PROCESS CONTROL SOFTWARE SPECIFICATION IN PCSL

J. Ludewig

Kernforschungszentrum Karlsruhe GmbH, Institut für Datenverarbeitung in der Technik, Postfach 3640, D-7500 Karlsruhe, Federal Republic of Germany

Abstract. This paper presents the most important features of PCSL, a language dedicated to the specification of process control software. PCSL is primarily intended to improve software reliability.

Simple structures of both active system components and data are supported by the language. Different constructs for static data, for dynamic data moving through the system (data flow), and for resources permit easy modeling of the situations common in process control applications. PCSL is accepted by an advanced, table-driven version of the well known PSA.

As an example, a simple system for data collection is partially specified in PCSL. Finally, the current state is indicated, and limitations are discussed.

Keywords. Computer programming; process control; software specification; parallel processes; synchronization; software engineering.

BACKGROUND AND GOALS

Our institute is a division of a nuclear research center. Therefore, some part of our manpower is devoted to research on specification, design, test, and maintenance of process control software, in order to improve software reliability.

This paper concentrates on a language for software specification. We expect a formal language to be used early in program development not only to decrease the number of errors caused by insufficient documentation, but also to improve software quality through restricting the analyst to a set of constructs which are easy to use and easy to transfer into programming languages.

PSL/PSA

After surveying the best-known languages and tools for specification and design of software (Ludewig, Streng, 1978), we decided to try the PSL/PSA-system from ISDOS-Project, University of Michigan, USA (Teichroew, Hershey, 1977). It was installed in 1977, and put to test to find out whether or not it is applicable in our environment.

PSL/PSA stands for "Problem Statement Language" and "Analyzer". PSA is a large program of some 50,000 lines of code (mainly written in FORTRAN IV).

It allows the users

- to enter system descriptions written in PSL into the PSA-Database in an arbitrary number of sessions,
- to delete or change information in the DB,
- to generate reports from the DB.

PSL, the input-language for PSA, is based on a model describing systems by objects and their relations. There are predefined sets of object-types (e.g. PROCESS, INPUT, ENTITY, RESPONSIBLE-PROBLEM-DEFINER) and relation-types (e.g. "uses" and "derives", "subpart", "responsible for"). Only objects of certain types are allowed in every component of a relation.

At our institute, we observed some difficulties with PSL when used for process control software. They were caused by

- a lack of means to describe dynamic properties of systems (including parallelism and synchronization),
- a rather unprecise conception of data-flow in PSL,
- a hierarchy of data-object-types styled for commercial systems rather than for process control software.

Users also expected the language not only to allow for documentation and communication but also to guide them to a well structured, unambiguous design which is easy to implement. PSL does not have such a bias to support any particular method of design.

THE GENERALIZED ANALYZER

In 1977, ISDOS announced a new system called the Generalized Analyzer (GA). The GA (Teichroew, 1978; Kang and others, 1977) is an extension of PSA. While PSA was designed just for PSL, the GA is table driven, allowing the user to provide his own language definition. Compared to PSA, there is a second level called the META-system on top of the first one (see fig. 1). The META-system has the same structure but its DB contains the language tables to control the GA.

