informatica 74 – Bled, 7.–12. oktobra 1974

A MODULAR AND STRUCTURED CONTROL LANGUAGE INTERPRETER

P.Ancilotti, M.Fusani, N.Lijtmaer, C.Thanos

Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
Pisa, Italy

The purpose of this paper is to describe the functional characteristics and the structure of the Control Language Interpreter (CLI) for the Pisa Software Laboratory (PSL) which has been implemented at IEI-CNR on a virtual machine generated by CP-67 on an IBM 360/67 computer. The CLI is itself a modular system and enables the user to converse with the PSL using the typewriter as a system console. This paper discusses a "guided" modularization as a mechanism to achieve the aims of the CLI, namely: Modularity, extensibility and reliability.

1. INTRODUCTION

The main purpose of the Pisa Software Laboratory -PSL- is to provide a special environment within which researchers, students and designers may experiment in building new software systems, interactively. Experiments may follow a structured design to obtain modular software systems [1]. In this context, a software system is a set of independent modules connected to fit the needs of the user. Each module is a functional unit and is programmed regardless of the others using the rules of structured programming. Thus, programs should be written using the three basic control structures: Concatenation, selection and repetition [2]. It should be noted that these restrictions need not be limited to higer level languages but it is also possible to induce a block structure and provide special macros for assembly language programs.

Communication between program modules is an area of increasing importance and interest in the design and implementation of reliable software systems. This communication is handled by what are known as interfaces, which in PSL are based on the port approach [3]. In this approach, modules must have input and output plugs, known as *ports*, dangling from them. An input port from a module can be connected to an output port of any other module through *mailboxes* (message buffers). Modules send messages along output ports and receive messages along input ports. Modules refer to ports with local names and as such do not need to know which other modules their ports are connected to.

One major advantage introduced by the port approach is *modularity*. In fact, Dennis considers modularity to be a property of computer systems and gives the following definition [4] :"A computer system has modularity if the linguistic level defined by the computer system meets these conditions:

a) Associated with the linguistic level is a class of objects that are the units of program representation. These objects are *program modules*.

b) The linguistic level must provide a means of *combining program modules* into larger program modules without requiring changes to any of the component modules. Further, the meaning of a program module must be independent of the context in which it is used."

The PSL meets the requirements described above:

a) PSL modules are the units of program representation and global variables references are not allowed in order to guarantee the context independence condition.

b) Furthermore, the PSL provides a mechanism to combine modules to obtain software systems.

In this picture, each module conceived as an entity is continuosly active, processing messages as long as they are available. Concurrency of operations is an inherent part of this notion of modularity. In fact, during execution each module becomes a cyclic sequential process and the software system may be viewed as a family of cooperating asynchronous processes. Moreover, each process runs in its own name space: Full protection between modules is achieved.

A software system design is done in two phases: The functional analysis phase and the module implementation phase. In the first one, the analysis of the specifications for the desired system determines the structure to be chosen and a complete list of the functions. Then, the functional specification for each module arises and the set of connections is defined. In the second phase, module programming takes place. Of course, this phase may be skipped if a library with the desired modules is available.

When these two phases are completed, modules are loaded and connected by means of the Control Language Interpreter (CLI), and then the software system may run.

2. GOALS AND FUNCTIONAL CHARACTERISTICS OF THE CONTROL LANGUAGE INTERPRETER

The CLI executes a set of commands which allow the user to create software systems supported by the PSL. By means of these commands three basic functions are performed:

1) to create system modules;

2) to connect modules;

3) to assign and/or modify the system parameters.

The goals to be meeted in the design of the CLI are the following:

a) *Extensibility and flexibility*
The CLI system must provide the facility to introduce additional Control Language Commands without programming again those CLI parts that are related with the basic functions provided. Side effects produced by the new parts must be avoided. Moreover, changes in the system must be easy to carry out.

b) *Reliability*
The CLI must be designed in such a way that its correctness should be easy to prove and errors could be isolated and detected. The CLI reliability is an essential requirement; in fact, since the CLI is a tool used by all the users to create new software systems, then the possible repercussions of errors increase. The user should be sure that any error detected is due only to bugs in his own system.

c) *Easiness to use*
The user should not be placed under heavy constraints: The commands should be free format and their sequences should not follow a precise order. Moreover, the CLI should be able to recognize the user mistakes and should permit the user to recover from them.

d) *Easiness of documentation*
The design and the specifications of the structure should be fully documented; in fact it should be easy to introduce changes or improvements.

