PCSL -
A process control software specification language

J. Ludewig
Abstract

This paper presents PCSL, a language for specification and design of process control software, and sketches analysis and reports based on this language. As an example, a system for process-data-collection is specified using PCSL.

PCSL, DSL and EPOS are reviewed with respect to PCSL. Finally, the current state of the project is depicted.

The appendix contains a short version of the PCSL-definition.

PCSL - Eine Prozeßrechner-Software-Spezifikationssprache

Zusammenfassung

Der Bericht beschreibt PCSL, eine Spezifikations- und Entwurfssprache für Prozeßrechnersoftware; Prüfungen und Reports, die auf dieser Sprache basieren, werden skizziert. Als Beispiel ist ein Datenerfassungssystem in PCSL beschrieben.

PSL, DSL und EPOS werden mit PCSL verglichen. Der Bericht schließt mit Angaben zum Stand der Arbeiten.

Eine Kurzfassung der PCSL-Definition ist als Anhang beigefügt.
1. Introduction
   1.1 Needs of software-designers in our environment
      1.1.1 Process control systems
      1.1.2 Safety and security
   1.2 History of PSL at the IDT
2. Basic principles of PCSL
   2.1 The requirements a tool for process-control software-development must meet
   2.2 A concept for active system components
   2.3 A concept for passive system components
   2.4 A concept for time, synchronization, and interaction
   2.5 Other elements of PCSL
3. Analysis and reports
   3.1 Checks for semantic correctness
   3.2 Checks for completeness and consistency
   3.3 Simulation
4. PUES - a sample application of PCSL
5. PCSL versus PSL, DSL and EPOS
   5.1 PSL
   5.2 DSL
   5.3 EPOS
6. Current state of PCSL
   6.1 The language
   6.2 The Generalized Analyzer
   6.3 Analysis-tools
7. References
Appendix: A PCSL-summary
1. Introduction

1.1 Needs of software-designers in our environment

Being a part of a nuclear research center, our task is to develop highly reliable process control systems for various applications in the area of nuclear power plants. Thus, any tool or method for our environment has to meet two requirements:

- it must reflect the particular needs of process control systems, and
- safety and security of the systems to be developed has to be its very first objective.

1.1.1 Process control systems

Process control systems differ from other computer systems, in particular from commercial ones, while operating systems are similar.

A typical process control system

- will work virtually forever, i.e. without a predetermined moment of termination,
- is event - or real-time-controlled, usually in several ways, making it necessary to introduce asynchronous tasks, which are almost (but in most cases only almost) independent from each other,
- is not very complicated from a mathematical or logical point of view, but moves lots of data both internally and across its very wide and complicated interfaces to the technical process and to the operators (dialogues),
- is even less testable than other computer systems due to its indeterminacy.
1.1.2 Safety and security

Our particular demand for safety and security makes it necessary
- to prevent by virtually all means errors from getting into the software,
and
- to detect errors which got there anyway.

The lack of generally accepted methods and tools for the early stages
of software development forced us to work in this field.

1.2 History of PSL at the IDT

In 1977, version 2.1 of the PSL/PSA-System /Teichroew, Hershey 77/ was
installed on our IBM 370/168 which is used for batch-operation; it is usually
controlled via TSO running on our 370/158. (Recently, the 168 has been re­
placed by a 3033, TSO is now on the 168.) In 1978, version 2.1 was replaced
by 4.2.

PSL/PSA was used for a few sample applications. As a result, we found
that batch is not as disadvantageous as expected. Users rather suffered from
difficulties with the language: There are several data-types (set, entity,
group, element, and also input and output) which confused us because in our
field there is no application for them. On the other hand, we have many prob­
lems with control and time which are difficult to describe in PSL if at all.

Thus, we were quite excited when we heard about the META-System because
we felt (and still feel) that this is our chance for a dedicated specifica­
tion-system.

A first version of our language was designed in January 1978 /Ludewig,
Streng - March 1978 (b)/ and processed by the META-Generator in August, the
Generalized Analyzer was installed by the end of July this year (see 6. and
Appendix 2). A revised version (PCSL.2) was introduced in October 1979.

This paper is based on PCSL.3 which contains some minor improvements.
PCSL.3 is being prepared for installation.
2. Basic principles of PCSL

PCSL (Process Control Software Specification Language) was designed after reviewing the most popular approaches in this area /Ludewig, Streng - March 1978 (a) and April 1978/ and with our experiences with PSL and the problems of process control software in mind.

This chapter is not intended to encompass a complete description of PCSL. Its point is only to show the basic ideas.

2.1 The requirements a tool for process-control software-development must meet

Beyond generally used concepts like control structures etc. we need special means to express:

- sequential and parallel action,
- interaction of processes, in particular synchronization and start or termination,
- all relevants types of data flow in sufficient detail,
- sharing and mutual exclusion of resources.

2.2 A concept for active system components

When a large software system is decomposed, a hierarchy of programs will emerge.

Every node is an active system component. Vertically adjacent nodes ('father' and 'son') may be related in different ways:

The son may be part of the father or just used by him. In the latter case, there may be several fathers for one son.

There are also different relations among 'brothers', i.e. sons of the same father. They may be active in parallel or sequentially or only one son is chosen for execution.
The approach of PCSL is restrictive in the way that every 'family' has to be of exactly one type:

The sons of each father form either a parallel group (in which case they are called ACTIVITIES) or a sequential group (then they are called STEPS) or an alternative group. In any case the father may be either a STEP or an ACTIVITY.

ACTIVITIES
parallel group

STEPS
sequential group

Alternative

Those three relations are restricted to tree-structure, i.e. no son may have more than one father. Though such a tree-structure has many advantages, it is not sufficient for the modeling of real systems because there are usually some parts that do not belong to one branch or the other but to several branches of the tree. (Error handling is a well-known example.) That is why one more relation called 'utilize' is defined in PCSL:

STEPS and ACTIVITIES

ACTIVITY

Thus, the rules of PCSL concerning the structure of active system components can be summarized as follows:

Every STACT (which stands for STEP or ACTIVITY) may be decomposed into a parallel or sequential or alternative group, or it may utilize an ACTIVITY. Also, a STACT must not be a member of more than one group. If it is utilized, it must not be in any group.
When this concept is applied to software systems in a top-down direction, in the first step the whole system is represented by an ACTIVITY. In process control, the global system will usually consist of coexisting parts which can be represented by sub-ACTIVITIES (subact-relation). Refinement by subact-relation may be repeated in several levels until the sequence of execution becomes more important. From then, step-, alternative-, and utilize-relations will be preferred for decomposition.

When a STACT is terminated by some external event (e.g. an interrupt), all dependents are terminated implicitly. So the terminate-relation must be used with care.

