations (*priority*, *delay*, *cycle*) are used for the description of procedures, blocks, and triggers.
- Media may be described by several relations (e.g., *has-type*, *capacity*).
- For classifying objects, the names of text-objects may be used as *keywords*. Texts, which are in general informal, may *reference* any object by preceding its name by an exclamation mark.

The basic syntactical structure of ESPRESO-S is the *section*. A section is used to explicitely introduce an object, to put some informal text on it, or to state its links to other objects. Examples are

```
procedure get-value          (* section-header only  *)
```

and

```
buffer raw-value             (* section header        *)
:                            (* beginning of s.-body  *)
$ unfiltered signals $;      (* text                  *)
capacity 12                  (* statement             *)
end raw-value                (* section-tail          *).
```

Since many statements may contain sections, the sections are nested. A specification is defined to be a sequence of sections.

To give some idea of the language, an incomplete specification for a mixer-system is shown below. This problem was used as a standard example by the Technical Committee on Application Oriented Specification of EWICS, the European Purdue Workshop.

The indentation shown below is the one generated on output. ESPRESO-S is a free format language, but the user can control the format within texts in order to preserve structures like tables, etc. Note the difference between comments and texts. Texts are enclosed by dollar-symbols, and stored when the specification is processed, while comments are simply discarded.

This example was checked by the tool ESPRESO-W.

```
informal problem:
$ The technical process to be controlled produces some
  material, which is made from three components, two
  liquids and some solid bricks. While the liquids are por-
  tioned one after the other on a scale, two bricks are
  supplied from a conveyor belt. After a certain time of
  mixing, the mixer is tipped, and the cycle starts again. $
end problem;

procedure control-all:
while process-running;       (* means cyclic execution of
                                control-all until process-running
                                is no longer true *)

sequential
   block control-supply:
   (* In the sequel, the redundant word block is omitted,
                                where possible *)
   parallel                  (* control-supply is refined by
                                parallel blocks *)
      liquid-supply:
      sequential             (* liquid-supply is refined by
                                sequential blocks *)
         weigh-A             (* weigh-A is not defined at this
                                point *)
      then
         weigh-B
      then
         empty-scale
      end liquid-supply
   parallel
      brick-supply           (* in parallel with liquid-supply *)
   end control-supply
then                         (* i.e., after termination of con-
                                trol-supply *)
   control-mixer
end control-all;

(* After specification of the overall behavior, some details
   are described. The order is irrelevant, and objects al-
   ready specified before may be extended. Note the
   references to z, a, and b in the text below.          *)

procedure generate-weighing-signals:
produces scales-reading      (* scales-reading is a buffer  *)
   where $ messages express that the scales have reached
           !z, or !a, or !b. $;
end;
```

```
type scale-values: values z, a, b end;
block weigh A:
$ controls the value !VA for liquid A $;
sequential
   weigh-A-start:
   writes VA-signal where $ becomes open $;
   end
then
   weigh-A-wait:
   consumes scales-reading
      where $ a check should make sure that level !a has
been reached. Otherwise, some error-handling is required. $;
   end
then
   weigh-A-end:
   writes VA-signal where $ becomes close $;
   end
end weigh-A;

block brick-supply:
sequential
   belt-start
then
   belt-run:
   consumes brick-signals where $ two bricks are waited
                                                    for $;
   end
then
   belt-stop  (* actions for controlling the belt are not yet
                                                    specified *)
end brick-supply;

block control-mixer:
sequential
   switch-on-mixer
then
   wait-mixing-time:  (* specifies the interval for mixing *)
   takes 1 of mix-delay
   end
then tip-mixer
then return-mixer
then switch-off-mixer
end control-mixer;
procedure control-all:
started-by dcy-pushbutton    (* supplements the previous
                                                    definition *)
end;

trigger dcy-pushbutton:
$ contains one token every time the button is pressed $;
capacity 1;  (* means that not more that one request is
                                                    stored *)
skip-input;  (* means that additional signals from the op-
                                                    erator are skipped *)
interface;   (* means that this trigger is located inside the
                control-system, but may be accessed from
                outside as well. Here, e.g., it is filled from
                outside. Therefore, a formal check for
                completeness of the accesses is not
                                                    possible. *)
end;

trigger brick-signals:
$ contains one token for every brick falling from the belt $;
external;    (* means that this trigger is located outside
                the control-system; although it has not to
                    be implemented, it may be accessed *)
end;

buffer scales-reading:
of-type scale-values
end;

(* Only one module is specified here. If the specification
   were further refined, submodules could be defined, for
   instance for all components which take care of the
   liquids. *)

module control-system:
comprises   procedure   control-all
and         trigger     dcy-pushbutton
and         trigger     brick-signals
and         module      liquid-supply-system:
            comprises   block weigh-A
            and         block weigh-B
            and         buffer      scales-reading
            and         type        scale-values
            (* scales-reading and scale-values are acces-
               sible only from within module liquid-
               supply-system                        *)
            end liquid-supply-system
end control-system.
```

