E73-03

SEAS

SPRING TECHNICAL MEETING 1973
on
INTERACTIVE COMPUTING

April 9-13, 1973

Teatro Novelli
(Conference Theatre)
Rimini, Italy

Organized by: SEAS Executive Board
Local Organizer: SEAS Committees A.Chiarini
Meeting Co-ordinator: Hans Jørgen Helms

Psl: An interactive system for producing well structured programs

P. Ancilotti , R. Cavina, M. Fusani, F. Gramaglia, N. Lijtmaer, E. Martinelli, C. Thanos

Istituto di Elaborazione dell'Informazione
Consiglio Nazionale delle Ricerche

Psl: An interactive system for producing well structured programs

1. Introduction

The purpose of this paper is to describe the Pisa Software Laboratory, PSL. The PSL enables its users to have a special environment in building, in an interactive way, new software systems. Experiments may be done following a structured design to obtain modular systems.

The PSL has been implemented at IEI-CNR, on a virtual machine generated by CP-67 on a IBM 360-67 computer.

A software system is a set of independent modules connected in order 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.

Modules interfacing and communication is handled by PSL using the port approach /1/.

One major advantage introduced by the port approach is modularity. In fact, the PSL meets Dennis requirements: PSL modules are the units of program representation and global variables references are not allowed, in order to guarantee the context independence condition. Furthermore, PSL provides a mechanism to combine modules without requiring changes to any of the component modules /2/. In this picture, each module, conceived as an entity, is continuously 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.

Furthermore, PSL provides a protection mechanism in order to ensure the reliability of the system even if the correctness of any module is not guaranteed.

PSL has proved to be an useful tool in the construction of reliable software systems: Among others, it has been used to design a control language interpreter. In this particular experiment, the project goal was to produce not only a reliable software system, but a piece of readable software, relatively easy to modify and maintain /3/.

2. Software Systems

The main goal of the Pisa Software Laboratory is to provide a special environment within which researchers, designers and students have the possibility to experiment in order to build new software systems.

We define a software system as a set of independently programmed and interconnected modules, created to fit the needs of a particular user.

Following Parnas /4,5/, a module is a functional unit while the connections between modules are the assumptions which the modules make about each other.

Each module plays a specific role in the software system: then, during the execution, it must exchange information with the other modules.

In order to reach this goal the modules must be connected by means of communication paths. Each module has at his disposal several ports through which it may exchange messages with other modules. As regards the message flow, ports are unidirectional; Then there are input ports and output ports.

We can establish an analogy between ports in the software system and input/output wires in hardware modules.

The connection between two modules is made by linking one output port of a module whit an input port of the other module by means of a mailbox. A mailbox is a message buffer. The connection,port-mailbox-port, constitutes the communication path between two modules.

The modules are not aware of which other module they receive messages from nor send messages to. The only thing a module knows about a communication path, that connects it to another module, is the name of its port. Ports are local names to the modules.

The PSL treats a module, during execution, as a sequential process; Then a software system becomes a collection of parallel asyncaronous processes (a family of cooperating processes). These processes communicate, as we previously pointed out, by means of communications paths, which are standardized and tighly controlled by the PSL.

Allowing processes to communicate only through ports has many advantages:

- Interprocess communication is quite fast because little checking or searching is required for each message during execution.

- Detecting deadlocks should be simpler

- A protection mechanism can be easily implemented because information flows in a controlled way.

The main objective of the PSL is to provide an environment in which software systems can be easily built. The main property of these software systems is modularity. This property is easily obtained in the PSL because all the modules are not aware of which other module they receive messages from nor send messages to. In such a way, any module of the software system supported by the PSL can be changed, replaced or modified in any way without changing the other modules.

3. Nucleus

The PSL nucleus is a software extension of the hardware machine, in our case of the IBM 360/65. The nucleus is the run-time support for any software system programmed on the PSL and it provides the usual facilities of an operating system to allocate logical and physical resources.

More precisely, the goals of the nucleus are:

a) The management of processes; That is the creation of a virtual processor for any process of the system. In other words the nucleus must provide the multiprograming facilities of a general purpose operating system.

b) The interaction among processes; That is the control of processes when they must interact either in the case of competition for common resources or in the case of cooperation when they must exchange information.

c) The creation of a set of functions needed to allocate the system resources. More particularly: the functions used by processes to access the message passing mechanism, to require or release memory blocks and to use input/output devices.