When all active system components are represented in this way, it does not make sense to say that a STACT is started, because its father and brothers determine precisely the time it is started, and it cannot be started at any other time. Termination, on the contrary, is well defined.

However, for the sake of simplicitely a start-relation is allowed in PCSL for a situation when a STACT has to be executed cyclically, triggered by some event or condition.

Without the start-relation, this STACT had to be decomposed into two STEPS, one of which is the waiting state that is terminated by the event.
2.3 A concept for passive system components

In all software-systems, two types of data can be distinguished:

- 'static' data which is once initialized and then read and rewritten in an undefined order. Reading and writing does not change the quantity of data.
- 'dynamic' data which moves through the system. Such data is produced and consumed by reading and writing, i.e. if there is only space for one item, every write-access must be followed by a read-access and vice versa.

Non-computer-examples are

- information on a blackboard (static)
- products in a buffer area between a producing and a consuming process (dynamic).

Examples from process-control-software are

- a table containing scaling factors of some measuring instrument (static),
- values from the technical process, which are filtered, checked, stored, printed etc. (dynamic).

In PCSL, the terms VARIABLE and BUFFER are used. VARIABLES contain static data, BUFFERS contain dynamic data. STACTS can access VARIABLES by 'initialize', 'read', and 'write', BUFFERS by 'initialize', 'produce', and 'consume'.

A VARIABLE may consist of subparts which, again, are VARIABLES, or it may have a TYPE (like BUFFERS do).

TYPES, also, may be refined by other TYPES.

In process control systems, BUFFERS are used for many kinds of data:
- messages to and from system-components
- input and output
- queues, also event-queues
- all kinds of data to be processed.
VARIABLES represent scaling factors, state-variables, switches, date and time.

BUFFERS are the links between cooperating STACTS. In a well structured system, such STACTS are executing in parallel. Their synchronization is done implicitly by communication via BUFFERS. A STACT should never consume from and produce for the same BUFFER!

BUFFERS can differ in several ways. If a buffer is full, a producer trying to deliver an item can either be blocked or not (in which case the item is lost).

The same is true for a consumer accessing an empty buffer. This can be expressed by the properties EMPTY and FULL, whose values can be either BLOCK or SKIP.

The items in some BUFFERS may be delivered and/or removed by several rather than just one access, e.g. messages to be output both on a screen and on a printer. In such cases, an INPUT-FAN and/or an OUTPUT-FAN may be specified in addition to the type of data in the BUFFER.

If a BUFFER can contain more than one item (CAPACITY), property ORDER (FIFO, LIFO or RANDOM) can be specified.

STACTS may occupy RESOURCES, e.g. memory or tape units. This can be stated in PCSL. Also, a VARIABLE or a BUFFER may be connected to a RESOURCE by the device-relation.
2.4 A concept for time, synchronization, and interaction

Time, synchronization, and interaction cannot be treated separately, because synchronization is a special case of interaction, and both deal with time.

EVENTS may be used to describe influences from outside or inside the system which terminate STACTS. (This means implicitly that either its successor is started or the upper STACT terminates.) The use of a start-relation has been restricted to cyclic STACTS, since we found that if allowed in other applications it is hard to provide a clear meaning.

INTERRUPTS are special events frequently found in process control applications. Examples are sensors which evoke an interrupt when a certain state transition has happened in the technical process, also the attention-key on the terminal.

The real-time-clock is an important source of signals triggering STACTS of the computer system. In PCSL, object-type TIMER is used to describe delayed and/or cyclic events. If no delay is specified, the TIMER will send the first pulse at the very begin of its activity; if no cycle is given, only one pulse is generated.

The time a TIMER or a STACT is active may be defined with respect to other STACTS by the as-long-as-relation.

Explicit description of synchronization by means of semaphores (Dijkstra 1968 (a)) or even more primitive tools tends to cause many bugs which are extremely difficult to detect. So we spent some time on the question which concepts might be both safe and easy to use.

In application software, most synchronization problems are either
- mutual exclusion or
- producer/consumer - access to some buffer or
- reader/writer - access to some data.

We think a concept which follows those three schemes will be sufficient and certainly safe and easy to use, because the user can just state what he wants the system to do rather than how. So we associate a predefined scheme of synchronization with all VARIABLES (reader/writer-scheme as described by
Courtois, Heyman, Parnas 1971), BUFFERS (producer/consumer-scheme, see e.g. Dijkstra 1968 (b)), and RESOURCES (mutual exclusion, expressed by the occupy-relation).

INTERVALS are used to define delays, cycles, etc.

2.5 Other elements of PCSL

KEYWORDS, MEMOS, and ATTRIBUTES with ATTRIBUTE-VALUES are just copied from PSL, also the text DESCRIPTION.

For alternatives and logical relations, CONDITIONS have been introduced. Primitive conditions are the RANGE-objects (RANGE-INT, RANGE-CNT, ...) which are subranges of the ranges of primitive variables (INTEGER, COUNT, REAL, BINARY, STRING).

E.g., a temperature T1 of type REAL may vary from 10 to 300 degree. If some action is required when T1 rises above 200 degree, a RANGE-REAL T1-normal can be defined with 200 degree as its upper bound. A STEP normal-step may depend on T1-normal (while-relation). Whenever T1 is not within T1-normal, normal-step is terminated, allowing its successor action-step to start.

Primitive variables (INTEGER etc.) can be restricted either to a range or to a list of values (BINARY and STRING only to a list).

PCSL contains also some elements intended for simulation-purposes, e.g. property MAXIMUM-DEVIATION for INTERVALS.
3. Analysis and reports

PCSL is not designed to be just the input language of an analyzer; its primary objective is to supply users with a set of concepts that allows simple but correct statement of well structured systems (thus preventing the design of badly structured systems).

But, of course, many evaluations and reports are desirable; some are necessary simply because it is not possible (at least not yet) to state semantical restrictions in the META-definition (e.g. when a relation is required to be tree-structured).

Simulation is a highly desirable but also very complicated way of analysis.

Thus, three types of analysis can be distinguished:
- checks for semantic correctness
- checks for completeness and consistency
- simulation.

3.1 Checks for semantic correctness

Since only very few semantical restrictions can be expressed in the META-language (namely in the CONNECTIVITY-statement), checks are necessary to ensure the proper use of PCSL.

The semantical rules are concerned with
- hierarchical structures of STACTS, VARIABLES and TYPES (e.g. any STEP must be contained within one and only one STACT).
- logical and numerical consistency (e.g. the sum of the probabilities of several alternatives must not be higher than 1.

A VARIABLE which is KEPT from one execution of the program until the next one (property CREATION) cannot consist of VARIABLES which are LOST every time).
- proper use of interdependent relations

(a STEP can be connected to its 'father' either by 'initstep' or by 'step' or by 'waitstep', but only one is allowed.