The example demonstrates some 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 not only to describe objects but also to add some information to the actions. The fact that many blocks are not yet refined, or that procedure generate-weighing-signals is not yet related to the top level procedure control-all does not matter as long as the specification is still being prepared.

IV. The Formal Definition of ESPRESO-S

Nobody can expect the analyst 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 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 specification language 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 further reason is that the semantics of a nonoperational language cannot 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.

1) Some specifications will be accepted by the tool, while others will not. 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.

Instead of BNF (Backus-Naur form) [6], which is easy to write and read, but limited to context free syntax, a so-called extended attribute grammar [7] was used. EAG's differ from the general attribute grammars [8] mainly by their particularly concise notation.

An EAG can be directly obtained from a BNF-grammar by adding so-called attributes. Every attribute of a particular syntactical variable of the grammar (e.g., the syntactical variable "statement") carries information:

- either on the environment of this construct (e.g., the statements before and after this statement or the name of the program);
- or about the actual value of the construct (e.g., what kind of statement, referring to which objects).

These types of attributes are called *inherited* and *synthesized*, respectively. See the Appendix for examples.

2) When a syntactically correct specification is processed by the tool ESPRESO-W, all redundant or meaningless information is discarded. For example, if an object is specified twice (which is not regarded as an error), only one definition is stored, comprising the union of the two sets of information. The mapping from the specification to its stored image is called *semantics*. Both the implementer and the user of the tool need to know the semantics of ESPRESO-S; therefore, it was defined as precisely as the syntax. This definition was almost for free, because one of the attributes required for the syntax definition can be regarded as the accumulated content of the specification up the the actual point of analysis. Thus, when the whole specification has been analyzed, the attribute contains most of the abstract specification. Extensions of the grammar beyond the syntax were necessary only for information not needed to detect inconsistencies and other errors, but must still be stored (e.g., texts).

3) Finally, and most important for the user, every construct of ESPRESO-S has some *meaning* which, eventually, must be reflected by the ultimate implementation of the specification. Many people tend to take an extreme position on the meaning, saying that it can only be completely defined (as in programming languages) or totally undefined (as in many specification languages). For ESPRESO-S, an attempt was made to find a compromise, based on an improved understanding of what a partially informal specification really can express.

The meaning of a specification written in ESPRESO-S is defined by its mapping onto a well defined language. This (hypothetical) language, called E-Pascal, differs from Pascal in two ways: it supports information hiding, and it comprises calls to a standard real-time operating system [9] for implementing parallelism and synchronization. A more powerful language like Ada, which was not available at that time, could have been used without extensions, but such a mapping would not have shown which particular requirements the specification language imposes on the implementation language and the runtime system.

The mapping, which can only be performed if the stored specification complies with some rules for formal completeness, results in a skeleton of the program. Some parts of the code are ready for compilation (e.g., data and control structures), while basic procedures and blocks need to be refined by the programmer. In particular, actions must be translated into assignment statements and arithmetics. The holes to be filled by the programmer are well isolated, and the programmer is hindered from any unspecified access to media and procedures.