The PSL nucleus is organized as a set of hierarchical levels. Each level modifies the underlying machine in order to create a new abstract machine more suitable from the point of view of modularity, reliability and concurrency. Processes running on the top level, see an abstract machine with a private processor for any process and where any abstract processor has facilities to exchange messages with other processors, to require or release memory blocks and to reliably handle I/O devices.

The first level is the short term scheduler. The goal of this component is the process management. For this purpose, active processes are maintained in a ready queue. This queue is priority ordered. When the running process is blocked the scheduler extracts from the ready queue the process with the highest priority. Then the execution of

this process is started. Processes with the same priority are handled in a FIFO order.

In order to simulate a higher degree of parallelism and to avoid the monopolising of processor time from CPU-bound processes, a time slice is assigned to each process. When the time slice ends the process is preempted and reinserted in the ready queue in a round-robin fashion.

At this level the protection mechanism of the IBM/360 is used to ensure the complete independence of processes. Each process has its name space and it can reference only its local variables. In such a way the PSL programming system allows a complete modularity since there are not global variables. Furthermore the system reliability is enhanced since errors are confined within processes.

At the second level the Dijkstra's primitives WAIT and SIGNAL on semaphores are implemented. these operations represent a very general synchronization mechanism that can be used to solve any synchronization problem both in the case of process competition for the use of common resources and in the case of process cooperation.

With the second level each abstract processor acquires the capability to wait for a particular event or to signal a particular event.

However it can be proved that wait and signal operations are very primitive and powerful and so they can be very dangerous if they are used in an undisciplined way.

For this reason wait and signal operations are not directly usable on the PSL. That is obtained with a third level whose goal is to mask these operations to provide new functions, implemented in terms of semaphore primitives, needed to exchange messages and to allocate the physical resources.

3.1 Message passing functions

The main component of the message passing mechanism of the PSL is the mailbox. This component is a nucleus data structure needed to contain, for a pair producer-consumer, messages sent by the producer but not yet received by the consumer. In other words a mailbox is a message buffer where messages are handled in a FIFO order.

Some semaphores are associated to a mailbox in order to allow process synchronization during the message passing activity. In particular there is a semaphore Mutex used to guarantee the mutual exclusion in the mailbox operations between senders and receivers. Furthermore there are two semaphores that guarantee the correct synchronization between senders and receivers. In particular they guarantee that messages can not be received before they are sent, or,

in other words, no message can be received from an empty mailbox.

Viceversa, if the mailbox is finite, that is if it contains almost n messages, the difference between the number of sent messages and the number of received messages can not be greater of n, or, in other words, no message can be sent to a full mailbox.

The use of the three semaphores associated to a mailbox has beeen proved correct from the point of view of the mutual exclusion and of the duplication or loss of messages. Furthermore it can be proved that no deadlock is possible through the use of a single mailbox.

Processes reference a mailbox through ports. Ports are local names to processes that can be bound to mailboxes during the creation of the software system.

A port can be conceived as a pointer that is initiated with the value nil. After the binding phase, during which ports are connected to mailboxes, any port contains the name (address) of the mailbox to which it is connected. The only way a process has to reach a mailbox is through a port.

We have conceived the connection between two processes as a unidirectional link. For this purpose we distinguish between input ports and output ports.

If P and Q are two processes where p is an output port of P, q is an input port of Q and M is mailbox, the connection p-M-q is the link that connects the producer P to the consumer 2. For each process the port, after the connection, is the local name of the link.

In order to allow processes to exchange messages through a link two functions are implemented: SEND and RECEIVE. The SEND function has two parameters: SEND (mes,por), where mes represents the message to be sent and por the name of the local port through which the message must be sent. After the execution of this function, the message mes will be in the mailbox connected to the port por of the executing process. The first parameter is a pointer containing the address of the memory block in which the message is allocated; the second parameter is an integer denoting a local port. First of all the port type is checked in order to control if por is an output port. Then, if the mailbox connected to por is not full, the message is inserted in the mailbox, otherwise the sending process is blocked waiting for a free slot in the mailbox.

Also the function RECEIVE has two parameters: RECEIVE (mes,por) where mes denotes the message to be received and por the local port through which the message is received. The first parameter is a pointer containing, after the function execution, the address of the memory block in which the received message is allocated. First of all the port type is checked in order to control if por is an input port. Then, if the mailbox connected to por is not empty, a message is extracted from the mailbox; otherwise the

receiving process is blocked.