If it is connected by 'step', it must be mentioned in a next-relation of at least one other STEP.)

All rules of this kind have been stated in /Ludewig 1978/ in a formal notation. Cp. the last example above:

\[(x,s_0) \in \text{initstep} \land (x,s_n) \in (\text{step} \cup \text{waitstep})\]

\[s_1,...,s_k: \forall j=1,...,k: (s_{j-1}, s_j) \in (\text{next} \cup \text{alternative}) \land s_k=s_n;\]

which is more precise (and more restrictive) then the informal statement above.

'initstep' is the set of ordered pairs the second of which is the initial STEP of the first (the order is defined elsewhere).

3.2 Checks for completeness and consistency

As far as possible completeness and consistency should be checked by automated tools. In the PSA-system, many reports serve this purpose, e.g. the data-process-interaction-report. In PCSL, we can check

- if there is only one root in the hierarchy of STACTS,
- if all VARIABLES are both written and read,
- if all BUFFERS are both produced for and consumed from,
- if all RESOURCES are occupied at least once, also if all other objects are connected in any way,
- if all BUFFERS with space for more than one item have a defined order (e.g. FIFO) etc.

If an object is PERIPHERAL to the system (e.g. a display which is read by a person not defined in the system) completeness is not required (e.g. within the system the display may be only written). Thus the property POSITION
(PERIPHERAL or INTERNAL) determines which checks are necessary. The same applies also to other properties.

Like the rules in 3.1, these rules are collected in /Ludewig 1978/.

3.3 Simulation

Since our ideas about simulation are still rather vague, only the outline of our approach is sketched in the following.

All relations describing actions of any type, e.g. execution of STACTS, reading and writing, etc., are defined as operations of a - virtual - PCSL-machine described in /Ludewig 1978/. In particular, this machine simulates all synchronization implicitly defined by the use of VARIABLES, BUFFERS, and RESOURCES (cp. 2.4) and the delays caused by program execution, transfers and waiting conditions. At decision points, control is passed either to the operator or to a random number generator which may be controlled by parameters (e.g. property PROBABILITY of CRITERIA).

As a result of this simulation, we expect indications of bottlenecks, of system parts which are critical with respect to execution time and others which are not, and also indications of structures that are not safe against deadlocks.

Simulation can also be used to compare different designs.

Many questions are not yet answered:

1. How shall the technical process be simulated?  
   (cp. /Baumann 1978/ for one possible way)

2. Should the simulation be restricted to actions as sketched above or should it also include data handling (i.e. deal with the content of data rather than with its mere existence)?

3. Should the simulator be implemented from scratch or on top of a system based on Petri-nets that was developed in our institute (Schumacher 1978)?
4. **PUES - a sample application of PCSL**

A good example should exhibit all characteristics in a real-life-application. It should nevertheless be small and understandable for everybody. In this sense, we did not find a good example to date. The following one is a compromise: PUES (Prozeß-Überwachungssystem = process monitoring system /Borrmann 1978/) is being used as a test-object for PCSL in a master thesis (/Vinzentz 1979/). PUES will be implemented at our institute for application in the nuclear research center Karlsruhe.
General remarks:

Since this is a real application, it does not exhibit all features of PCSL. Only ACTIVITY dialog-input-processing has been extended to show the use of STEPS. (The example did not contain any STEPS before because they usually do not arise before a lower level of abstraction is reached.)

We do not pretend that this is a correct description of a correct design because the design is still in progress, and there is no other formal specification as a reference.

Though a similar description in PCSL.2 was successfully supplied to the Generalized Analyzer, there may be syntactical errors in this input because we do not yet hold the META-tables of PCSL.3, the improved language.

The input is structured for the reader's convenience. The most important objects are listed below:

- process monitoring-system
- technical process
- operator
- measured-value-processing
- measured-value-logging
- data-preparation
- recording
- life-display
- error-handling
- alarm-handling
- dialogue
- dialogue-input-processing
- dialogue-output-processing
- output-periphery
- input-buffer
- raw-data-buffer
- result-buffer
- status-information-buffer
- output-buffer
- control-information-block
- order-table
- parameter-table
- hardware-table
DEF ACTIVITY process-monitoring-system;

DESCRIPTION;
This system monitors the state of the technical process by cyclic acquisition of measured values and state-information. The values are transformed to physical units and printed at the operator-terminal, also recorded on external memory.

SUBACTS ARE technical-process, operator, measured-value-processing, alarm-handling, dialogue, output-periphery;

DEF ACTIVITY technical-process;
POSITION PERIPHERAL;
PRODUCES FOR input-buffer;
EXECUTING REPEATEDLY;

DEF ACTIVITY operator;
EXECUTING REPEATEDLY;
CONSUMES FROM teletype-printer;
PRODUCES FOR teletype-keyboard;
DEF ACTIVITY measured-value-processing;

SUBACTS ARE
  start-data-logging,
  data-preparation,
  recording,
  life-display;

DEF ACTIVITY start-data-logging;

CARDINALITY number-of-orders;
EXECUTING REPEATEDLY;
STARTED BY trigger;
WRITES index-of-order;
UTILIZES measured-value-logging;
OCCUPIES measured-value-logging;

DESCRIPTION;
There is one trigger and one start-data-logging for every order, but all are executed by measured-value-logging. In the implementation, there will probably be only one start-data-logging to handle all orders.

DEF TIMER trigger;

CARDINALITY number-of-orders;
LOCAL TO measured-value-processing;
CYCLE cycle-of-order;
DELAY delay-of-order;

DEF VARIABLE index-of-order;
LOCAL TO measured-value-processing;

DESCRIPTION;
index-of-order must be one of the order-identifications which might be of type COUNT or STRING.
DEF ACTIVITY measured-value-logging;

DESCRIPTION;
Program that reads in process data and stores them ordered in the raw-data-buffer.

The process data consists of the values measured and converted to digital representation, and the status information which is a true/false-statement about the state of the process or plant. The data is supplied by process-interfaces.

The program will get its particular task from the data-aquisition-order, the index of which was passed as a parameter. In case of errors, the questionable data are marked to avoid misinterpretation by the receiver.

;  
READS data-aquisition-order, index-of-order, time, hardware-table, priority;
PROduces for raw-data-buffer;
ALTERNATIVE error-proc-in-mvl;

DEF ACTIVITY error-proc-in-mvl;
DEPENDING ON error-in-mvl;
WRITES control-information-block;
UTILIZES error-handling;

---

DEF ACTIVITY data-preparation;

DESCRIPTION;
Program that checks and converts the data read in. It is controlled by the presence of data in the raw-data-buffer, which are copied to a work-area and finally as physical values put out to the result-buffer.