The investigations about the meaning of specifications yielded results which are interesting and encouraging.

First of all, semiformal specifications are shown to have some clear meaning, not merely an intuitive one, which can only be exploited by a human reader. This knowledge should lead to tools which actually implement the transformation conceptualized in this work, or which can check implementations against specifications.

Secondly, many statements in the specifications, though resembling executable statements in programming languages, turned out to have a more static meaning. For instance, the "action"

procedure P *reads* variable V

is mapped onto

P′ (= implementation of P) *is granted access* to V′.

Depending on the program design and on the actual data at runtime, P′ may access V′ several times, or not at all.

Primarily, the investigations about the mapping to a programming language was not done with the intention to implement an automatic tool; it seemed that this would require too much manpower. Now, as ESPRESO is being used, there is a strong demand for an implementation of the mapping, which is hence considered for future realization.

Examples of the definition of ESPRESO-S are given in the Appendix.

V. The Specification Tool ESPRESO-W

The most important components of ESPRESO-W are those required for checking, storing, and modifying specifications. At the current stage, there is no special ESPRESO-editor; a user wishing to create a specification does so by providing a sequential text-file, which is then *converted* into the internal representation, and stored in the so-called ESPRESO-file, a simple substitute for a dedicated data base system. The stored information is called *reference specification*. If it is to be modified, it is first *deconverted*, i.e., translated back into the language ESPRESO-S, edited with a standard text editor, and converted again. Therefore, the ESPRESO-file-management and the programs for conversion and deconversion constitute the kernel of ESPRESO-W (see Fig. 1).

If the tool kept the deconverted parts in the reference specification (as PSA does), the user would be likely to become confused when he entered his modified files, because the tool would report inconsistencies. Our solution to this problem was to make both the tool for conversion and for deconversion

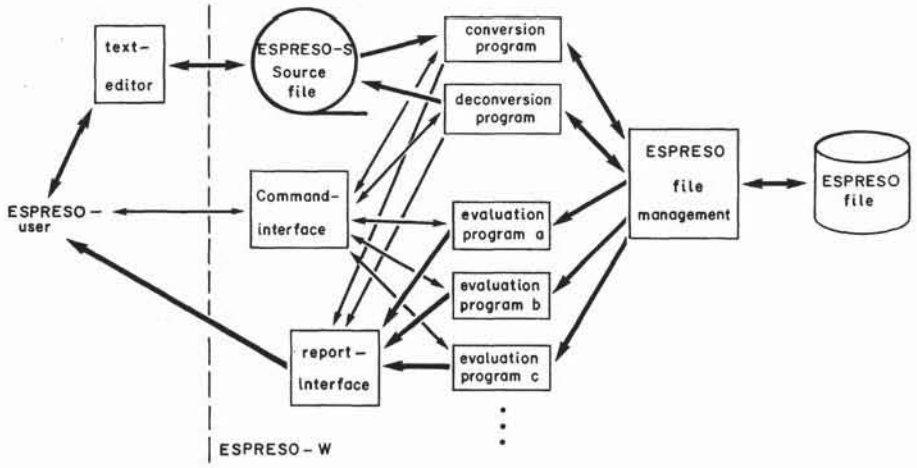


Fig. 1. Structure and data flow of ESPRESO-W.

"consume" their input, when it is transformed. This means that after conversion the source file will contain only those parts of the specification which were rejected, because they contain errors or are not consistent with the reference specification. Symmetrically, after deconversion the reference specification no longer contains the parts which were translated into ESPRESO-S. Loss of data due to misoperation is prevented by automatic generation of safety copies. This symmetrical "either inside or outside" philosophy is very comprehensible.

The information to be deconverted can be selected and limited by the user in various ways, e.g., he may choose certain objects or all objects related to certain keywords.

Desirable functions for evaluating the specifications are listed in [10, 3.3, pp. 43 ff.]. Their global aims are

- to check automatically if the reference-specification satisfies some rules which were intentionally not incorporated into the grammar, for instance that a name must not be referenced outside its range,
- to extract information for manual checks, e.g., information on the data flow,
- to generate documentation and descriptions of subtasks, which can be passed to programmers etc.