3. GUIDELINES FOR CLI DESIGN

After the definition of the objectives given in the previous paragraph, the next question is how to build the CLI in order to achieve these goals.

To design the CLI two alternative solutions have been taken into account. The first one was to extend the PSL nucleus to execute the CLI functions; the other was *to build the CLI just like a software system supported by PSL*. The latter solution has been chosen: In fact, the goal of the extensibility would be impossible to meet by using the former, while the other goals are more difficult to achieve. Furthermore flexibility, extensibility, reliability and easiness of documentation are standard characteristics of a modular software system [5].

As far as reliability is concerned two remarks may be outlined:

a) Since each PSL module is written using the structured programming approach, the CLI may be amenable to proofs of correctness.

b) Since each PSL module becomes during execution a sequential process with its own address space, undesired interactions between the CLI modules are avoided. In fact, the only interactions between PSL modules are achieved by means of a message passing mechanism. Furthermore memory references to the dynamic nucleus area are forbidden and then full protection is obtained.

A drawback to build the CLI like a software system is that there is no way to initialize the nucleus data structures. However, to perform the basic functions, the CLI must allocate some specific data structures: User process descriptors, mailboxes, etc. In fact these objects will be handled by the PSL dynamic nucleus during the execution phase to control process activity [1]. To overcome this obstacle an interface between the CLI and the dynamic nucleus is interposed: This interface is called the *static nucleus* and must be present whenever a static interaction takes place. For example every interaction during debugging time must be also supported by the static nucleus.

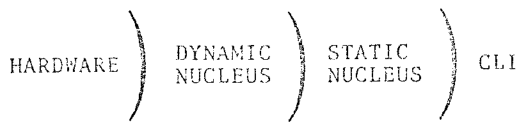

fig. 3.1

4. THE PSL STATIC NUCLEUS

The static nucleus must carry out the mapping function from the CLI module address space into the dynamic nucleus address space. This function is applied to the data structures created and referenced by the CLI modules. The mapping function is implemented by two procedures: *ALLOCATE* and *TEST*. While *ALLOCATE* is invoked by a module to initialize a specific data structure, *TEST* reflects its status. Two parameters must be provided. Since there are several types of data structures to be allocated, or tested, the first parameter specifies the *object type*. The second one defines the address of the object.

All the objects to be allocated by the static nucleus have a fixed format.

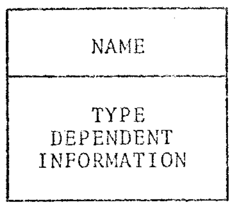

fig. 4.1

Two items compose the object format. The first item is the object unique name, while the second one consists of type dependent data. The length of the second field is variable and changes from type to type. Some object types are listed below:

Module Control Block	- *MCB* specifies the required information to initialize the named process descriptor.
Port Information Descriptor	- It denotes the module the named port belongs to, and the mailbox to which the port must be connected.
Mailbox	- It denotes the dimension of the named mailbox.
Priority	- It specifies the MCB to which the priority must be assigned.
Clock	- It indicates the time-slice length.

Other objects were also defined: The previous list is not complete and is given as an example of objects frequently referenced by the CLI.

5. THE CLI COMMANDS

To accomplish the three basic functions pointed out above, the CLI must interpret and execute a set of commands. The CLI commands are classified, according to their functions, into four subsets, namely:

1) Process creation commands.

2) Module connection commands.

3) Parameter assignment commands.

4) Control commands.

The first subset performs the functions related with the creation of processes. Only one command is provided:

CREATE PROCESS (<name>,<module>,<priority>)

All the three parameters must be supplied. Otherwise an error condition will be detected by the CLI and an error message will be sent to the user. The first parameter specifies the process unique name, instead the second one denotes the name of the program module to be retrieved from the library. This program module will become, during execution, the sequential process specified by the first parameter. Note that the same program module may be invoked in different *CREATE PROCESS* commands, but they always lead to distinct sequential processes. The third parameter specifies the priority to be assigned to the process.