;  
EXECUTING REPEATEDLY;
READS evaluation-parameters;
CONSUMES FROM raw-data-buffer;
OCCUPIES work-area;
DEF ACTIVITY
DESCRIPTION;

DEF RESOURCE
DESCRIPTION;

LOCAL TO

###

## **work-area;**

###

2.3

DEF ACTIVITY
DESCRIPTION;

EXECUTING
CONSUMENT FROM
CONSUMENT FROM
PRODUCES FOR
ALTERNATIVE

DEF ACTIVITY
DESCRIPTION;

DEPENDING ON
WRITES
UTILIZES
DEF BUFFER
DESCRIPTION;
The external memory is treated as a buffer rather than a resource, because it is only occupied but not released by this system.

LOCAL TO recording;
POSITION PERIPHERAL;

DEF ACTIVITY life-display;
DESCRIPTION;
Output of a subset of all the values in the result-buffer.
'Characteristic values' are predetermined results which are plotted or printed or displayed.

EXECUTING REPEATEDLY;
READS control-information-block;
CONSUMES FROM result-buffer;
PRODUCES FOR output-buffer;
ALTERNATIVE error-proc-in-l-display;

DEF ACTIVITY error-proc-in-l-display;
DEPENDING ON error-in-l-display;
WRITES control-information-block;
UTILIZES error-handling;
DEF ACTIVITY error-handling;

DESCRIPTION;

Sends error-messages to the control-desk and handles those error-situations that cannot be resolved by the programs internally. May be called by any other system component. All information about the error is passed via parameters in the control-information-block. This routine may influence the data-aquisition by changes in the control-information-block.

An error is a situation in which the acquisition and transmission of data does not accord to the schedule. This may be caused by one of the following two reasons:
1. breakdown of hardware-components,
2. wrong parameters supplied from the user that cannot be identified by the checks for plausibility of the dialogue-system.

In any case, the error must be handled by an operator's action as soon as possible. An error-message is generated.

READS control-information-block;

PRODUCES FOR output-buffer;

DEF ACTIVITY alarm-handling;

DESCRIPTION;

User-oriented routines that handles alarms from the technical process. The steps to be taken may be either independent (including I/O), or may influence the actions of the data-aquisition-system via the content of the control-information-block.

EXECUTING REPEATEDLY;

STARTED BY alarm;

WRITES control-information-block;

PRODUCES FOR output-buffer;

DEF INTERRUPT alarm;

LOCAL TO alarm-handling;
DEF ACTIVITY dialogue;

DESCRIPTION;
Module for communication with the operator. Both control- and evaluation-parameters may be modified. The actual state of the process and the system may be inquired. The dialogue is triggered by input from the keyboard.

Users define and change orders for data-aquisition in a dialogue, prompted by the system.

SUBACTS ARE dialogue-input-processing, dialogue-output-processing;

DEF RESOURCE teletype;
LOCAL TO dialogue;

DEF ACTIVITY dialogue-input-processing;
EXECUTING REPEATEDLY;
STARTED BY attention-key;
INITSTEP read-input;
SUBSTEPS input-check;
DEF INTERRUPT LOCAL TO attention-key;
alarm-handling;

DEF STEP read-input;
OCCUPIES teletype;
CONSUMES FROM teletype-keyboard;
WRITES line-buffer;
NEXT STEP input-check;

DEF STEP input-check;
READS line-buffer;
ALTERNATIVES input-error,
data-aquisition-order-proc,
evaluation-parameters-proc,
system-state-inquiry-proc;
DEF STEP
NEXT STEP input-error;
read-input;

DEF STEP
WRITES data-aquisition-order-proc;
data-aquisition-order;

DEF STEP
WRITES evaluation-parameters-proc;
evaluation-parameters;

DEF STEP
WRITES system-state-inquiry-proc;
system-state-inquiry;

DEF BUFFER
teletype-keyboard;

DESCRIPTION;
the keyboard is used to enter
* data-aquisition-orders
* evaluation-parameters
* system-state-inquiries

CONSUMED BY read-input;
PRODUCED BY operator;
CAPACITY 1 OF input-line;

DEF TYPE
input-line;
CONSTS OF 80 OF input-character;

DEF ACTIVITY
dialogue-output-processing;
EXECUTING REPEATEDLY;
CONSUMES FROM status-information-buffer;
OCCUPIES teletype;
PRODUCES FOR teletype-printer;
DEF BUFFER
        teletype-printer;
DESCRIPTION;
The teletype-printer is used to transmit
* status-informations
* error-messages
;
CONSUMED BY operator;
CAPACITY 1 OF output-line;

DEF TYPE
        output-line;
CONSISTS OF 120 OF output-character;

DEF ACTIVITY
        output-periphery;
SUBACTS ARE display-drive,
           printer-drive;

DEF ACTIVITY
        display-drive;
EXECUTING REPEATEDLY;
CONSUMES FROM output-buffer;
PRODUCES FOR display;
DEF BUFFER
        display;
POSITION PERIPHERAL;
LOCAL TO display-drive;

DEF ACTIVITY
        printer-drive;
EXECUTING REPEATEDLY;
CONSUMES FROM output-buffer;
PRODUCES FOR printer;
DEF BUFFER
        printer;
POSITION PERIPHERAL;
LOCAL TO printer-drive;
### DEFINITION OF MAIN-DATA-AREAS

#### 6.1
**DEF BUFFER**
- **input-buffer;**
  - **PRODUCED BY** technical-process;
  - **CONSUMED BY** measured-value-logging;
  - **CAPACITY** size-input-data-buffer OF input-data;
  - **DESCRIPTION:**
    - The input-data may be either
      - measured values
      - status-information

#### 6.2
**DEF BUFFER**
- **raw-data-buffer;**
  - **LOCAL TO** measured-value-processing;
  - **PRODUCED BY** measured-value-logging;
  - **CONSUMED BY** data-preparation;
  - **CAPACITY** size-raw-data-buffer OF raw-data;

**DEF TYPE**
- **raw-data;**
  - **CONSISTS OF** time-record-2, order-identification-2, raw-data-2;

#### 6.3
**DEF BUFFER**
- **result-buffer;**
  - **DESCRIPTION:**
    - The output-fan indicates that both recording and life-display get every item in the buffer;
    - **OUTPUT-FAN** 2;
    - **LOCAL TO** measured-value-processing;
    - **PRODUCED BY** data-preparation;
    - **CONSUMED BY** recording, life-display;
    - **CAPACITY** size-result-buffer OF result;
DEF TYPE
result;
CONSISTS OF
time-record-1,
order-identification-1,
result-1;

DESCRIPTION;
The results are in physical units.
;
---

DEF BUFFER
status-information-buffer;
PRODUCED BY
data-preparation;
CONSUMED BY
dialogue;
CAPACITY
size-status-information-buffer
OF
status-information;

DESCRIPTION;
status-information-buffer contains status-information.
;
---

DEF BUFFER
output-buffer;
OUTPUT-FAN
2;
PRODUCED BY
life-display,
BY
alarm-handling,
BY
error-handling,
BY
recording;
CONSUMED BY
display-drive;
CAPACITY
size-output-buffer OF output-data;

DESCRIPTION;
output-buffer may contain
* process-state-descriptions
* sample-values
* event-messages
;
DEF VARIABLE control-information-block;

DESCRIPTION;
Central information that specifies the monitoring and processing to be done. In particular, the control-information-block contains all parameters for control and evaluation that may be subject to modifications. The content of the control-information-block may be modified by alarm-handling, dialogue, and error-handling.