Since the ESPRESO-file was designed and implemented as an abstract data structure, which is accessed only via a given set of operations, implementation of the evaluation programs was easy. All programs were written in Pascal.

Theoretically, the program for conversion could have been automatically generated from the EAG. Even though we had access to such a generating system, we did not consider its use, because ESPRESO-W was required to fit into a minicomputer, and still perform reasonably. Nevertheless, the EAG was very useful for implementation [11]. No questions ever arose due to ambiguities, and implementation was easily separated as a task for a master's thesis. The structure of the grammar was used as a guide for structuring the programs; while the basic constructs of the language (sections) are handled by special code, the large number of similar statements is converted (parsed) and deconverted by table driven routines. As the syntax is recursive, these programs are recursive as well.

VI. Evaluation of ESPRESO

With regard to the goals stated in the opening, ESPRESO is obviously far from being perfect, but in some aspects it seems to be superior to other systems.

Informal statements are possible, formalization is supported. The absence of arithmetics and other elements of programming-in-the-small prevents the analyst from getting lost in details. The tool makes sure that all information is collected within one central ESPRESO-file.

The language ESPRESO-S is simple and well defined. It was shown that (and how) the specifications can be mapped into programming languages. Graphical representations were considered, but are not yet fully defined.

The tool-set ESPRESO-W, as far as it is implemented, fulfils its purpose, although many more components are desirable. It is flexible and portable, and can be applied to identify logical links between specified objects, even those which exist only in the informal parts, thereby easing software modifications. Under realistic conditions, however, more support and control is necessary to force all changes being implemented and documented at all levels affected. Only an integrated set of tools which controls all accesses to every kind of documentation, including the code, will meet this goal.

Describing the environment is not really possible in ESPRESO, and it seems that no solution for this problem is within reach. PAISLey was reported to achieve this aim [3], but it seems that the environment is only simulated on a fairly low level, which is possible in any simulation language. What is required is a problem oriented language which allows to describe the environment in a most natural way.

Parallelism and communication is well taken care of in ESPRESO (see below). Time can be specified, but is not yet fully logically integrated into the concepts. The reason is that, according to our experience, timing conditions tend to be rather complicated. Constructs for specifying them would thus be in conflict with the goal of simplicity. As a compromise, informal extensions are allowed within those statements requiring timing constraints.

Comparing ESPRESO with similar systems, it can be classi-

fied as a descendent of the PSL/PSA-SREM-family. All these were designed to support the process of formalization. The most important characteristics of ESPRESO can be found in the approach to communication and synchronization. In PSL, only very little is present for dynamics, while RSL supports the distinction of sequential and parallel activities. But communication on the basis of the stimulus/response-concept is clumsy and tends to obscure the dynamic structure of the system. The set of media introduced in ESPRESO is in my opinion both problem oriented and elegant. A second difference is that ESPRESO has the advantage of precise definition. On the other hand, the set of tools in ESPRESO-W is very limited, compared to its ancestors.

The other mighty family of specification systems comprises those which are based on rigorous concepts, such as algebraic specification. HDM/SPECIAL and HOS represent this philosophy.

Between these two groups, there is a stimulating dichotomy: While the latter is based on formal languages, adapted to the needs of requirements analysis, the former are genuine specification languages, tuned for the largest possible formality. Regarding the definition of ESPRESO, it seems to be close to the threshold on one side; on the other side, PAISLey [3] may be its counterpart. Both approaches have complementary advantages and deficiencies: While the more formal languages lend themselves naturally to all formal operations, like automatic or controlled program generation, simulation, and verification, the semiformal ones fit better to the rather chaotic ideas about systems which are not yet fully conceptualized. Time will show if one is superior over the other, or if both should be applied in sequential steps, or if they even can be combined.