Summarizing, the actions of the above command consist in :

a) MCB allocation.

b) Program module loading.

The commands of the second subset are concerned with the connections between modules. These commands are:

CONNECT (<MCB>,<port>,<mailbox>)

DISCONNECT (<MCB>,<port>,<mailbox>)

Since every connection implies a link between one port (belonging to a specific MCB) and one mailbox, three parameters must be provided. Note that while no more than one mailbox may be connected to any given port, it is allowed to connect more than one port to any mailbox.

The first action developped by the *CONNECT* command is a *TEST* procedure call with the mailbox name as a parameter. If the named mailbox was not previously declared, then *ALLOCATE* (<mailbox>,<address>) takes place and thus a new mailbox with the maximun length is created. In any case a port information descriptor is allocated and thus the connection is achieved.

The *DISCONNECT* command operates in the opposite way.

The parameter assignment commands are:

MAILBOX DIMENSION (<mailbox>,<size>)

CHANGE PRIORITY (<MCB>,<priority>)

QUANTUM CLOCK (<time-slice>)

The *MAILBOX DIMENSION* command allocates a mailbox with the declared size. The *CHANGE PRIORITY* command modifies the priority of the specified MCB. *QUANTUM CLOCK* allocates the time-slice length.

The last subset of commands allows the user to perform control functions, namely:

PERIPHERAL FILE (<device>,<module | system>)

This command denotes the device where the module or the system resides and must precede the *CREATE PROCESS* command. Disk device is assumed by default of the *PERIPHERAL FILE* command.

SYSTEM (<system name>)

This command denotes that the set of commands needed to create a software system resides in disk or in the device specified by the previous *PERIPHERAL FILE* command. The *SYSTEM* command allows the user to load a *library system* without typing all the commands needed to create this system.

DEFINE SYSTEM (<system name>)

This command is the dual of the previous one and allows to store a system in the library.

END

It denotes the end of the set of the building commands.

START

This command asks the nucleus to execute the new software system. It must follow the *END* command.

REMARK: The *CONNECT*, *DISCONNECT* and *CHANGE PRIORITY* commands may be given before the relative *CREATE PROCESS* command. In this case no error message is sent to the user, but the execution of those commands is delayed until the *CREATE PROCESS* arrives. This method allows the CLI to correct automatically certain mistakes on the sequences of commands typed by the user.

6. CRITERIA FOR THE CLI MODULARIZATION

Before giving the structure description of the CLI, some remarks on the criteria chosen for the CLI modularization are presented [6-7].

REMARK I: There are a number of design decisions which are questionable and likely to change. The first criterion, then, must be *changeability*: That is the system must be designed in order to allow the user to confine any change to only one module.

REMARK II: PSL gives a parallel environment and considers modules like sequential processes. The second criterion, then, is how to ensure a high order of *parallelism*: The interfaces between modules must be clearly defined and they must be chosen to reveal, as little as possible, its inner workings. Modules must not correspond to steps in the processing but to independent tasks.

REMARK III: A single function must be performed by a single module and implemented and tested just once, thus standardizing the way such function is performed. *Functional analysis* must be carried out trying to minimize the system complexity.

According to these criteria the CLI was designed as follows:

a) All logical or physical resources involved were specified.

b) The management of each resource was assigned to one module.

c) Some particular functions (Syntactical Analysis, Error Analysis, etc.) were singled out. Each of them is performed by one module.

The structure of the CLI and the implications of this choice in the CLI behaviour are described in the next paragraph.