SUBPARTS ARE order-tabel,
parameter-tabel,
hardware-tabel;

DEF VARIABLE order-tabel;
SUBPARTS data-aquisition-order;

DEF VARIABLE data-aquisition-order;
CARDINALITY number-of-orders;
SUBPARTS ARE list-of-measuring-points,
order-identification,
priority;

DEF VARIABLE parameter-tabel;
SUBPARTS ARE tabel-of-standard-values,
previous-values,
scaling-factor;

DEF VARIABLE hardware-tabel;
SUBPARTS ARE hardware-addresses,
dma-numbers,
bit-numbers,
distorsion-factor;
5. **PCSL versus PSL, DSL and EPOS**

In this chapter, PCSL is compared with PSL (its 'father'), (its 'older brother') and EPOS, a system which is being developed at the Stuttgart-University (its 'colleague').

PSL, DSL and PCSL are very similar as far as style and structure of the languages are concerned, because they all have to be defined by the META-system which implies many restrictions (structure of sections and statements, relations, etc.). Such similarities are not discussed here.

5.1 **PSL**

PCSL differs from PSL:

- in the object-types for data: Instead of a large number of object types related to data base terminology, only VARIABLES and BUFFERS are available; (COUNT, INTEGER, REAL, BINARY, STRING are just primitive VARIABLES.)
- in its means to handle time and synchronization which are very poor in PSL;
- in the distinction of two object types instead of PROCESS. This allows to describe parallelism easily;
- in the presence of logical relations (while, while not etc., object-types CRITERION, RANGE-INT etc.); In PSL, logical relations are only described by informal texts.

In general, PCSL is more restricted to a special area than PSL. As a benefit from this restriction, a more precise meaning can be attached to all elements of the language.
5.2 DSL

Similar to PCSL, DSL /Bodart, Pigneur 1979 a, b/ is based on PSL. (Neither PCSL nor DSL is a superset of PSL!) Also, DSL stresses the dynamic system aspects in order to allow more dynamic analysis and simulation which is impossible with PSL as it is. PCSL and DSL differ slightly in their means and goals:

PCSL is less analysis-oriented than DSL, we tried to develop a language which is useful even if used without any tools for analysis, just by the concepts which (hopefully) contribute to reliability because they are easy to understand and restrict their users to reliable constructions.

The focal point of DSL seems to be on analysis and simulation. EVENTS and RESOURCES are most important, the use of RESOURCES can be described in more detail than in PCSL, but there is no bias for constructs that hide details of implementation within complex operations like 'produce' and 'consume' in PCSL do.

DSL consists of two levels, the one for the GA, and the so-called nested language for simulation. A similar approach will be necessary for PCSL, though we tried to have most information for simulation on the primary level.

5.3 EPOS

EPOS /Biewald et al. 1979/ is a system for specification and design of process control software. It is directed towards PEARL, a process control programming language supported by our government.

EPOS consists of a language (with different parts for requirements and design), an analysis-system which encompasses a simulator, a report generator, and a user interface for controlling the whole system.

EPOS is being implemented in PEARL on a minicomputer AEG 80-20. It is planned to be used in life-size-applications in 1980.
EPOS-R, the subsystem for requirements, acts as a text- and graphic editing facility with little relational and logical capabilities.

Such capabilities are found in EPOS-S. (This name is inconsistently used for both the language and the system to process it.) Object-types are ACTION, DATA, INTERFACE, EVENT, CONDITION.

There are relations for data-flow (INPUT, OUTPUT), control (TRIGGERED), processing (PROCESSED) and decomposition.

 ACTIONS are decomposed in a procedural way (sequence, if-then-else, etc.).

Apparently, relations can be specified only in one direction (not from either end).

EPOS-A (analysis) and EPOS-D (documentation) correspond to the analysis-and report-generation-capabilities of PSA. In the paper cited above, seven types of analysis are listed, including syntax check, consistency checks and simulation. Much plotted output is planned.

Compared to PCSL, EPOS differs in many ways. Its major advantage is its installation on a minicomputer. Also, the distinction (and tracing) between requirements and design seems to be a good feature. Finally, the hierarchically structured informal information (e.g. DESCRIPTION = PURPOSE + DATA + FULFILS + TEST + PERFORMANCE + NOTE) and the closed syntactical constructs (e.g. DESCRIPTION ... DESCRIPTIONEND) should be noted.

On the other hand, the means of PCSL for data flow and resource allocation are more elegant because they encompass all synchronization needed. The PSL-like languages are more flexible because they allow for stepwise accumulation of the problem statement, 'complementary relations', and implicit declaration.
6. **Current state of PCSL**

6.1 **The language**

PCSL was defined in August 1978 /Ludewig, Streng - Oct. 1978/; recently (July 1979) a minor revision was made to remove some mistakes from the first version and to change some keywords. (This paper is based on the revised version.)

6.2 **The Generalized Analyzer**

By July 1979, the most important parts of the Generalized Analyzer (GA) were operational at IDT /Berliner, Ludewig, Pozzi 1979/: 

- IP (Input PSL)
- DBSM (Data Base Summary)
- NS (Name Selection)
- FS (Formated Statement).

DP and RP (Delete and Replace) will follow soon. Thus, PCSL is available at IDT. A command-language interpreter for our batch-installation is being developed.

6.3 **Analysis-tools**

No steps towards design and implementation of modules for analysis and report-generation have been taken beyond the formal specifications given in /Ludewig 1978/.
7. References


Biewald, J., P. Göhner, R. Lauber, H. Schelling: EPOS — A specification and design techniques for computer controlled real-time automation systems. Proc. of the 4th Int. Conf. on Software Engineering, München, Sept. 1979, p. 245-250


Bodart, F., Y. Pigneur: Dynamic specification language and a simulation model analyzer for an information system. Inst. d'Informatique, FNDP, Namur, Sept. 1979


The primary reports listed below contain information of a provisional nature and were compiled primarily to promote the up-to-date internal exchange of information among the institutes and external partners cooperating with the Karlsruhe Nuclear Research Center. Any dissemination of these reports or its contents requires the consent of the Patents and Licenses Department of KfK.