VII. Current State and Future Plans (in August 1982)

The kernel of ESPRESO-W was operational in 1981. Recently, some tools for evaluation became available (report on the hierarchy of procedure-calls, identification of not explicitly defined objects), others are being implemented and will soon be installed (report on the hierarchy of modules, checks for names used outside their legal ranges, data flow report).

ESPRESO-W was first implemented on a minicomputer (SIEMENS R-20); it is now installed both at Karlsruhe (under IBM/OS) and at Brown Boveri Research Center, Baden, Switzerland (under VAX/VMS). The former system is being used for the specification of a protection system for a nuclear power plant, while the latter serves as a handy experimental system.

The current implementation was basically completed by one person, so the only goal was to prove the feasability of the concepts, while practical considerations were less important. Here at BBC, we can test how well our ideas fit to the industrial reality. Two problems attract our particular attention, the acceptance of the tool, and its ability to manage very large specifications.

A system is not acceptable if it suffers from low availability, poor response time, an unreliable database, fragmentary manuals, or a clumsy command language. A primitive substitute for a data base is insufficient for voluminous data. While the scientist may brush aside these deficiencies, they have a disastrous meaning in an industrial environment. Therefore, we must provide material for users' training, build a comfortable interface with graphics, and incorporate a proper database system into ESPRESO-W.

As indicated above, the grammar of ESPRESO-S specifies the conversion program. Since this specification turned out to be both precise and efficient, it would be useful to extend it for deconversion, evaluation, and report generation. Some preliminary investigations showed that this would not cause any problems.

VIII. Conclusions

The work on ESPRESO yielded results on three levels.

- Different from many other specification systems, ESPRESO is based on a carefully chosen set of concepts. Most of the attempts to modify its concepts in response to some practical problems failed, because the modifications would have destroyed the simple elegance of the system. The lesson from this experience (which is not at all a new one) is that a consistent set of concepts must be defined in the very beginning; while it cannot be created later, it can easily be ruined.
- From the author's point of view, the formal definition of ESPRESO-S is the most important result. It proves that such a definition is not a sport for theorists, but a comparatively small investment which immediately pays. Whoever proposes a new specification language without providing a complete formal definition of it, commits an error. How can he convince anybody to prepare a specification of his system, if he himself does not specify his language and its tools? (see [12]).
- Using Pascal for implementation demonstrated the superiority of this language compared to Fortran and Algol 60, but it also exhibited some difficulties due to the absence of modules. Modules are necessary for implementing large software systems according to the principle of information hiding, which contributes significantly to software quality. The ideas of Pascal do not conflict with this principle, and most Pascal-compilers actually support some kind of modules, but their differences decrease the portability of programs. It would be more desirable to have a standard for modules and separate compilation in Pascal [11]. Modula-2 [14] seems to meet this requirement.

Results of investigations on the process of specification and on its relation to the design process, which were a basis of the development of ESPRESO, were published elsewhere [13].

Appendix
Examples of the Formal Definition of ESPRESO-S

This Appendix contains simplified examples from the definition of ESPRESO-S, and also a sample from the original definition.

A. The Grammar of ESPRESO-S

1) Definition of the Syntax: 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 the 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 excerpt, 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. This can be achieved, however, 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 the 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.

Note that the distinction between *inherited* (down) and *synthesized* attributes (up) is made only for the reader's convenience.

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

2) Definition of the Semantics: 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 φ NEW-INFO)>.
```

The last parameter is the new context, which consists of the inherited context plus the information derived from the application of production rules on "variable-name." ϕ is an operator, especially defined for this grammar. The meaning of ϕ is similar to the union-operator for sets. But if all the information in the specification would just be entered into the context-attribute, it might become inconsistent. Therefore, SET-A ϕ SET-B is the union of the sets, if they are consistent; otherwise, the result is undefined (e.g., "X is a procedure" ϕ "X is a data"). An undefined result means an error-message during conversion. Otherwise, the information is superimposed, and only the consistent, nonredundant subset of the result is kept. Thus, the context attribute can never become redundant or contradictory.

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

3) A Typical Production Rule: Below, the application of one typical production rule, which was not simplified, is demonstrated. This rule was taken from [10, p. 120]:

```
<following blocks                ↓ C1 ↓ BLOCK1 ↓ PROCK
                                   ↑ C2>
  ::= empty <where C2 = C1 φ (sequent,BLOCK1)→
                                   (PROCK,undef)>          /r2/
    | <following block           ↓ C1 ↓ BLOCK1 ↓ PROCK
                                   ↑ BLOCK ↑ C>
      <following blocks          ↓ C φ (sequent,BLOCK1)→
                                   (PROCK,BLOCK)
                                   ↓ BLOCK ↓ PROCK
                                     ↑C2>.                 /r2/
```

This rule is required to produce lines 3 and 4 in the following excerpt from a specification:

```
1) block ABC:
   ···
2) sequential block ALPHA: ··· end ALPHA
3) then     block BETA:    ··· end BETA
4) then     block GAMMA:   ··· end GAMMA;
   ···
5) end ABC.
```

The ellipses indicate statements which are not relevant here.

The production rule can be explained as follows:

"following blocks" is either empty, or one "following block," which in turn is followed by more "following blocks." This is the context-free syntax. In the example above, the second alternative is applied once to produce lines 3 and 4 together, and then once again for line 4 only; finally, the first alternative is used, since the statement ends in line 4.

The attributes in this production are:

- PROCK — the upper level procedure or block (in this example: ABC),
- BLOCK1 — the previously defined block (first ALPHA, then BETA),
- BLOCK — the first block following BLOCK1 (first BETA, then GAMMA),
- C1, C und C2 — various versions of the context.

C1, PROCK, and BLOCK1 represent inherited attributes, since they are defined by the (left-) context. BLOCK is the name of an object produced from "following block," thus being a synthesized attribute there, whereas it is used by the "following blocks" as an inherited attribute.

"where C2 = · · · " defines the value of C2 and does not produce any terminal symbols.

By the ϕ-operation, the context-attribute is extended, before it is passed on to "following blocks." If (sequent, BETA) is already linked to objects other than GAMMA or the pseudo-object undef, the specification is incorrect.

"/r2/" is only a reference to a list of relations. Other references help to find definitions which are not located in the same paragraph.

The reader should note that the word "attribute" is used here in a slightly imprecise way. BLOCK, C1, etc. are in fact not attributes but the names of meta-variables, which must be replaced by actual attributes, when the grammar is used to produce any specification. The relation between the name of a meta-variable and its value (i.e., the attribute) is the same as the relation between name ("X") and value ("2.17") of a variable used in a program.

The following example (in German) is taken from [10], page 122 f. The complete definition of ESPRESO-S covers 18 pages, not including the definitions and explanations of notations and symbols.

```
(***   4.4 in Kap.7 (Anh.)  Puffer- und Trigger-Angaben   ************)

(***   4.4.1 in Kap.7 (Anh.)  Puffer-Angaben   ************************)

< Puffer-Angabe                      ↓ K1 ↓ PUFFER ↑ K2 >
 ::= < Puffer-Trigger-Angabe         ↓ K1 ↓ PUFFER ↑ K2 >          /4.4.3/
   ] < Puffertyp                     ↓ K1 ↓ PUFFER ↑ K2 >
   ] < statischer Speicherbedarf     ↓ K1 ↓ PUFFER ↑ K2 >
   ] < dynamischer Speicherbedarf    ↓ K1 ↓ PUFFER ↑ K2 >
   ] < Pufferorganisation            ↓ K1 ↓ PUFFER ↑ K2 >.