7. CLI: STRUCTURE AND BEHAVIOUR

The set of modules and the CLI topological structure are shown in fig. 7.1. In the pic

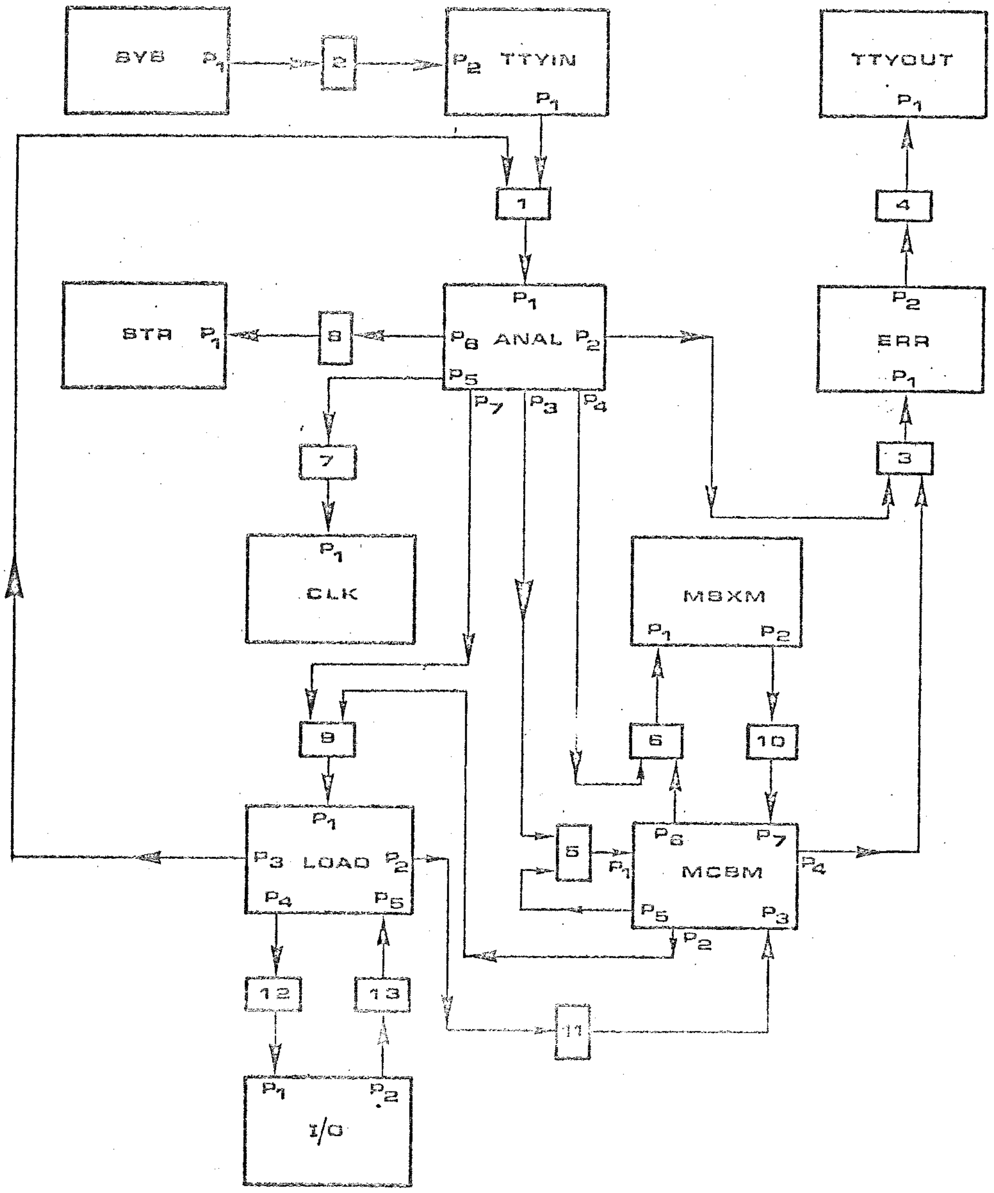
SYS
P1
2
P2
TTYIN
P1
TTYOUT
P1
1
4
P1
ANAL
P2
P6
P5
P7
P3
P4
STR
P1
8
P2
ERR
P1
3
7
P1
CLK
MSXM
P1
P2
9
6
10
P1
LOAD
P3
P2
P4
P5
5
P6
P7
MCSM
P1
P4
P5
P3
P2
12
13
11
P1
P2
I/O

fig. 7.1

ture large boxes represent modules, the small ones are mailboxes, arrows specify the directed connections and P's identify ports.