This message passing mechanism allows processes to know only about local names. Global variables are avoided. When a process sends a message through a port it does not know which process is the receiving one. This depends on the connections established during the creation of the software system. In such a way a module can be used as a component in many different configurations whithout changing its internal structure. This characteristic of PSL enhances the software modularity.

During the creation of a software system all the connections among processes must be established as well as the dimensions of all the mailboxes. In order to increment the flexibility of the message passing mechanism, connections are allowed in which a mailbox can be connected with many senders and many receivers.

3.2 Memory management

Memory in PSL is divided in two disjoined and differently handled parts:

Part 1: It is defined at confuration time. It has a fixed size and contains: nucleus data and code, code of the user modules, mailboxes, ports.

Part B: It is the rest of the whole memory and is managed by the nucleus, that allocates memory blocks (called user objects) to processes through the memory functions GET and RELEASE. This part contains: user objects, messages, free memory areas.

This section deals with the memory Part B.

Each PSL process (representative of an user module) may be given a memory area (namely, an user object) upon requesting it to the nucleus. Two kinds of request are allowed:

a) by a RECEIVE on a port
b) by a GET

Similarly, the user object can be released:

a) by a SEND to a port
b) by a RELEASE

As regards the memory management the effect of the SEND and RECEIVE functions is the transformation of a user object in a message and viceversa. In fact a producer process requires to the memory manager a free area (user object)

through the function GET, then fills it with the information to be sent and inserts this memory area, now containing the message, in the mailbox through the function SEND. On the other side the consumer process removes the message area from the mailbox through the function RECEIVE, then it takes away the information contained in this area, obtaining a free area (user object) that can be returned to the memory manager through the function RELEASE.

The effect of GET and RELEASE is then the creation or deletion of user objects, by extracting or putting pieces into the free (property of the nucleus) memory space.

In this section we discuss only about GET and RELEASE functions.

When a process needs some memory space, it can request it by issuing a GET (that behaves as a machine instruction of the virtual processor the process is running) specifying the requested size (as number of pages) of the user object. As a result of this function, the process can obtain, in its name space, an user object of the requested size. In a similar way, any process, by issuing a RELEASE, gets rid of anyone of the user objects it owns.

After a certain time since the start of the system, the part B of memory will be composed of a set of occupied areas (user objects and messages). The holes will also have different lenghts. At any time an N-pages object is requested, this request is <u>feasible</u> if a hole bigger than N pages does exsist; otherwise it is <u>unfeasible</u>.

The PSL takes care of memory requests as follows: before the start of the system, a size-ordered list of holes (LLP) is created, and initialized to the unique hole representing the free area of memory part B.

1- If the request is feasible: The hole of minimum size not less then N is searched for and is removed from the LLD list. If the hole size is greater than N, the hole is divided in two parts, one of which (N-sized) is given to the requesting process. The free part is linked to the LLD list.

2- If the request is unfeasible: The requesting process is swiched to a waiting state, and it is linked to the list of all the processes waiting for available memory space (LATM).

Every time an user object is released, a check is made if its memory space can be added to an adjacent hole to form an unique bigger hole. The resulting hole is to be linked to the LLD. If this hole is bigger than the maximum sized hole already in the list, a mechanism is triggered for possibly waking up some of the waiting processes.

The strategy is to switch from the waiting list to the ready list the highest priority processes having feasible

memory requests. Summarizing, there are the following problems:

- Handling of the hole list LLD
- Taking care of the possible union of adjacent holes after a release
- Handling of the process waiting list LATM

Before going on to discuss the different policies, it is convenient to make a remark: Some details of the chosen approach will depend on the particular available hardware structure, in our case a virtual machine generated by the IBM CP 367. The most important restriction is that the whole memory space is composed of 2K byte memory blocks (pages), each of them being the minimum memory unit protectable by a key.

3.2.1 <u>Handling the LLD list</u>

Two operations can be executed:

a) Linking a new hole into the list

b) Searching and removing an opportunely sized hole from the list.

When a GET N is being executed, a check is made if the request is feasible, by looking at the first element (the biggest hole) in the LLD list. If the request is feasible, then the list is scanned upward for the first hole whose size S is such that S>=N; then the hole is removed. If S=N then the LLD handling is finished; otherwise a S-N sized hole is created and linked to the list.

A quicker method would be to create a multiple list structure, each list being associated with a fixed hole size: in the case of no hole of a particular size, the header of the corresponding list will point to the first hole in the immediately larger hole list. Holes bigger than a prefixed size could be associated to an unique size-ordered list.