< Puffertyp          ↓ K1 ↓ PUFFER       ↑ K2 >
 ::= < Typzuordnung  ↓ K1 ↓ PUFFER ↑ TYP ↑ K2 >.                   /5.1/

< statischer Speicherbedarf  ↓ K1 ↓ PUFFER ↑ K2 ϕ (smr,PUFFER)→ANZAHL >
 ::= stat-memory-requ-Symbol                                       /r42/
     < Zahlenangabe          ↓ K1 ↑ ANZAHL ↑ K2 >.                 /4.9/

< dynamischer Speicherbedarf ↓ K1 ↓ PUFFER ↑ K2 ϕ (mpi,PUFFER)→ANZAHL >
 ::= memory-per-item-Symbol                                        /r43/
     < Zahlenangabe          ↓ K1 ↑ ANZAHL ↑ K2 >.                 /4.9/

< Pufferorganisation              ↓ K1 ↓ PUFFER ↑ K2 >
 ::= fifo-Symbol
     < wobei K2 = K1 ϕ (überlauf-verhalten,PUFFER)→fifo   >        /a5/
   ] by-priority-Symbol
     < wobei K2 = K1 ϕ (überlauf-verhalten,PUFFER)→prior  >.       /a5/

(***   4.4.2 in Kap.7 (Anh.)  Trigger-Angaben   ********************)

< Trigger-Angabe                 ↓ K1 ↓ TRIGGER ↑ K2 >
 ::= < Puffer-Trigger-Angabe     ↓ K1 ↓ TRIGGER ↑ K2 >          /4.4.3/
   ] < Verzögerung               ↓ K1 ↓ TRIGGER ↑ K2 >
   ] < Periode                   ↓ K1 ↓ TRIGGER ↑ K2 >.

< Verzögerung ↓ K1 ↓ NAME
                 ↑ K2 ϕ (verzög,NAME)→(FRIST,ANZAHL) >             /r26/
 ::= delay-Symbol < Zeitangabe ↓ K1 ↑ FRIST ↑ ANZAHL ↑ K2 >.       /4.8/

(* NAME ∈ {TRIGGER,PROZEDUR}                                        *)

< Periode      ↓ K1 ↓ TRIGGER
                 ↑ K2 ϕ (periode,TRIGGER)→(FRIST,ANZAHL) >        /r27/
 ::= cycle-Symbol < Zeitangabe ↓ K1 ↑ FRIST ↑ ANZAHL ↑ K2 >.       /4.8/

(***   4.4.3 in Kap.7 (Anh.)  Angaben zu beiden Arten   ***********)

< Puffer-Trigger-Angabe          ↓ K1 ↓ PUGGER ↑ K2 >
 ::= < Restriktion ↑ RESTRIKT    ↓ K1 ↓ PUGGER ↑ K2 >          /5.2/
     < test RESTRIKT ∈ {liefer-r,hol-r,start-r,maskier-r} >
   ] < Kapazitätsangabe          ↓ K1 ↓ PUGGER ↑ K2 >
   ] < Pufferverhalten           ↓ K1 ↓ PUGGER ↑ K2 >.

< Kapazitätsangabe  ↓ K1 ↓ NAME ↑ K2 ϕ (kapazität,PUGGER)→ANZAHL > /r41/
 ::= capacity-Symbol
     < Zahlenangabe ↓ K1          ↑ ANZAHL ↑ K2 >.                /4.9/