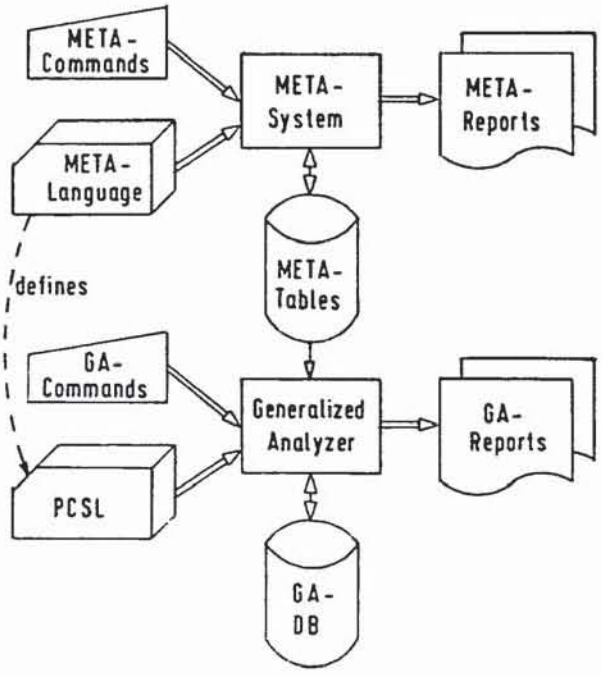

Fig. 1. META-System and Generalized Analyzer.

In the META-input, the user defines symbols, object-types and relation-types for his own language.

BASIC IDEAS OF PCSL

After learning about the GA, we started to develop a language called PCSL (Process-Control-Software Specification-Language) in 1978. Under the conditions imposed by the META-system, we tried to meet the following requirements:

PCSL should be small (i.e. all elements of minor importance in process control applications are eliminated, e.g. the RESPONSIBLE-PROBLEM-DEFINER). It should support simple structures, hierarchies in particular, but prevent obscure design.

PCSL should include a set of constructs for parallelism and coordination of tasks. Every element of the language should be well defined.

The language is intended to be really problem-oriented, i.e., the elements of PCSL are not abstractions of programming language constructs, but of situations common in process control software. Therefore, formal completeness and minimality are not aimed at.

Beside PSL, several languages influenced PCSL, above all RSL (Alford, 1978) and MASCOT (Jackson, 1977).

Process Structures

In PCSL, active subsystems (processes, tasks, procedures) are called ACTIVITIES.[1] ACTIVITIES may contain or call other ACTIVITIES similar to the PROCESSES in PSL, but PCSL provides more restricted and thus more precise types of decomposition:

There are four hierarchical relations among ACTIVITIES, named "subact", "substep", "alternative", and "utilize".

Subact-relation. If an ACTIVITY A_0 is refined by sub-activities A_1 through A_n, execution of A_0 means parallel execution of A_1, A_2, ..., and A_n (see fig. 2). When they all have terminated, A_0 will terminate. Thus, the subact-relation provides a restricted structure for parallelism.

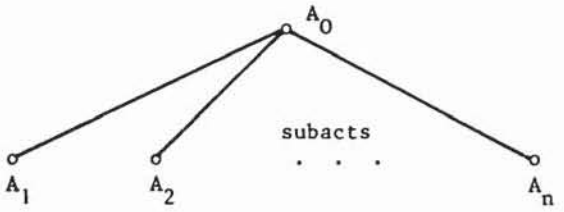

Fig. 2. Subact-Relation.

Substep-relation. If an ACTIVITY A_0 is decomposed into substeps S_1 through S_n, this implies sequential execution of S_1, then S_2, then ..., then S_n, when A_0 is executed (see fig. 3). Due to limitations of the META-system there is no way to enter the substeps as an ordered set, so we had to introduce two more relations named "init step" to identify the first one and "next" to define the sequence of execution. We decided to use a special object-type STEP instead of ACTIVITY for the substeps to achieve more checking on input. Except this difference between ACTIVITIES and STEPS, they are very similar. Most statements about ACTIVITIES in this paper are valid for STEPS as well.

[1] Capital letters are used for object-types and objects throughout this paper.

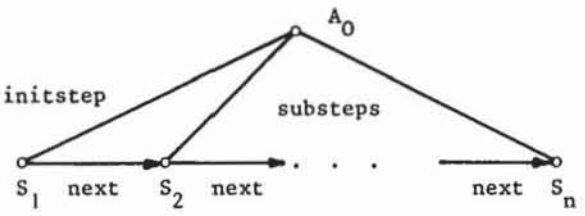

Fig. 3. Substep-Relation.

Alternative-relation. When one out of several different ACTIVITIES is executed depending on the value of some selection-function, the alternative-relation is used (see fig. 4).

When A_0 is executed, (at most) one out of A_1 through A_n is executed.