3.2.2 Compacting the holes

Whenever a process releases one of its user objects, the nucleus executes the following steps:

First a search is made in the LLD list to check if the memory space occupied by the user object is adjacent to a hole. If at least one adjacent hole is found (two is obviously the maximum), it is removed from the list: then a new hole is formed by the conjunction of the removed one

with the memory space of the object. If no adjacent hole exists, then the user object becomes a new hole. In any case the new hole is inserted into the list.

In order to speed this operation up, the holes are inserted in a second list, LLI, ordered by increasing memory addresses. Thus, since the address of the last released user object is available, both the addresses of the preceding and of the following holes in LLI are easily obtained.

3.2.3 Handling unfeasible requests

As mentioned above, after a given time since the start of the system a set of processes will be waiting for some memory space. When (after a release) a hole is made available that can satisfy some of the requests, it is necessary to follow a policy to establish which one(s) of the waiting processes has to be awakened. This policy must be conceived along with the policy of inserting processes into the waiting list LATM. Some different ordering criteria may be followed:

a) by the arrival time of the request

b) by static priority

c) by evaluated priority

c) by requested size

Since each process in PSL has associated a priority level, only b) and c) have been taken into account, being a) the particular case if all the processes have the same priority level.

The possible awaking policies are as follows:

1) The first process in the LLD list is awakened and, if its request is still unfeasible, no more processes are awakened.

2) The first process having a request not larger than the new hole is awakened.

In the current implementation, the awaking policy 2) has been adopted along with the insertion rule b) listed above. In this way, small request processes are not too heavily delayed, but hight priority and memory hungry processes can be delayed. A better solution would be reducing process priority as the request encreases, (insertion rule d and awaking policy 1).

3.3 Input/Output

As far as I/O devices are concerned, a unique function DOIO has been implemented to control the I/O operations. This function has three parameter DOIO (dev,com,sts) where dev identifies the peripheral device to be used, com the particular I/O command and sts the device status set at the end of the operation.

The notable point is that such an instruction does not only initiate the data transfer, but performs it to completion strictly according to the principle of sequential program execution. In this way side effects of automatic interruption are avoided. Interrupts imply execution of instruction sequences between any two program instructions and their occurrences can not be foreseen.

The function DOIO uses the hardware interrupt mechanism but avoids that interrupts appears to users. In such a way a process executes exactly the same sequence of actions corresponding to the program text. This characteristic helps in the static checking of programs.

For this purpose a process executing a DOIO is blocked and it is waked up only when the I/O command is terminated. This event is signaled through a normal program interrupt.

Obviously when a process is blocked the cpu is switched to another process. As a result the programmer is encouraged to decompose his program in those concurrent parts that are natural to its own inner logic instead of according to a system configuration typical for a particular installation.

3.4 Control Language Interpreter

The CLI executes a set of commands which allow the user to create software systems supported by 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

To design the CLI a solution has been chosen that allow to build the CLI just like a software system supported by PSL.

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 the their functions, into four subsets, namely:

1) Process creation commands.

2) Module connection commands.

3) Parameter assignment commands.

4) Control commands.

Typical command for any class, are the following:

CREATE PROCESS (name,module,priority)

in which the first parameter identifies the process unique name, the second identifies the program module that will become, during execution the sequential process specified by the first parameter, the third specifies the process priority.

CONNECT (module,port,mailbox)

that connects the local port identified by the second parameter, of the module identified by the first parameter, to the mailbox identified by the third parameter.

MAILBOX DIMENSION (mailbox,size)

that allocates a mailbox with the declared size.

START

This command asks the nucleus to execute the new software system.

A more detailed discussion of the CLI system is contained in a paper to be printed: "A modular and structured control language interpreter.

3.5 Conclusions

The Pisa software laboratory has been described. This laboratory is specially oriented to design new modular software systems. Furthermore the PSL allows to experiment with allocation strategies and guarantees full protection of PSL and modules.

3,6 References

/1/ R.M. Balzer PORTS-A metod for dynamic interprogram communication and job control Spring Joint Computer Conference. 1971.

/2/ J.B. Dennis Modularity Advanced Course on Software Engineering Ed. by F.L. Bauer. Springer-Verlag. 1973.

/3/ W. Corwin; W. Wulf SL 230 - A Software Laboratory - Intermediate report Carnagie Mellon University. Computer Science Department Report 1972.

/4/ D.L. Parnas Information Distribution Aspects of Design Methodology IFIP Proceedings. Ljubliana 1971.

/5/ D.L. Parnas On the criteria to be used in decomposing systems into modules Comm. ACM December 1972