Borrmann, H.: Konzept der Meßwerterfassung und -bereitstellung im Rahmen eines universellen Prozeßüberwachungssystems, August 1978

Ludewig, J.: Definition der Struktur und Semantik von PCSL, August 1978


Ludewig, J., W. Streng: META-Definition von PCSL, October 1978
This is a short but almost complete definition of PCSL; only the synonyms of keywords are missing. It was derived from META-definition by deleting all information relevant for database-organisation and error-handling but not for the language itself. All relevant parts have been compressed without loss of information.

The connectivity is stated in the headers corresponding to the sequence of components in the COMBINATION-statements. (e.g. MANY,ONE means the 2nd component of the relation may be connected to several 1st components, but for every 1st component, only one 2nd is allowed).

Lines starting with a number in parentheses describe statements for the relation defined before; the number indicates the component to which the statement applies (i.e. in whose section the statement may appear).

Parentheses indicate parts of statements that may be repeated n times, n = 0,1,2,... Parts in square brackets are optional.

Table of contents of the appendix:

1 Noise-words
2 Object-types
3 Texts
4 Properties
   .1 Properties for objects
   .2 Properties used only as types
5 Relations and statements
   .1 through .39 ordered by relation-names. All statements follow immediately after the relation.
Noise-words may be inserted in any statement without any meaning.

Object-types

- Activity, step,
- Buffer, variable, resource, type,
- Binary, count, integer, real, string,
- Event, interrupt, timer, interval,
- Condition,
- Range-bin, range-cnt, range-int, range-real, range-str,
- Keyword, memo, attribute, attribute-value,
- Function, process-variable;

Since object-type variable may be replaced by the simple-typed variables binary, count, integer, real, and string, in most applications, any-variable is used for the whole set.

Texts and the object-types they are allowed with

- Algorithm (function);
- Description (all);
- Objective (activity, step);
- Code (activity, step);
- Simulation (all);
### .1 PROPERTIES and their values (object-types in parentheses) ###

- **BIAS:** READER, WRITER (ANY-VARIABLE);
- **CREATION:** SUPPLIED, KEPT, LOST (ANY-VARIABLE);
- **EMPTY:** BLOCK, SKIP (BUFFER);
- **FULL:** BLOCK, SKIP (BUFFER);
- **FAN-IN:** INTEGER 1 THRU 1000000 (BUFFER);
- **FAN-OUT:** INTEGER 1 THRU 1000000 (BUFFER);
- **ORDER:** FIFO, LIFO, RANDOM, BY-PRIORITY (BUFFER);
- **POSITION:** INTERNAL, PERIPHERAL (ANY-VARIABLE, BUFFER, ACTIVITY,STEP);
- **EXECUTING:** ONCE, REPEATEDLY (ACTIVITY,STEP);
- **PRIORITY:** INTEGER 1 THRU 1000000 (ACTIVITY,STEP);
- **STATUS:** NEW, EXISTING (ACTIVITY,STEP);
- **MAXIMUM-DEVIATION:** NUMBER 0.0 THRU 1.0 (INTERVAL);
- **PROBABILITY:** NUMBER 0.0 THRU 1.0 (CONDITION);

### .2 PROPERTIES used only as types and their values ###

- **BINARY-RANGE:** STRING;
- **COUNT-RANGE:** INTEGER 0 THRU 1000000;
- **INTEGER-RANGE:** INTEGER;
- **REAL-RANGE:** NUMBER;
- **STRING-RANGE:** STRING;
- **TIME-UNIT-RANGE:** M-SEC, SEC, MIN, HOURS, DAYS, WEEKS, MONTHS, YEARS;
- **ARITH-COMPARATOR:** EQ, NE, GE, LE, GT, LT;
- **LOGIC-COMPARATOR:** EQ, NE;
App. 5.1 ### alternative-relation (ONE, MANY) ###

COMBINATION alternative-father ACTIVITY
WITH alternative-son ACTIVITY;
COMBINATION alternative-father STEP
WITH alternative-son STEP;

(1) ALTERNATIVES ARE alternative-son (, alternative-son );
(2) FACULTATIVE IN alternative-father;

App. 5.2 ### attribute-relation (MANY, MANY, ONE) ###

COMBINATION attribute-left-hand-part ALL BUT ATTRIBUTE
WITH attribute-part ATTRIBUTE
WITH attribute-value-part ATTRIBUTE-VALUE;

(1) ATTRIBUTE attribute-part attribute-value-part
(, attribute-part attribute-value-part );
(2) VALUES ARE attribute-value-part FOR attribute-left-hand-part
(, attribute-value-part FOR attribute-left-hand-part );

App. 5.3 ### capacity-relation (MANY, MANY, ONE) ###

COMBINATION capacity-buffer BUFFER
WITH capacity-unit TYPE
WITH capacity-count COUNT VALUE-FOR COUNT-RANGE;

(1) CAPACITY capacity-count OF capacity-unit;
(2) capacity-count ITEMS IN capacity-buffer
(, capacity-count ITEMS IN capacity-buffer );

App. 5.4 ### cardinality-relation (MANY, ONE) ###

COMBINATION card-subject-part ACTIVITY, ANY-VARIABLE, BUFFER,
TIMER, RESOURCE
WITH cardinality-part COUNT VALUE-FOR COUNT-RANGE;

(1) CARDINALITY IS cardinality-part;
(2) APPLIES TO card-subject-part (, card-subject-part );
### App. 5.5 ### consume-relation (MANY,MANY,ONE) ######

**COMBINATION consumer**  
**ACTIVITY, STEP**  
**WITH cons-buffer**  
**BUFFER**  
**WITH cons-count**  
**COUNT VALUE-FOR COUNT-RANGE;**

(1) **CONSUMES**  
([cons-count [OF BUFFER-UNIT]])  
**FROM** cons-buffer  
(,  
([cons-count [OF BUFFER-UNIT]])  
**FROM** cons-buffer);

(2) **CONSUMED**  
([cons-count [OF BUFFER-UNIT]])  
**BY** consumer  
(,  
([cons-count [OF BUFFER-UNIT]])  
**BY** consumer);

### App. 5.6 ### contain-relation (MANY,MANY,ONE) ######

**COMBINATION containing-part**  
**VARIABLE, TYPE**  
**WITH contained-part**  
**TYPE**  
**WITH con-repetition-part**  
**COUNT VALUE-FOR COUNT-RANGE;**