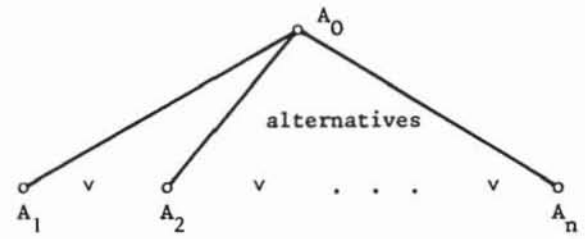

Fig. 4. Alternative-Relation.

Utilize-relation. The three relations described above require the system to be tree-structured, i.e. no ACTIVITY (or STEP) may be used by two "father"-ACTIVITIES. To allow modelling of procedures common to several ACTIVITIES, the relation "utilize" may be used (see fig. 5).

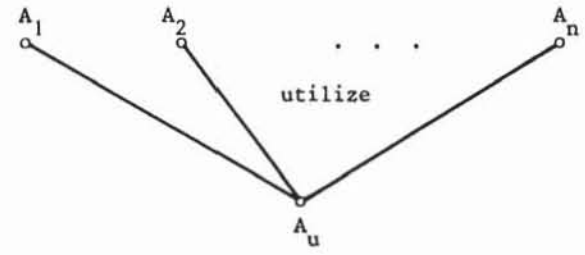

Fig. 5. Utilize-Relation.

While the father may have several sons when decomposed by "subact", "substep", or "alternative", but his sons must not have more than one father, the rules are just reversed for "utilize".

Since every ACTIVITY is decomposed in at most one of the four patterns described above, the user is forced to define a clear system structure.

Data and Implicit Synchronization

The problems about synchronization in application software may be classified in three groups:

- mutual exclusion, which is necessary when resources are shared among parallel processes,
- producer/consumer-coordination which has to be provided for parallel processes which perform sequential transformations on data,
- reader/writer-coordination which must be done when parallel processes access data writing or both reading and writing at the same time.

Programming languages for process control either contain a simple but powerful mechanism which can be used for any kind of problem (e.g. semaphors or monitors) or a combination of several constructs dedicated to particular types of problems.

In PCSL, we tried to avoid any such language element by defining data types with implicit synchronization. Whenever an ACTIVITY occupies a RESOURCE, mutual exclusion is implied. When the ACTIVITY delivers items to a BUFFER ("produces"), it will implicitely be delayed when there is no space for the items. The same applies when an ACTIVITY tries to fetch items from a BUFFER ("consumes") which is empty. Finally, while an ACTIVITY changes the value of a VARIABLE ("writes"), all other attempts to read or write that VARIABLE are delayed.

VARIABLES may be defined by hierarchical refinement or by TYPES, which again may be decomposed, forming records (= structures) or arrays, unless they are pointers or primitive types.

Readers should notice the difference between BUFFERS and RESOURCES. An object which just provides space to store items of various types, has to be a RESOURCE in PCSL. In a BUFFER all items are of the same type, and they are removed ("consumed") under a predefined strategy, e.g. fifo or by-priority. Thus, user defined names of BUFFERS will describe the content rather than the container, e.g. "messages", "raw-data", etc., just like the label on a box or on a barrel does.

RESOURCES may also be used to define critical sections. E.g., when the updating of several VARIABLES is not allowed to be interrupted, a RESOURCE is defined to be the abstract "roof" of these VARIABLES, and the access can be protected by occupying that RESOURCE.

Programmers who are used to look for a "clever" way of using constructs of a language might try to exploit the full power of buffers, e.g. to misuse them as semaphors. This is not intended with PCSL. Users should try to have enough of the simple constructs in PCSL, when used properly, rather than to outwit the system (and, in most cases, themselves).

Other Elements of PCSL

Events. There is an object-type EVENT in PCSL for any source of events influencing the execution of ACTIVITIES. For two sorts of events, those caused by interrupts and those from the clock, there are special types named INTERRUPT and TIMER.