Four physical resources and two logical resources are involved in the CLI design. Both resources and relative modules are listed below:

Typewriter	*TTYIN*
Printer	*TTYOUT*
Disk	*I/O*
Clock	*CLK*
Mailboxes	*MBXM*
Module Control Blocks (with their own port descriptors)	*MCBM*

One specific module must be added for each other peripheral device.

While the modules listed above are resource handlers, the other modules are related with the specific functional characteristics of the CLI, namely:

Syntactical analysis	*ANAL*
Error management	*ERR*
Loading functions	*LOAD*
System start functions	*STR*
System end functions	*SYS*

An user types the commands on the typewriter. *TTYIN* sends these commands like messages through the port P_1 to mailbox 1, then *ANAL*, which is waiting for this type of message, analyzes them. If a syntactical error is detected, then, wrong messages are sent through port P_2 to mailbox 3. *ERR* will receive these messages generating the corresponding error messages to be printed by *TTYOUT*. Otherwise, if the command is valid, it is sent to different mailboxes according to its type.

The *MAILBOX DIMENSION* and the *QUANTUM CLOCK* commands will be received by *MBXM* and *CLK*, respectively. While the first module allocates mailboxes, the second one allocates the new clock.

MCBM is concerned with the MCB management. Then, every command, namely: *CREATE PROCESS*, *CONNECT*, *DISCONNECT* and *PRIORITY*, which initializes and/or modifies a particular MCB is received by *MCBM*.

For each *CREATE PROCESS*, *MCBM* sends through port P_2 to mailbox 9 a message that specifies the module to be loaded and waits on P_3 for the answer. If *MCBM* receives an OK, then the MCB is allocated, otherwise an error message is sent to *ERR*.

LOAD stores the module while *I/O* handles the chosen device. Note that both modules could be considered like only one function: Inherent concurrency in this function recommends to split it into two modules regarding the parallelism criterion and then increasing the efficency of the CLI system.

For every *CONNECT*, *DISCONNECT* and *PRIORITY* command *MCBM* calls the *TEST* procedure to know if the MCB specified by these commands was previously defined. If MCB does not exist the commands are delayed: They are sent through the output port P_5 to the same mailbox from which they were received. Otherwise *MCBM* allocates the specific objects required by those commands.

To perform *CONNECT* and *DISCONNECT* commands *MCBM* sends also a message to *MBXM* to test if the named mailbox was yet defined and waits for an answer before allocating the port information descriptor.

END command allows *MCBM* to check if mailbox 5 is empty, otherwise sends a message to *ERR*. This condition arises when there are delayed commands trying to modify inexistent MCB's.

A more detailed discussion of the CLI system is contained in [8].

8. CONCLUSIONS

This paper has described a design methodology for the development of a Control Language Interpreter, that runs under PSL. The basic objectives were reliability, flexibility, extensibility, easiness to use and to document. These objectives were achieved by a "guided" modularization based on the three criteria: Changeability, parallelism and functional analysis.

9. REFERENCES

[1] Ancilotti,P.; Cavina,R.; Fusani,M.; Gramaglia,F.; Lijtmaer,N.; Martinelli,E.; Thanos,C. "Designing a software laboratory". *Proceedings of the 8th Yugoslav International Symposium*, Bled 1973.

[2] Dijkstra,E.W. "Notes on structured programming". *Structured Programming*. Academic Press 1972.

[3] Corwin,W.; Wulf,W. "SL230 - Software Laboratory intermediate report". Carnegie-Mellon University, May 1972.

[4] Dennis,J.B. "Modularity". *Advanced Course on Software Engineering*. Bauer, editor, Springer-Verlag, 1973.

[5] Parnas,D.L. "Information Distribution Aspects of Design Methodology". *IFIP, Proceedings - Ljubljana, Yugoslavia 71*.

[6] Parnas,D.L. "On the Criteria to be used in Decomposing Systems into Modules". *Comm. ACM 15, Dec. 1972.*

[7] Liskov,B.H. "A Design Methodology for Reliable Software Systems". *Proc. of the Fall Joint Computer Conference 1972 AFIPS Vol. 41, part I.*

[8] Ancilotti,P.; Fusani,M.; Lijtmaer,N.; Thanos,C. "The PSL Control Language Interpreter". *IEI - CNR Internal Report (to be printed).*