< Pufferverhalten                ↓ K1 ↓ PUGGER ↑ K2 >
 ::= skip-input-Symbol
     < wobei K2 = K1 ϕ (überlaufverhalten,PUGGER)→skip  >         /a4/
   ] block-input-Symbol
     < wobei K2 = K1 ϕ (überlaufverhalten,PUGGER)→block >.        /a4/
```

B. *Definition of the Meaning*

Most units of the specification like variables, buffers, etc. can be transformed into complete declarations, which the programmer need no longer access. 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 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.

ACKNOWLEDGMENT

The basic work on ESPRESO was carried out at the Nuclear Research Center, Karlsruhe, West Germany, where the implementation of ESPRESO-W is being continued. K. Eckert's master's thesis layed the foundations of this implementation. The theoretical work was supported by Prof. R. Baumann and Prof. F. L. Bauer of the Technical University of Munich.

Many ideas were taken from other systems, in particular from PSL/PSA [1], SREM [14], and MASCOT [15].

My colleagues of the Computer Science Group at Brown Boveri Research Center continuously contribute to the work by their discussions and comments. The referees' comments definitely helped to improve this paper.

REFERENCES

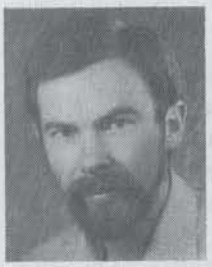
[1] D. Teichroew and E. A. Hershey III, "PSL/PSA: A computer aided technique for structured documentation and analysis of information processing systems," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 41–48, 1977.

[2] J. Ludewig, "Process control software specification in PCSL," in *IFAC/IFIP Workshop on Real-Time Programming*, V. Haase, Ed. Graz, Apr. 1980. Pergamon, 1980, pp. 103–108.

[3] P. Zave, "An operational approach to requirements specification for embedded systems," *IEEE Trans. Software Eng.*, vol. SE-8, pp. 250–269, 1982.

[4] E. Knuth and P. Rado, "Principles of computer aided system

description," Hungarian Acad. Sci., Budapest, Hungary, Rep. 117/1981, 1980.
[5] P.P.-S. Chen, "The entity relationship model–Towards a unified view of data," *ACM Trans. Data Base Syst.*, vol. 1, pp. 9–36, 1976.
[6] P. Naur, Ed., "Revised report on the algorithmic language ALGOL 60," *Numerische Mathematik*, vol. 4, pp. 420–453, 1963.
[7] D. A. Watt, "An extended attribute grammar for PASCAL," *SIGPLAN Notices*, vol. 14, no. 2, pp. 60–74, 1979.
[8] D. E. Knuth, "Semantics of context-free languages," *Math. Syst. Theory*, vol. 2, no. 2, 127–145, 1968; also vol. 5, no. 1, pp. 95–96, 1971.
[9] Th. Lalive d'Epinay, Ed., "TC 8 up to date report," presented at the European Workshop on Industrial Comput. Syst. (EWICS), Tech. Committee on Real-Time Oper. Syst., paper I-1-8, 1979.
[10] J. Ludewig, "Zur Erstellung der Spezifikation von Prozessrechner-Software," Doctoral dissertation, Tech. Univ. Munich; reprinted as KfK-Rep. 3060, Kernforschungszentrum Karlsruhe, 1981.
[11] K. Eckert and J. Ludewig, "ESPRESO-W, ein Werkzeug für die Spezifikation von Prozessrechner-Software," in *Werkzeuge der Programmiertechnik*, G. Goos, Ed. Springer-Verlag, 1981, pp. 101–112.
[12] J. Ludewig, "Specification of a specification language," in *IFAC/IFIP Workshop on Real-Time Programming*, T. Hasegawa, Kyoto, Japan, 1981. Pergamon, 1982, pp. 63–68.
[13] K. Eckert and J. Ludewig, "Computer aided specification of process control software," *IEEE Computer*, pp. 12–20, May 1982.
[14] N. Wirth, *Programming in Modula-2*. Springer-Verlag, 1982.
[15] M. Alford, "A requirements engineering methodology for real time processing requirements," *IEEE Trans. Software Eng.*, vol. SE-3, pp. 60–69, 1977.
[16] K. Jackson and H. F. Harte, "The achievement of well structured software in real-time applications," in *IFAC/IFIP Workshop on Real-Time Programming*, Rocquencourt, June 1976. Pergamon, 1976, pp. 229–238.



Jochen Ludewig received the M.S. degree in electrical engineering from the Technical University Hannover, and the Postgraduate Certificate and Ph.D. degrees in computer science from the Technical University Munich.

From 1975 until 1980, he was at the Institute for Industrial Data Processing, which is part of the Nuclear Research Center, Karlsruhe, West Germany. There, his main activity was on software documentation and specification, including the installation and extension of PSL/PSA. Since 1981, he has been a member of the Computer Science Group at the Brown Boveri Research Center, Baden, Switzerland.

Dr. Ludewig is a member of Gesellschaft für Informatik (German Society for Informatics) and EWICS (European Workshop on Industrial Computer Systems), Technical Committee on Application Oriented Specification. He takes part in the Study Group on Ada for Specifications, which is sponsored by the European Community.