For TIMERS, a delay and/or a cycle may be defined. The events will happen first after the delay and then after every cycle.

Events may terminate the execution of an ACTIVITY (provided it is executing). If an ACTIVITY has the property "cyclic", every execution is started by an event (usually a TIMER).

In a hierarchical structure, it is difficult to define the meaning of terminating an ACTIVITY which is decomposed into other ACTIVITIES. Either all descendants have to be terminated (which is practically impossible) or the dynamic structure of the system will not be hierarchical any more. Also, termination of elementary ACTIVITIES which access RESOURCES, BUFFERS or VARIABLES, will be difficult to implement. Therefore, termination will be actually performed by the affected ACTIVITY itself, when it gets to a well defined state, usually at the end of a cycle.

Starting has to be handled with care as well. Every cyclic ACTIVITY is assumed to be contained within an additional ACTIVITY that is executed at its time (i.e. when its father is executing etc.) Only when this "container" is activ and the ACTIVITY itself is not, the latter may be started.

Logical objects and relations. CONDITIONS are special (boolean) variables that can be used to control selection for alternative-relations. Also, continuation of a cyclic ACTIVITY may depend on a CONDITION.

Subranges of variables are used in the same way, e.g. RANGE-REAL Temperature-OK of REAL Temperature. Subranges and CONDITIONS can be logically combined to form more complex CONDITIONS, e.g. TRUE WHILE Temperature-OK AND Normal-State.

Descriptive elements. Like in PSL, KEYWORDS and ATTRIBUTES with ATTRIBUTE-VALUE may be defined and used in PCSL. Texts for informal descriptions can be attached to any object; for ACTIVITIES, there is a special type of text named "code" which can be used to enter the code as a part of the documentation.

EXAMPLE

This is a very small example to show some important elements of PCSL. A larger example describing a real system can be found in Ludewig (1980a).

The problem can be stated as follows: Every 3 seconds, 100 values have 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; the content of the screen is updated whenever the operator presses a button.

In fig. 6, the problem is re-stated in terms of PCSL.

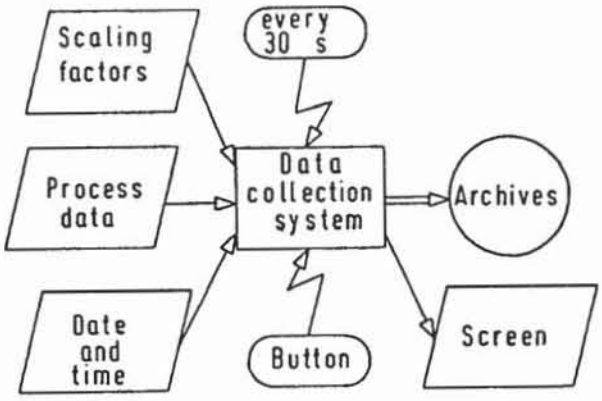

Fig. 6. Outlines of a Data Collection System.

The symbols are used as follows:

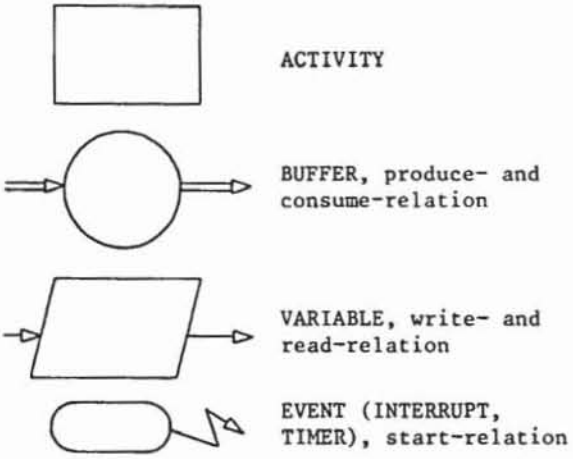

Figure 6 may be translated into PCSL:

```
DEFINE ACTIVITY  Data-collection;
   OBJECTIVE;
   checks data from the technical process,
   puts reduced data to the archives.
   Updates the display when the operator
   pushes a button on the display.
   ;
   EXECUTING REPEATEDLY;
   STARTED BY     Timer1, Button;
   READS          Process-data,
                  Scaling-factors,
                  Date-and-time;
   PRODUCES FOR   Archives;
   WRITES         Screen;

## This is a comment. Comments are not    ##
## entered into the data-base. They are   ##
## lost after input.                      ##
## Note that the text under OBJECTIVE     ##
## is not a comment. It is part of the    ##
## object Data-collection and will be     ##
## stored in the data-base.               ##
```

This is only a first sketch, of course. The problem can then be refined as follows, using the subact-relation:

```
DEFINE ACTIVITY  Data-collection;
   OBJECTIVE;
   ...
   ;
   SUBACTS ARE   Input-processing,
                 Data-reduction-and-storing,
                 Update-screen;

## Property EXECUTING REPEATEDLY has been ##
## removed from this ACTIVITY because     ##
## each of its sons is cyclic (see below).##
```

The sub-ACTIVITIES are defined as follows:

```
DEFINE ACTIVITY  Input-Processing;
   OBJECTIVE;
   checks data from the technical process,
   puts normalized data to a buffer. Also,
   a variable containing the actual state
   is updated.                              ;

   EXECUTING REPEATEDLY;
   STARTED BY    Timer1;
   READS         Process-data,
                 Scaling-factors,
                 Date-and-time;
   PRODUCES FOR  Normalized-data;
   WRITES        Actual-state;

################################################

DEFINE TIMER     Timer1;
   DESCRIPTION;
   Controls the cyclic data-collection.     ;
   CYCLE         3 OF SEC;

################################################

DEFINE ACTIVITY  Data-reduction-and-storing;
   OBJECTIVE;
   Data from the process is filtered and
   written to a permanent file.
   ;
   DESCRIPTION;
   The filtering is accomplished by a
   Butterworth-low-pass with a cut-off-
   frequency of 30 seconds.
   ;
   EXECUTING REPEATEDLY;
   CONSUMES FROM Normalized-data;
   PRODUCES FOR  Archives;

## Note that this ACTIVITY is controlled  ##
## by the BUFFERS only.                   ##

################################################

DEFINE ACTIVITY  Update-screen;
   OBJECTIVE;
   The actual state is diplayed.            ;
   EXECUTING REPEATEDLY;
   STARTED BY    Button;
   READS         Actual-state;
   WRITES        Screen;
```

Some of the objects used by the above ACTIVITIES are:

```
DEFINE INTERRUPT Button;

################################################


DEFINE VARIABLE  Process-data;
   POSITION PERIPHERAL;
   CONSISTS OF   Measured-value-list;

################################################


DEFINE BUFFER    Normalized-data;
   FULL SKIP;
   CAPACITY      10 OF Data-block;

## That allows for buffering up to 30 sec ##
## when the filtering process is blocked. ##
## From then, data is lost due to the     ##
## property FULL SKIP.                    ##

################################################


DEFINE VARIABLE  Date-and-time;
   DESCRIPTION;
   Date-and-time is available as a system-
   function.                                ;
   POSITION EXTERNAL;
   CONSISTS OF   20 Characters;

################################################


DEFINE VARIABLE  Actual-state;
   CONSISTS OF   Data-block;

DEFINE TYPE      Data-block;
   CONSISTS OF   Header,
                 Measured-value-list;

DEFINE TYPE      Measured-value-list;
   CONSISTS OF   100 Measured-values;

DEFINE TYPE      Measured-values;
   CONSISTS OF   Measuring-point-ID,
                 Value;

DEFINE VARIABLE  Screen;
   CONSISTS OF   20 Lines;

DEFINE TYPE      Lines;
   CONSISTS OF   80 Characters;

################################################
################################################
```

In a real software-development, all other objects had to be defined in a similar way. Some might be further decomposed until a level of detail is reached where PCSL is no longer suitable. Then, the programming-language code may be entered as a text for every ACTIVITY in order to have just one complete documentation of the system.