**COMBINATION containing-part**  
**INTERVAL**  
**WITH contained-part**  
**INTERVAL VALUE-FOR TIME-UNIT-RANGE**  
**WITH con-repetition-part**  
**COUNT VALUE-FOR COUNT-RANGE;**

(1) **CONSISTS OF**  
([con-repetition-part] contained-part  
(,  
[con-repetition-part] contained-part);

(2) **CONTAINED**  
([con-repetition-part [TIMES]])  
**IN** containing-part  
(,  
[con-repetition-part [TIMES]]  
**IN** containing-part);

### App. 5.7 ### control-relation (MANY,MANY,ONE) ######

**COMBINATION controled-var**  
**PROCESS-VARIABLE, ANY-VARIABLE**  
**WITH controller**  
**ANY-VARIABLE**  
**WITH control-fct**  
**FUNCTION;**

(1) **CONTROLED BY** controller [USING control-fct]  
(,  
controller [USING control-fct]);

(2) **CONTROLS**  
controled-var [USING control-fct]  
(,  
controled-var [USING control-fct]);

### App. 5.8 ### criterion-relation (MANY,ONE) ######

**COMBINATION alternative-part**  
**ACTIVITY, STEP**  
**WITH criterion-part**  
**CONDITION;**

(1) **DEPENDING ON** criterion-part;

(2) **APPLIES TO** alternative-part (, alternative-part);
### App. 5.9 ### cycle-relation (MANY,MANY,ONE) #######

COMBINATION cycle-timer TIMER
WITH cycle-time-unit INTERVAL VALUE-FOR TIME-UNIT-RANGE
WITH cycle-count COUNT VALUE-FOR COUNT-RANGE;

(1) CYCLE [ cycle-count OF ] cycle-time-unit;

### App. 5.10 ### delay-relation (MANY,MANY,ONE) #######

COMBINATION delayed-timer TIMER
WITH delay-time-unit INTERVAL VALUE-FOR TIME-UNIT-RANGE
WITH delay-count COUNT VALUE-FOR COUNT-RANGE;

(1) DELAY [ delay-count OF ] delay-time-unit;

### App. 5.11 ### device-relation (MANY,MANY) #######

COMBINATION device-content ANY-VARIABLE, BUFFER
WITH device RESOURCE;

(1) DEVICE device (, device );

(2) HOUSES device-content (, device-content );

### App. 5.12 ### implies-relation (MANY,MANY) #######

COMBINATION implies-step STEP, CONDITION
WITH implies-condition CONDITION,
RANGE-INT,
RANGE-CNT,
RANGE-REAL,
RANGE-BIN,
RANGE-STR;

(1) IMPLIES implies-condition (, implies-condition );

(2) IMPLIED BY implies-step (, implies-step );
App. 5.13 ### inhibit-relation (MANY,MANY) #######

COMBINATION inhibitor ACTIVITY, STEP
WITH inhibited-event EVENT, TIMER, INTERRUPT;
(1) INHIBITS inhibited-event (, inhibited-event );
(2) INHIBITED BY inhibitor (, inhibitor );

App. 5.14 ### initiate-relation (MANY,MANY) #######

COMBINATION initiater ACTIVITY, STEP
WITH data-initiated ANY-VARIABLE;
(1) INITIATES data-initiated (, data-initiated );
(2) INITIATED BY initiater (, initiater );

App. 5.15 ### initstep-relation (ONE,ONE) #######

COMBINATION init-father ACTIVITY, STEP
WITH init-son STEP;
(1) INITIAL-STEP IS init-son;
(2) FIRST-STEP OF init-father;

App. 5.16 ### intvl-def-relation (MANY,MANY,ONE) ####

COMBINATION defined-interval INTERVAL
WITH intvl-def-unit VALUE-FOR TIME-UNIT-RANGE
WITH intvl-def-count VALUE-FOR COUNT-RANGE;
(1) AVERAGE-LENGTH [ intvl-def-count OF ] intvl-def-unit;

App. 5.17 ### intvl-use-relation (MANY,ONE) #######

COMBINATION intvl-user ACTIVITY, STEP, BUFFER, EVENT, INTERRUPT
WITH used-interval INTERVAL;
(1) INTERVAL used-interval;
(2) APPLIES TO intvl-user (, intvl-user );
App. 5.18 ### keyword-relation (MANY, MANY) ###

COMBINATION keyed-part ALLBUT KEYWORD
WITH keyword-part KEYWORD;

(1) KEYWORD ARE keyword-part (, keyword-part );

(2) APPLIES TO keyed-part (, keyed-part );

App. 5.19 ### local-relation (MANY, ONE) ###

COMBINATION local-object ALLBUT ACTIVITY, STEP
WITH local-stact ACTIVITY, STEP;

(1) LOCAL TO local-stact;

(2) LIMITS local-object (, local-object );

App. 5.20 ### memo-relation (MANY, MANY) ###

COMBINATION memo-part MEMO
WITH memoed-part ALLBUT MEMO;

(2) SEE-MEMO memo-part (, memo-part );

(1) APPLIES TO memoed-part (, memoed-part );

App. 5.21 ### next-relation (MANY, ONE) ###

COMBINATION precessor STEP
WITH successor STEP;

(1) NEXT [ STEP ] IS successor;

(2) ENTERED FROM precessor (, precessor );
### App. 5.22 ### observe-relation (MANY, MANY, ONE) ###

COMBINATION observed-var PROCESS-VARIABLE, ANY-VARIABLE
   WITH observer ANY-VARIABLE
   WITH observe-fct FUNCTION;

(1) OBSERVED BY observer [ USING observe-fct ]
   (, observer [ USING observe-fct ] );

(2) OBSERVES observed-var [ USING observe-fct ]
   (, observed-var [ USING observe-fct ] );

### App. 5.23 ### occupy-relation (MANY, MANY, ONE) ###

COMBINATION occupier ACTIVITY, STEP, BUFFER, VARIABLE
   WITH occupied-resource RESOURCE, ACTIVITY
   WITH occupy-count COUNT VALUE-FOR COUNT-RANGE;

(1) OCCUPIES [ occupy-count OF ] occupied-resource
   (, [ occupy-count OF ] occupied-resource );

(2) OCCUPIED BY occupier [ occupy-count ITEMS ]
   (, occupier [ occupy-count ITEMS ] );

### App. 5.24 ### produce-relation (MANY, MANY, ONE) ###

COMBINATION producer ACTIVITY, STEP
   WITH prod-buffer BUFFER
   WITH prod-count COUNT VALUE-FOR COUNT-RANGE;

(1) PRODUCES [ prod-count [ OF BUFFER-UNIT ] ] FOR prod-buffer
   (, [ prod-count [ OF BUFFER-UNIT ] ] FOR prod-buffer );

(2) PRODUCED [ prod-count [ OF BUFFER-UNIT ] ] BY producer
   (, [ prod-count [ OF BUFFER-UNIT ] ] BY producer );