CURRENT STATE

PCSL was first defined in 1978. A very early and partially incomplete version of the GA became operational in June 1979. Experiences from tests resulted in PCSL.2 (October 1979). PCSL.3 will be soon available, differing from PCSL.2 by some minor improvements.

PCSL/GA was used for small tests and for two fairly large examples developed from real software projects at our institute as part of a master thesis.

LIMITATIONS OF PCSL

Though we can define object- and relation-types and also the statements to express relations, there are many restrictions for any language defined with the META-system.

From our point of view, the most desirable features not (or not yet) available are means to define a hierarchical (i.e. recursive) syntax, to limitate the scopes of names, and to supply more information about semantic restrictions for the GA.

A hierarchical syntax is desirable because it provides the easiest and least error-prone way to describe hierarchical structures.

Limited scopes of names are necessary to prevent naming conflicts and to allow definition of modules.

Semantic restrictions would guarantee the content of the GA-DB to be formally correct. Today, we cannot make sure that the user will define and enter correct structures only, because we cannot check it (at least not at input-time).

Finally, it would be nice to have a software-specification system that uses the same machine as the planned software will do. Since we do not use large main-frames but rather small computers, such a system had to be much smaller than the GA.

FUTURE DEVELOPMENT

A system called ESPRESO which is based on similar concepts as PCSL but meets most of the requirements stated above, is currently being implemented (Ludewig, 1980b).

When ESPRESO was developed, some of the concepts taken from PCSL were improved. So there will be some feedback to PCSL, when ESPRESO is finished and tested. Two concepts will probably be affected:
Events (EVENT, TIMER, INTERRUPT) will be regarded as messages that do not only exist at one point of time but remain existent until they have been received. Since messages need BUFFERS, there will be some special type of BUFFER for events.

Logical objects and relations will be reduced, because in the present version expressions are allowed for CONDITIONS but not for other variables. Since expressions do not fit well into a nonprocedural specification-language, CONDITIONS will become variables like REALS, INTEGERS, etc., without special relations.

CONCLUSION

PCSL is a nonprocedural language for software specification and design like PSL, but its scope is limited to process control applications. Thus, the gain of simplicity and clearness is paid by a loss of generality. The concepts available in PCSL will guide the users to simple structures. They can easily be translated into a high level language like PEARL. In particular, the language elements for parallelism and implicit coordination allow the user to stay on a high level of abstraction instead of going down to the bit.

Sample applications proved the practicability of PCSL; a full scale evaluation by using PCSL in a real software project is not yet done. Also, decisions about additional tools for special reports taylored to PCSL are still pending.

REFERENCES

Alford, M.W. (1978). Software requirements engineering methodology (SREM) at the age of two. In proceedings of *COMPSAC 78*, Chicago, Ill., November 13-16, 1978.

Jackson, K. (1977). Language design for modular software construction. In B. Gilchrist (Ed.), *Information Processing 77*, North Holland Publishing Company, Amsterdam.

Kang, K., E.A. Hershey, Y. Yamamoto, R. Marcellus (1977). Example of use of META Language, META Generator and Generalized Analyzer. *ISDOS META System Memorandum META-13*, 20 pp., unpublished.

Ludewig, J., W. Streng (1978). *Methods and Tools for Software Specification and Design - A Survey*. European Purdue Workshop, TC for Safety and Security (TC 7), Paper No. 149, 23 pp.

Ludewig, J. (1980a). *PCSL - a process control software specification language*. KfK 2874, Kernforschungszentrum Karlsruhe, 45 pp.

Ludewig, J. (1980b). *ESPRESO - ein Entwicklungssystem für Prozeßrechner-Software*. Doctoral dissertation, TU München, being prepared.

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

Teichroew. D. (1978). Overview of the META-System and the Generalized Analyzer. *ISDOS META System Memorandum META-1*, revised version, 13 pp., unpublished.