### App. 5.25 ### read-relation (MANY, MANY, ONE) ###

COMBINATION reader ACTIVITY, STEP
   WITH data-read ANY-VARIABLE
   WITH read-interval INTERVAL;

(1) READS data-read [ INTERVAL read-interval ]
   (, data-read [ INTERVAL read-interval ] );

(2) READ BY reader [ INTERVAL read-interval ]
   (, reader [ INTERVAL read-interval ] );
App. 5.26  ### start-relation (MANY,MANY)  

COMBINATION starting-part EVENT, TIMER, INTERRUPT
WITH started-part ACTIVITY, STEP;

(1) STARTS started-part (, started-part );

(2) STARTED BY starting-part (, starting-part );

App. 5.27  ### step-relation (ONE,MANY)  

COMBINATION step-father ACTIVITY, STEP
WITH step-son STEP;

(1) SUBSTEPS ARE step-son (, step-son );

(2) STEP OF step-father;

App. 5.28  ### subact-relation (ONE,MANY) 

COMBINATION activity-father ACTIVITY, STEP
WITH activity-son ACTIVITY;

(1) SUBACTS ARE activity-son (, activity-son );

(2) PARALLEL IN activity-father;

App. 5.29  ### subpart-relation (ONE,MANY) 

COMBINATION top-part VARIABLE
WITH sub-part ANY-VARIABLE;

(1) SUBPARTS ARE sub-part (, sub-part );

(2) PART OF top-part;
### App. 5.30 ### subrange-relation (MANY,MANY,ONE,MANY) ####

COMBINATION range-object INTEGER 
  WITH range-name RANGE-INT 
  WITH cmp-operator VALUE-FOR ARITH-COMPARATOR 
  WITH cmp-value INTEGER VALUE-FOR INTEGER-RANGE;

COMBINATION range-object COUNT 
  WITH range-name RANGE-CNT 
  WITH cmp-operator VALUE-FOR ARITH-COMPARATOR 
  WITH cmp-value COUNT VALUE-FOR COUNT-RANGE;

COMBINATION range-object REAL 
  WITH range-name RANGE-REAL 
  WITH cmp-operator VALUE-FOR ARITH-COMPARATOR 
  WITH cmp-value REAL VALUE-FOR REAL-RANGE;

COMBINATION range-object BINARY 
  WITH range-name RANGE-BIN 
  WITH cmp-operator VALUE-FOR LOGIC-COMPARATOR 
  WITH cmp-value BINARY VALUE-FOR BINARY-RANGE;

COMBINATION range-object STRING 
  WITH range-name RANGE-STR 
  WITH cmp-operator VALUE-FOR LOGIC-COMPARATOR 
  WITH cmp-value STRING VALUE-FOR STRING-RANGE;

(1) SUBRANGE range-name IF cmp-operator cmp-value 
    (, range-name IF cmp-operator cmp-value );

(2) IF range-object cmp-operator cmp-value;

### App. 5.31 ### terminate-relation (MANY,MANY) ######

COMBINATION terminating-part EVENT, TIMER, INTERRUPT 
  WITH terminated-part STEP, ACTIVITY;

(1) TERMINATES terminated-part (, terminated-part );

(2) TERMINATED BY terminating-part (, terminating-part );

### App. 5.32 ### true-while-not-relation (MANY,MANY) ###

COMBINATION twn-condition CONDITION 
  WITH twn-subrange CONDITION, RANGE-INT, RANGE-CNT, 
    RANGE-REAL, RANGE-BIN, RANGE-STR;

(1) TRUE-WHILE NOT twn-subrange ( NOR twn-subrange );

(2) NEG-RELATED TO twn-condition (, twn-condition );
### App. 5.33 ### true-while-relation (MANY, MAY) ###

COMBINATION true-condition CONDITION
WITH true-subrange CONDITION, RANGE-INT, RANGE-CNT,
RANGE-REAL, RANGE-BIN, RANGE-STR;

(1) TRUE-WHILE true-subrange (AND true-subrange);

(2) POS-RELATED TO true-condition (, true-condition);

### App. 5.34 ### utilize-relation (MANY, MANY) ###

COMBINATION utilizing-part ACTIVITY, STEP
WITH utilized-part ACTIVITY;

(2) UTILIZED BY utilizing-part (, utilizing-part);

(1) UTILIZES utilized-part (, utilized-part);

### App. 5.35 ### value-list-relation (MANY, MANY) ###

COMBINATION value-object INTEGER
WITH actual-value VALUE-FOR INTEGER-RANGE;

COMBINATION value-object COUNT
WITH actual-value VALUE-FOR COUNT-RANGE;

COMBINATION value-object REAL
WITH actual-value VALUE-FOR REAL-RANGE;

COMBINATION value-object BINARY
WITH actual-value VALUE-FOR BINARY-RANGE;

COMBINATION value-object STRING
WITH actual-value VALUE-FOR STRING-RANGE;

(1) VALUE-LIST IS actual-value (, actual-value);

### App. 5.36 ### value-range-relation (MANY, ONE, ONE) ###

COMBINATION valued-are-part INTEGER
WITH value1-part INTEGER VALUE-FOR INTEGER-RANGE
WITH value2-part INTEGER VALUE-FOR INTEGER-RANGE;

COMBINATION valued-are-part COUNT
WITH value1-part COUNT VALUE-FOR COUNT-RANGE
WITH value2-part COUNT VALUE-FOR COUNT-RANGE;

COMBINATION valued-are-part REAL
WITH value1-part REAL VALUE-FOR REAL-RANGE
WITH value2-part REAL VALUE-FOR REAL-RANGE;

(1) VALUE-RANGE value1-part THROUGH value2-part;
INTRODUCTION waitstep-relation (ONE, ONE)  

COMBINATION wait-father ACTIVITY  
WITH wait-son STEP;  

(1) WAITING-STEP IS wait-son;  
(2) WAITING IN wait-father;  

INTRODUCTION while-relation (MANY, MANY)  

COMBINATION while-step STEP, CONDITION  
WITH while-condition CONDITION, 
RANGE-INT, 
RANGE-CNT, 
RANGE-REAL, 
RANGE-BIN, 
RANGE-STR;  

(1) WHILE while-condition ( OR while-condition );  
(2) GUARANTEES while-step (, while-step );  

COMBINATION writer ACTIVITY, STEP  
WITH data-written ANY-VARIABLE  
WITH write-interval INTERVAL;  

(1) WRITES data-written [ INTERVAL write-interval ]  
(, data-written [ INTERVAL write-interval ] );  
(2) WRITTEN BY writer [ INTERVAL write-interval ]  
(, writer [ INTERVAL write-interval ] );